Java

Java Programming Notes

WWW.LEARNLONER.COM

Contents

Chapter 1: Getting started with Java Language

Java SE Version	Code Name	End-of-life (free1)	Release Date
Java SE 10 (Early Access)	*None*	future	2018-03-20
Java SE 9	*None*	future	2017-07-27
Java SE 8	Spider	future	2014-03-18
Java SE 7	Dolphin	2015-04-14	2011-07-28
Java SE 6	Mustang	2013-04-16	2006-12-23
Java SE 5	Tiger	2009-11-04	2004-10-04
Java SE 1.4	Merlin	prior to 2009-11-04	2002-02-06
Java SE 1.3	Kestrel	prior to 2009-11-04	2000-05-08
Java SE 1.2	Playground	prior to 2009-11-04	1998-12-08
Java SE 1.1	*None*	prior to 2009-11-04	1997-02-19
Java SE 1.0	Oak	prior to 2009-11-04	1996-01-21

Section 1.1: Creating Your First Java Program

Create a new file in your text editor or IDE named `HelloWorld.java`. Then paste this code block into the file and save:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Run live on Ideone

Note: For Java to recognize this as a **`public class`** (and not throw a compile time error), the filename must be the same as the class name (`HelloWorld` in this example) with a `.java` extension. There should also be a **`public`** access modifier before it.

Naming conventions recommend that Java classes begin with an uppercase character, and be in camel case format (in which the first letter of each word is capitalized). The conventions recommend against underscores (_) and dollar signs ($).

To compile, open a terminal window and navigate to the directory of `HelloWorld.java`:

```
cd /path/to/containing/folder/
```

Note: cd is the terminal command to change directory.

Enter `javac` followed by the file name and extension as follows:

```
$ javac HelloWorld.java
```

It's fairly common to get the error `'javac' is not recognized as an internal or external command, operable program or batch file.` even when you have installed the `JDK` and are able to run the program from `IDE` ex. `eclipse` etc. Since the path is not added to the environment by default.

In case you get this on windows, to resolve, first try browsing to your `javac.exe` path, it's most probably in your `C:\Program Files\Java\jdk(version number)\bin`. Then try running it with below.

```
$ C:\Program Files\Java\jdk(version number)\bin\javac HelloWorld.java
```

Previously when we were calling `javac` it was same as above command. Only in that case your `OS` knew where `javac` resided. So let's tell it now, this way you don't have to type the whole path every-time. We would need to add this to our `PATH`

To edit the `PATH` environment variable in Windows XP/Vista/7/8/10:

- Control Panel ⇒ System ⇒ Advanced system settings
- Switch to "Advanced" tab ⇒ Environment Variables
- In "System Variables", scroll down to select "PATH" ⇒ Edit

You cannot undo this so be careful. First copy your existing path to notepad. Then to get the exact PATH to your `javac` browse manually to the folder where `javac` resides and click on the address bar and then copy it. It should look something like `c:\Program Files\Java\jdk1.8.0_xx\bin`

In "Variable value" field, paste this **IN FRONT** of all the existing directories, followed by a semi-colon (;). **DO NOT DELETE** any existing entries.

```
Variable name  : PATH
Variable value : c:\Program Files\Java\jdk1.8.0_xx\bin;[Existing Entries...]
```

Now this should resolve.

For Linux Based systems try here.

Note: *The `javac` command invokes the Java compiler.*

The compiler will then generate a bytecode file called `HelloWorld.class` which can be executed in the Java Virtual Machine (JVM). The Java programming language compiler, `javac`, reads source files written in the Java programming language and compiles them into `bytecode` class files. Optionally, the compiler can also process annotations found in source and class files using the `Pluggable` Annotation Processing API. The compiler is a command line tool but can also be invoked using the Java Compiler API.

To run your program, enter `java` followed by the name of the class which contains the `main` method (`HelloWorld` in our example). Note how the `.class` is omitted:

```
$ java HelloWorld
```

Note: *The `java` command runs a Java application.*

This will output to your console:

> Hello, World!

You have successfully coded and built your very first Java program!

Note: In order for Java commands (`java`, `javac`, etc) to be recognized, you will need to make sure:

- A JDK is installed (e.g. Oracle, OpenJDK and other sources)

- Your environment variables are properly set up

You will need to use a compiler (`javac`) and an executor (`java`) provided by your JVM. To find out which versions you have installed, enter `java -version` and `javac -version` on the command line. The version number of your program will be printed in the terminal (e.g. `1.8.0_73`).

A closer look at the Hello World program

The "Hello World" program contains a single file, which consists of a `HelloWorld` class definition, a `main` method, and a statement inside the `main` method.

```
public class HelloWorld {
```

The `class` keyword begins the class definition for a class named `HelloWorld`. Every Java application contains at least one class definition (Further information about classes).

```
    public static void main(String[] args)  {
```

This is an entry point method (defined by its name and signature of `public static void main(String[])`) from which the `JVM` can run your program. Every Java program should have one. It is:

- `public`: meaning that the method can be called from anywhere mean from outside the program as well. See Visibility for more information on this.
- `static`: meaning it exists and can be run by itself (at the class level without creating an object).
- `void`: meaning it returns no value. ***Note:*** *This is unlike C and C++ where a return code such as* `int` *is expected (Java's way is* `System.exit()`*).*

This main method accepts:

- An array (typically called `args`) of `String`s passed as arguments to main function (e.g. from command line arguments).

Almost all of this is required for a Java entry point method.

Non-required parts:

- The name `args` is a variable name, so it can be called anything you want, although it is typically called `args`.
- Whether its parameter type is an array (`String[] args`) or Varargs (`String... args`) does not matter because arrays can be passed into varargs.

Note: A single application may have multiple classes containing an entry point (`main`) method. The entry point of the application is determined by the class name passed as an argument to the `java` command.

Inside the main method, we see the following statement:

```
        System.out.println("Hello, World!");
```

Let's break down this statement element-by-element:

Element	Purpose
`System`	this denotes that the subsequent expression will call upon the `System` class, from the `java.lang` package.

`.`	this is a "dot operator". Dot operators provide you access to a classes members1; i.e. its fields (variables) and its methods. In this case, this dot operator allows you to reference the `out` static field within the `System` class.
`out`	this is the name of the static field of `PrintStream` type within the `System` class containing the standard output functionality.
`.`	this is another dot operator. This dot operator provides access to the `println` method within the `out` variable.
`println`	this is the name of a method within the PrintStream class. This method in particular prints the contents of the parameters into the console and inserts a newline after.
`(`	this parenthesis indicates that a method is being accessed (and not a field) and begins the parameters being passed into the `println` method.
`"Hello, World!"`	this is the String literal that is passed as a parameter, into the `println` method. The double quotation marks on each end delimit the text as a String.
`)`	this parenthesis signifies the closure of the parameters being passed into the `println` method.
`;`	this semicolon marks the end of the statement.

Note: *Each statement in Java must end with a semicolon (`;`).*

The method body and class body are then closed.

```
    }  // end of main function scope
}      // end of class HelloWorld scope
```

Here's another example demonstrating the OO paradigm. Let's model a football team with one (yes, one!) member. There can be more, but we'll discuss that when we get to arrays.

First, let's define our `Team` class:

```
public class Team {
    Member member;
    public Team(Member member) {  // who is in this Team?
        this.member = member;  //  one 'member' is in this Team!
    }
}
```

Now, let's define our `Member` class:

```
class Member {
    private String name;
    private String type;
    private int level; // note the data type here
    private int rank; // note the data type here as well

    public Member(String name, String type, int level, int rank) {
        this.name = name;
        this.type = type;
        this.level = level;
        this.rank = rank;
    }
}
```

Why do we use **`private`** here? Well, if someone wanted to know your name, they should ask you directly, instead of reaching into your pocket and pulling out your Social Security card. This **`private`** does something like that: it prevents outside entities from accessing your variables. You can only return **`private`** members through getter functions (shown below).

After putting it all together, and adding the getters and main method as discussed before, we have:

```
public class Team {
    Member member;
    public Team(Member member) {
        this.member = member;
    }

    // here's our main method
    public static void main(String[] args) {
       Member myMember = new Member("Aurieel", "light", 10, 1);
       Team myTeam = new Team(myMember);
       System.out.println(myTeam.member.getName());
       System.out.println(myTeam.member.getType());
       System.out.println(myTeam.member.getLevel());
       System.out.println(myTeam.member.getRank());
    }
}

class Member {
    private String name;
    private String type;
    private int level;
    private int rank;

    public Member(String name, String type, int level, int rank) {
        this.name = name;
        this.type = type;
        this.level = level;
        this.rank = rank;
    }

    /* let's define our getter functions here */
    public String getName() { // what is your name?
        return this.name; // my name is ...
    }

    public String getType() { // what is your type?
        return this.type; // my type is ...
    }

    public int getLevel() { // what is your level?
        return this.level; // my level is ...
    }

    public int getRank() { // what is your rank?
        return this.rank; // my rank is
    }
}
```

Output:

```
Aurieel
light
10
1
```

Run on ideone

Once again, the `main` method inside the `Test` class is the entry point to our program. Without the `main` method, we cannot tell the Java Virtual Machine (JVM) from where to begin execution of the program.

1 - Because the `HelloWorld` class has little relation to the `System` class, it can only access **`public`** data.

Chapter 2: Type Conversion

Section 2.1: Numeric primitive casting

Numeric primitives can be cast in two ways. *Implicit* casting happens when the source type has smaller range than the target type.

```
//Implicit casting
byte byteVar = 42;
short shortVar = byteVar;
int intVar = shortVar;
long longVar = intvar;
float floatVar = longVar;
double doubleVar = floatVar;
```

Explicit casting has to be done when the source type has larger range than the target type.

```
//Explicit casting
double doubleVar = 42.0d;
float floatVar = (float) doubleVar;
long longVar = (long) floatVar;
int intVar = (int) longVar;
short shortVar = (short) intVar;
byte byteVar = (byte) shortVar;
```

When casting floating point primitives (`float`, `double`) to whole number primitives, the number is **rounded down**.

Section 2.2: Basic Numeric Promotion

```
 static void testNumericPromotion() {

    char char1 = 1, char2 = 2;
    short short1 = 1, short2 = 2;
    int int1 = 1, int2 = 2;
    float float1 = 1.0f, float2 = 2.0f;

    // char1 = char1 + char2;      // Error: Cannot convert from int to char;
    // short1 = short1 + short2;   // Error: Cannot convert from int to short;
    int1 = char1 + char2;          // char is promoted to int.
    int1 = short1 + short2;        // short is promoted to int.
    int1 = char1 + short2;         // both char and short promoted to int.
    float1 = short1 + float2;      // short is promoted to float.
    int1 = int1 + int2;            // int is unchanged.
}
```

Section 2.3: Non-numeric primitive casting

The `boolean` type cannot be cast to/from any other primitive type.

A `char` can be cast to/from any numeric type by using the code-point mappings specified by Unicode. A `char` is represented in memory as an unsigned 16-bit integer value (2 bytes), so casting to `byte` (1 byte) will drop 8 of those bits (this is safe for ASCII characters). The utility methods of the `Character` class use `int` (4 bytes) to transfer to/from code-point values, but a `short` (2 bytes) would also suffice for storing a Unicode code-point.

```
int badInt   = (int)  true; // Compiler error: incompatible types
```

```
char char1   = (char)   65; // A
byte byte1   = (byte)  'A'; // 65
short short1 = (short) 'A'; // 65
int int1     = (int)   'A'; // 65

char char2   = (char) 8253; // ‽
byte byte2   = (byte)  '‽'; // 61 (truncated code-point into the ASCII range)
short short2 = (short) '‽'; // 8253
int int2     = (int)   '‽'; // 8253
```

Section 2.4: Object casting

As with primitives, objects can be cast both explicitly and implicitly.

Implicit casting happens when the source type extends or implements the target type (casting to a superclass or interface).

Explicit casting has to be done when the source type is extended or implemented by the target type (casting to a subtype). This can produce a runtime exception (`ClassCastException`) when the object being cast is not of the target type (or the target's subtype).

```
Float floatVar = new Float(42.0f);
Number n = floatVar;                //Implicit (Float implements Number)
Float floatVar2 = (Float) n;        //Explicit
Double doubleVar = (Double) n;      //Throws exception (the object is not Double)
```

Section 2.5: Testing if an object can be cast using instanceof

Java provides the **`instanceof`** operator to test if an object is of a certain type, or a subclass of that type. The program can then choose to cast or not cast that object accordingly.

```
Object obj = Calendar.getInstance();
long time = 0;

if(obj instanceof Calendar)
{
    time = ((Calendar)obj).getTime();
}
if(obj instanceof Date)
{
    time = ((Date)obj).getTime(); // This line will never be reached, obj is not a Date type.
}
```

Chapter 3: Getters and Setters

This article discusses getters and setters; the standard way to provide access to data in Java classes.

Section 3.1: Using a setter or getter to implement a constraint

Setters and Getters allow for an object to contain private variables which can be accessed and changed with restrictions. For example,

```
public class Person {

    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        if(name!=null && name.length()>2)
           this.name = name;
    }
}
```

In this `Person` class, there is a single variable: `name`. This variable can be accessed using the `getName()` method and changed using the `setName(String)` method, however, setting a name requires the new name to have a length greater than 2 characters and to not be null. Using a setter method rather than making the variable `name` public allows others to set the value of `name` with certain restrictions. The same can be applied to the getter method:

```
public String getName(){
   if(name.length()>16)
      return "Name is too large!";
   else
      return name;
}
```

In the modified `getName()` method above, the `name` is returned only if its length is less than or equal to 16. Otherwise, `"Name is too large"` is returned. This allows the programmer to create variables that are reachable and modifiable however they wish, preventing client classes from editing the variables unwantedly.

Section 3.2: Why Use Getters and Setters?

Consider a basic class containing an object with getters and setters in Java:

```
public class CountHolder {
  private int count = 0;

  public int getCount() { return count; }
  public void setCount(int c) { count = c; }
}
```

We can't access the `count` variable because it's private. But we can access the `getCount()` and the `setCount(int)` methods because they are public. To some, this might raise the question; why introduce the middleman? Why not just simply make they count public?

```
public class CountHolder {
```

```
  public int count = 0;
}
```

For all intents and purposes, these two are exactly the same, functionality-wise. The difference between them is the extensibility. Consider what each class says:

- **First**: "I have a method that will give you an `int` value, and a method that will set that value to another `int`".
- **Second**: "I have an `int` that you can set and get as you please."

These might sound similar, but the first is actually much more guarded in its nature; it only lets you interact with its internal nature as **it** dictates. This leaves the ball in its court; it gets to choose how the internal interactions occur. The second has exposed its internal implementation externally, and is now not only prone to external users, but, in the case of an API, **committed** to maintaining that implementation (or otherwise releasing a non-backward-compatible API).

Lets consider if we want to synchronize access to modifying and accessing the count. In the first, this is simple:

```
public class CountHolder {
  private int count = 0;

  public synchronized int getCount() { return count; }
  public synchronized void setCount(int c) { count = c; }
}
```

but in the second example, this is now nearly impossible without going through and modifying each place where the `count` variable is referenced. Worse still, if this is an item that you're providing in a library to be consumed by others, you do **not** have a way of performing that modification, and are forced to make the hard choice mentioned above.

So it begs the question; are public variables ever a good thing (or, at least, not evil)?

I'm unsure. On one hand, you can see examples of public variables that have stood the test of time (IE: the `out` variable referenced in `System.out`). On the other, providing a public variable gives no benefit outside of extremely minimal overhead and potential reduction in wordiness. My guideline here would be that, if you're planning on making a variable public, you should judge it against these criteria with **extreme** prejudice:

1. The variable should have no conceivable reason to **ever** change in its implementation. This is something that's extremely easy to screw up (and, even if you do get it right, requirements can change), which is why getters/setters are the common approach. If you're going to have a public variable, this really needs to be thought through, especially if released in a library/framework/API.
2. The variable needs to be referenced frequently enough that the minimal gains from reducing verbosity warrants it. I don't even think the overhead for using a method versus directly referencing should be considered here. It's far too negligible for what I'd conservatively estimate to be 99.9% of applications.

There's probably more than I haven't considered off the top of my head. If you're ever in doubt, always use getters/setters.

Section 3.3: Adding Getters and Setters

Encapsulation is a basic concept in OOP. It is about wrapping data and code as a single unit. In this case, it is a good practice to declare the variables as **`private`** and then access them through `Getters` and `Setters` to view and/or modify them.

```
public class Sample {
```

```
  private String  name;
  private int age;

  public int getAge() {
    return age;
  }

  public void setAge(int age) {
    this.age = age;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
}
```

These private variables cannot be accessed directly from outside the class. Hence they are protected from unauthorized access. But if you want to view or modify them, you can use Getters and Setters.

`getXxx()` method will return the current value of the variable `xxx`, while you can set the value of the variable `xxx` using `setXxx()`.

The naming convention of the methods are (in example variable is called `variableName`):

- All non **`boolean`** variables

```
getVariableName()   //Getter, The variable name should start with uppercase
setVariableName(..) //Setter, The variable name should start with uppercase
```

- **`boolean`** variables

```
 isVariableName()     //Getter, The variable name should start with uppercase
 setVariableName(...) //Setter, The variable name should start with uppercase
```

Public Getters and Setters are part of the Property definition of a Java Bean.

Chapter 4: Reference Data Types

Section 4.1: Dereferencing

Dereferencing happens with the `.` operator:

```
Object obj = new Object();
String text = obj.toString(); // 'obj' is dereferenced.
```

Dereferencing *follows* the memory address stored in a reference, to the place in memory where the actual object resides. When an object has been found, the requested method is called (`toString` in this case).

When a reference has the value **`null`**, dereferencing results in a NullPointerException:

```
Object obj = null;
obj.toString(); // Throws a NullpointerException when this statement is executed.
```

`null` indicates the absence of a value, i.e. *following* the memory address leads nowhere. So there is no object on which the requested method can be called.

Section 4.2: Instantiating a reference type

```
Object obj = new Object(); // Note the 'new' keyword
```

Where:

- `Object` is a reference type.
- `obj` is the variable in which to store the new reference.
- `Object()` is the call to a constructor of `Object`.

What happens:

- Space in memory is allocated for the object.
- The constructor `Object()` is called to initialize that memory space.
- The memory address is stored in `obj`, so that it *references* the newly created object.

This is different from primitives:

```
int i = 10;
```

Where the actual value `10` is stored in `i`.

Chapter 5: Java Compiler - 'javac'

Section 5.1: The 'javac' command - getting started

Simple example

Assuming that the "HelloWorld.java" contains the following Java source:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

(For an explanation of the above code, please refer to Getting started with Java Language .)

We can compile the above file using this command:

```
$ javac HelloWorld.java
```

This produces a file called "HelloWorld.class", which we can then run as follows:

```
$ java HelloWorld
Hello world!
```

The key points to note from this example are:

1. The source filename "HelloWorld.java" must match the class name in the source file ... which is `HelloWorld`. If they don't match, you will get a compilation error.
2. The bytecode filename "HelloWorld.class" corresponds to the classname. If you were to rename the "HelloWorld.class", you would get an error when your tried to run it.
3. When running a Java application using `java`, you supply the classname NOT the bytecode filename.

Example with packages

Most practical Java code uses packages to organize the namespace for classes and reduce the risk of accidental class name collision.

If we wanted to declare the `HelloWorld` class in a package call `com.example`, the "HelloWorld.java" would contain the following Java source:

```
package com.example;

public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

This source code file needs to stored in a directory tree whose structure corresponds to the package naming.

```
.    # the current directory (for this example)
|
 ----com
     |
```

```
      ----example
          |
           ----HelloWorld.java
```

We can compile the above file using this command:

```
$ javac com/example/HelloWorld.java
```

This produces a file called "com/example/HelloWorld.class"; i.e. after compilation, the file structure should look like this:

```
.    # the current directory (for this example)
|
 ----com
     |
      ----example
          |
           ----HelloWorld.java
           ----HelloWorld.class
```

We can then run the application as follows:

```
$ java com.example.HelloWorld
Hello world!
```

Additional points to note from this example are:

1. The directory structure must match the package name structure.
2. When you run the class, the full class name must be supplied; i.e. "com.example.HelloWorld" not "HelloWorld".
3. You don't have to compile and run Java code out of the current directory. We are just doing it here for illustration.

Compiling multiple files at once with 'javac'.

If your application consists of multiple source code files (and most do!) you can compile them one at a time. Alternatively, you can compile multiple files at the same time by listing the pathnames:

```
$ javac Foo.java Bar.java
```

or using your command shell's filename wildcard functionality ....

```
$ javac *.java
$ javac com/example/*.java
$ javac */**/*.java #Only works on Zsh or with globstar enabled on your shell
```

This will compile all Java source files in the current directory, in the "com/example" directory, and recursively in child directories respectively. A third alternative is to supply a list of source filenames (and compiler options) as a file. For example:

```
$ javac @sourcefiles
```

where the `sourcefiles` file contains:

```
Foo.java
Bar.java
```

```
com/example/HelloWorld.java
```

Note: compiling code like this is appropriate for small one-person projects, and for once-off programs. Beyond that, it is advisable to select and use a Java build tool. Alternatively, most programmers use a Java IDE (e.g. NetBeans, eclipse, IntelliJ IDEA) which offers an embedded compiler and incremental building of "projects".

Commonly used 'javac' options

Here are a few options for the `javac` command that are likely to be useful to you

- The `-d` option sets a destination directory for writing the ".class" files.
- The `-sourcepath` option sets a source code search path.
- The `-cp` or `-classpath` option sets the search path for finding external and previously compiled classes. For more information on the classpath and how to specify it, refer to the The Classpath Topic.
- The `-version` option prints the compiler's version information.

A more complete list of compiler options will be described in a separate example.

References

The definitive reference for the `javac` command is the Oracle manual page for `javac`.

Section 5.2: Compiling for a different version of Java

The Java programming language (and its runtime) has undergone numerous changes since its release since its initial public release. These changes include:

- Changes in the Java programming language syntax and semantics
- Changes in the APIs provided by the Java standard class libraries.
- Changes in the Java (bytecode) instruction set and classfile format.

With very few exceptions (for example the **`enum`** keyword, changes to some "internal" classes, etc), these changes are backwards compatible.

- A Java program that was compiled using an older version of the Java toolchain will run on a newer version Java platform without recompilation.
- A Java program that was written in an older version of Java will compile successfully with a new Java compiler.

Compiling old Java with a newer compiler

If you need to (re-)compile older Java code on a newer Java platform to run on the newer platform, you generally don't need to give any special compilation flags. In a few cases (e.g. if you had used **`enum`** as an identifier) you could use the `-source` option to disable the new syntax. For example, given the following class:

```
public class OldSyntax {
    private static int enum;  // invalid in Java 5 or later
}
```

the following is required to compile the class using a Java 5 compiler (or later):

```
$ javac -source 1.4 OldSyntax.java
```

Compiling for an older execution platform

If you need to compile Java to run on an older Java platforms, the simplest approach is to install a JDK for the oldest

version you need to support, and use that JDK's compiler in your builds.

You can also compile with a newer Java compiler, but there are complicated. First of all, there some important preconditions that must be satisfied:

- The code you are compiling must not use Java language constructs that were not available in the version of Java that you are targeting.
- The code must not depend on standard Java classes, fields, methods and so on that were not available in the older platforms.
- Third party libraries that the code depends must also be built for the older platform and available at compile-time and run-time.

Given the preconditions are met, you can recompile code for an older platform using the `-target` option. For example,

```
$ javac -target 1.4 SomeClass.java
```

will compile the above class to produce bytecodes that are compatible with Java 1.4 or later JVM. (In fact, the `-source` option implies a compatible `-target`, so `javac -source 1.4 ...` would have the same effect. The relationship between `-source` and `-target` is described in the Oracle documentation.)

Having said that, if you simply use `-target` or `-source`, you will still be compiling against the standard class libraries provided by the compiler's JDK. If you are not careful, you can end up with classes with the correct bytecode version, but with dependencies on APIs that are not available. The solution is to use the `-bootclasspath` option. For example:

```
$ javac -target 1.4 --bootclasspath path/to/java1.4/rt.jar SomeClass.java
```

will compile against an alternative set of runtime libraries. If the class being compiled has (accidental) dependencies on newer libraries, this will give you compilation errors.

Chapter 6: Documenting Java Code

Documentation for java code is often generated using javadoc. Javadoc was created by Sun Microsystems for the purpose of generating API documentation in HTML format from java source code. Using the HTML format gives the convenience of being able to hyperlink related documents together.

Section 6.1: Building Javadocs From the Command Line

Many IDEs provide support for generating HTML from Javadocs automatically; some build tools (Maven and Gradle, for example) also have plugins that can handle the HTML creation.

However, these tools are not required to generate the Javadoc HTML; this can be done using the command line `javadoc` tool.

The most basic usage of the tool is:

```
javadoc JavaFile.java
```

Which will generate HTML from the Javadoc comments in `JavaFile.java`.

A more practical use of the command line tool, which will recursively read all java files in `[source-directory]`, create documentation for `[package.name]` and all sub-packages, and place the generated HTML in the `[docs-directory]` is:

```
javadoc -d [docs-directory] -subpackages -sourcepath [source-directory] [package.name]
```

Section 6.2: Class Documentation

All Javadoc comments begin with a block comment followed by an asterisk (`/**`) and end when the block comment does (`*/`). Optionally, each line can begin with arbitrary whitespace and a single asterisk; these are ignored when the documentation files are generated.

```
/**
 * Brief summary of this class, ending with a period.
 *
 * It is common to leave a blank line between the summary and further details.
 * The summary (everything before the first period) is used in the class or package
 * overview section.
 *
 * The following inline tags can be used (not an exhaustive list):
 * {@link some.other.class.Documentation} for linking to other docs or symbols
 * {@link some.other.class.Documentation Some Display Name} the link's appearance can be
 * customized by adding a display name after the doc or symbol locator
 * {@code code goes here} for formatting as code
 * {@literal <>[]()foo} for interpreting literal text without converting to HTML markup
 * or other tags.
 *
 * Optionally, the following tags may be used at the end of class documentation
 * (not an exhaustive list):
 *
 * @author John Doe
 * @version 1.0
 * @since 5/10/15
 * @see some.other.class.Documentation
 * @deprecated This class has been replaced by some.other.package.BetterFileReader
 *
```

```
 * You can also have custom tags for displaying additional information.
 * Using the @custom.<NAME> tag and the -tag custom.<NAME>:htmltag:"context"
 * command line option, you can create a custom tag.
 *
 * Example custom tag and generation:
 * @custom.updated 2.0
 * Javadoc flag: -tag custom.updated:a:"Updated in version:"
 * The above flag will display the value of @custom.updated under "Updated in version:"
 *
 */
public class FileReader {
}
```

The same tags and format used for `Classes` can be used for `Enums` and `Interfaces` as well.

Section 6.3: Method Documentation

All Javadoc comments begin with a block comment followed by an asterisk (`/**`) and end when the block comment does (`*/`). Optionally, each line can begin with arbitrary whitespace and a single asterisk; these are ignored when the documentation files are generated.

```
/**
 * Brief summary of method, ending with a period.
 *
 * Further description of method and what it does, including as much detail as is
 * appropriate.  Inline tags such as
 * {@code code here}, {@link some.other.Docs}, and {@literal text here} can be used.
 *
 * If a method overrides a superclass method, {@inheritDoc} can be used to copy the
 * documentation
 * from the superclass method
 *
 * @param stream Describe this parameter.  Include as much detail as is appropriate
 *               Parameter docs are commonly aligned as here, but this is optional.
 *               As with other docs, the documentation before the first period is
 *               used as a summary.
 *
 * @return Describe the return values.  Include as much detail as is appropriate
 *         Return type docs are commonly aligned as here, but this is optional.
 *         As with other docs, the documentation before the first period is used as a
 *         summary.
 *
 * @throws IOException Describe when and why this exception can be thrown.
 *                     Exception docs are commonly aligned as here, but this is
 *                     optional.
 *                     As with other docs, the documentation before the first period
 *                     is used as a summary.
 *                     Instead of @throws, @exception can also be used.
 *
 * @since 2.1.0
 * @see some.other.class.Documentation
 * @deprecated  Describe why this method is outdated. A replacement can also be specified.
 */
public String[] read(InputStream stream) throws IOException {
    return null;
}
```

Section 6.4: Package Documentation

Version ≥ Java SE 5

It is possible to create package-level documentation in Javadocs using a file called `package-info.java`. This file must be formatted as below. Leading whitespace and asterisks optional, typically present in each line for formatting reason

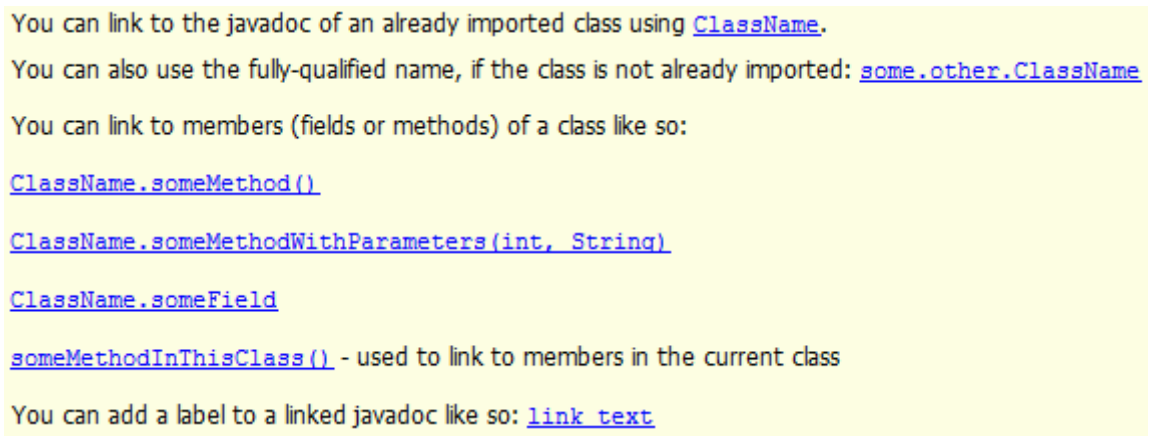
```
/**
 * Package documentation goes here; any documentation before the first period will
 * be used as a summary.
 *
 * It is common practice to leave a blank line between the summary and the rest
 * of the documentation; use this space to describe the package in as much detail
 * as is appropriate.
 *
 * Inline tags such as {@code code here}, {@link reference.to.other.Documentation},
 * and {@literal text here} can be used in this documentation.
 */
package com.example.foo;

// The rest of the file must be empty.
```

In the above case, you must put this file `package-info.java` inside the folder of the Java package `com.example.foo`.

Section 6.5: Links

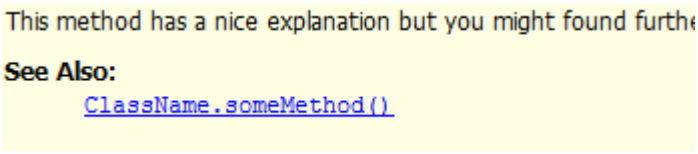
Linking to other Javadocs is done with the `@link` tag:

```
/**
 * You can link to the javadoc of an already imported class using {@link ClassName}.
 *
 * You can also use the fully-qualified name, if the class is not already imported:
 *  {@link some.other.ClassName}
 *
 * You can link to members (fields or methods) of a class like so:
 *  {@link ClassName#someMethod()}
 *  {@link ClassName#someMethodWithParameters(int, String)}
 *  {@link ClassName#someField}
 *  {@link #someMethodInThisClass()} - used to link to members in the current class
 *
 * You can add a label to a linked javadoc like so:
 *  {@link ClassName#someMethod() link text}
 */
```



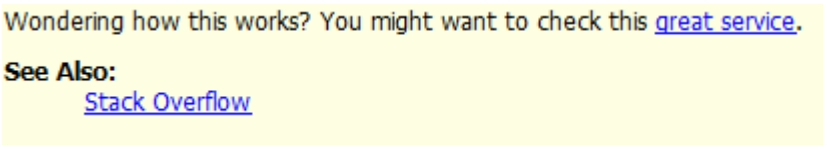
With the `@see` tag you can add elements to the *See also* section. Like `@param` or `@return` the place where they appear is not relevant. The spec says you should write it after `@return`.

```
/**
 * This method has a nice explanation but you might found further
 * information at the bottom.
 *
 * @see ClassName#someMethod()
 */
```



If you want to add **links to external resources** you can just use the HTML `<a>` tag. You can use it inline anywhere or inside both `@link` and `@see` tags.

```
/**
 * Wondering how this works? You might want
 * to check this <a href="http://stackoverflow.com/">great service</a>.
 *
 * @see <a href="http://stackoverflow.com/">Stack Overflow</a>
 */
```



Section 6.6: Code snippets inside documentation

The canonical way of writing code inside documentation is with the `{@code }` construct. If you have multiline code wrap inside `<pre></pre>`.

```
/**
 * The Class TestUtils.
 * <p>
 * This is an {@code inline("code example")}.
 * <p>
 * You should wrap it in pre tags when writing multiline code.
 * <pre>{@code
 *  Example example1 = new FirstLineExample();
 *  example1.butYouCanHaveMoreThanOneLine();
 * }</pre>
 * <p>
 * Thanks for reading.
 */
class TestUtils {
```

Sometimes you may need to put some complex code inside the javadoc comment. The `@` sign is specially problematic. The use of the old `<code>` tag alongside the `{@literal }` construct solves the problem.

```
/**
 * Usage:
 * <pre><code>
 * class SomethingTest {
 * {@literal @}Rule
```

```
 *  public SingleTestRule singleTestRule = new SingleTestRule("test1");
 *
 * {@literal @}Test
 *  public void test1() {
 *      // only this test will be executed
 *  }
 *
 *  ...
 * }
 * </code></pre>
 */
class SingleTestRule implements TestRule { }
```

Section 6.7: Field Documentation

All Javadoc comments begin with a block comment followed by an asterisk (`/**`) and end when the block comment does (`*/`). Optionally, each line can begin with arbitrary whitespace and a single asterisk; these are ignored when the documentation files are generated.

```
/**
 * Fields can be documented as well.
 *
 * As with other javadocs, the documentation before the first period is used as a
 * summary, and is usually separated from the rest of the documentation by a blank
 * line.
 *
 * Documentation for fields can use inline tags, such as:
 * {@code code here}
 * {@literal text here}
 * {@link other.docs.Here}
 *
 * Field documentation can also make use of the following tags:
 *
 * @since 2.1.0
 * @see some.other.class.Documentation
 * @deprecated Describe why this field is outdated
 */
public static final String CONSTANT_STRING = "foo";
```

Section 6.8: Inline Code Documentation

Apart from the Javadoc documentation code can be documented inline.

Single Line comments are started by `//` and may be positioned after a statement on the same line, but not before.

```
public void method() {

  //single line comment
  someMethodCall(); //single line comment after statement

}
```

Multi-Line comments are defined between `/*` and `*/`. They can span multiple lines and may even been positioned between statements.

```
public void method(Object object) {

  /*
```

```
    multi
    line
    comment
  */
  object/*inner-line-comment*/.method();
}
```

JavaDocs are a special form of multi-line comments, starting with `/**`.

As too many inline comments may decrease readability of code, they should be used sparsely in case the code isn't self-explanatory enough or the design decision isn't obvious.

An additional use case for single-line comments is the use of TAGs, which are short, convention driven keywords. Some development environments recognize certain conventions for such single-comments. Common examples are

- `//TODO`
- `//FIXME`

Or issue references, i.e. for Jira

- `//PRJ-1234`

Chapter 7: Command line Argument Processing

Parameter	Details
args	The command line arguments. Assuming that the `main` method is invoked by the Java launcher, `args` will be non-null, and will have no **`null`** elements.

Section 7.1: Argument processing using GWT ToolBase

If you want to parse more complex command-line arguments, e.g. with optional parameters, than the best is to use google's GWT approach. All classes are public available at:

https://gwt.googlesource.com/gwt/+/2.8.0-beta1/dev/core/src/com/google/gwt/util/tools/ToolBase.java

An example for handling the command-line `myprogram -dir "~/Documents" -port 8888` is:

```
public class MyProgramHandler extends ToolBase {
   protected File dir;
   protected int port;
   // getters for dir and port
   ...

   public MyProgramHandler() {
       this.registerHandler(new ArgHandlerDir() {
            @Override
            public void setDir(File dir) {
                this.dir = dir;
            }
       });
       this.registerHandler(new ArgHandlerInt() {
            @Override
            public String[] getTagArgs() {
               return new String[]{"port"};
            }
            @Override
            public void setInt(int value) {
               this.port = value;
            }
       });
   }
   public static void main(String[] args) {
      MyProgramHandler myShell = new MyProgramHandler();
      if (myShell.processArgs(args)) {
         // main program operation
         System.out.println(String.format("port: %d; dir: %s",
            myShell.getPort(), myShell.getDir()));
      }
      System.exit(1);
   }
}
```

`ArgHandler` also has a method `isRequired()` which can be overwritten to say that the command-line argument is required (default return is **`false`** so that the argument is optional.

Section 7.2: Processing arguments by hand

When the command-line syntax for an application is simple, it is reasonable to do the command argument

processing entirely in custom code.

In this example, we will present a series of simple case studies. In each case, the code will produce error messages if the arguments are unacceptable, and then call `System.exit(1)` to tell the shell that the command has failed. (We will assume in each case that the Java code is invoked using a wrapper whose name is "myapp".)

A command with no arguments

In this case-study, the command requires no arguments. The code illustrates that `args.length` gives us the number of command line arguments.

```
public class Main {
    public static void main(String[] args) {
        if (args.length > 0) {
            System.err.println("usage: myapp");
            System.exit(1);
        }
        // Run the application
        System.out.println("It worked");
    }
}
```

A command with two arguments

In this case-study, the command requires at precisely two arguments.

```
public class Main {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.err.println("usage: myapp <arg1> <arg2>");
            System.exit(1);
        }
        // Run the application
        System.out.println("It worked: " + args[0] + ", " + args[1]);
    }
}
```

Note that if we neglected to check `args.length`, the command would crash if the user ran it with too few command-line arguments.

A command with "flag" options and at least one argument

In this case-study, the command has a couple of (optional) flag options, and requires at least one argument after the options.

```
package tommy;
public class Main {
    public static void main(String[] args) {
        boolean feelMe = false;
        boolean seeMe = false;
        int index;
        loop: for (index = 0; index < args.length; index++) {
            String opt = args[index];
            switch (opt) {
            case "-c":
               seeMe = true;
               break;
            case "-f":
```

```
                feelMe = true;
                break;
            default:
                if (!opts.isEmpty() && opts.charAt(0) == '-') {
                    error("Unknown option: '" + opt + "'");
                }
                break loop;
            }
        }
        if (index >= args.length) {
            error("Missing argument(s)");
        }

        // Run the application
        // ...
    }

    private static void error(String message) {
        if (message != null) {
            System.err.println(message);
        }
        System.err.println("usage: myapp [-f] [-c] [ <arg> ...]");
        System.exit(1);
    }
}
```

As you can see, processing the arguments and options gets rather cumbersome if the command syntax is complicated. It is advisable to use a "command line parsing" library; see the other examples.

Chapter 8: The Java Command - 'java' and 'javaw'

Section 8.1: Entry point classes

A Java entry-point class has a `main` method with the following signature and modifiers:

```
public static void main(String[] args)
```

> Sidenote: because of how arrays work, it can also be `(String args[])`

When the `java` command starts the virtual machine, it loads the specified entry-point classes and tries to find `main`. If successful, the arguments from command line are converted to Java `String` objects and assembled into an array. If `main` is invoked like this, the array will *not* be **`null`** and won't contain any **`null`** entries.

A valid entry-point class method must do the following:

- Be named `main` (case-sensitive)
- Be **`public`** and **`static`**
- Have a **`void`** return type
- Have a single argument with an array `String[]`. The argument must be present and no more than one argument is allowed.
- Be generic: type parameters are not allowed.
- Have a non-generic, top-level (not nested or inner) enclosing class

It is conventional to declare the class as **`public`** but this not strictly necessary. From Java 5 onward, the `main` method's argument type may be a `String` varargs instead of a string array. `main` can optionally throw exceptions, and its parameter can be named anything, but conventionally it is `args`.

JavaFX entry-points

From Java 8 onwards the `java` command can also directly launch a JavaFX application. JavaFX is documented in the JavaFX tag, but a JavaFX entry-point must do the following:

- Extend `javafx.application.Application`
- Be **`public`** and not **`abstract`**
- Not be generic or nested
- Have an explicit or implicit **`public`** no-args constructor

Section 8.2: Troubleshooting the 'java' command

This example covers common errors with using the 'java' command.

"Command not found"

If you get an error message like:

```
java: command not found
```

when trying to run the `java` command, this means that there is no `java` command on your shell's command search path. The cause could be:

- you don't have a Java JRE or JDK installed at all,
- you have not updated the `PATH` environment variable (correctly) in your shell initialization file, or
- you have not "sourced" the relevant initialization file in the current shell.

Refer to "Installing Java" for the steps that you need to take.

"Could not find or load main class"

This error message is output by the `java` command if it has been unable to find / load the entry-point class that you have specified. In general terms, there are three broad reasons that this can happen:

- You have specified an entry point class that does not exist.
- The class exists, but you have specified it incorrectly.
- The class exists and you have specified it correctly, but Java cannot it find it because the classpath is incorrect.

Here is a procedure to diagnose and solve the problem:

1. Find out the full name of the entry-point class.
    - If you have source code for a class, then the full name consists of the package name and the simple class name. The instance the "Main" class is declared in the package "com.example.myapp" then its full name is "com.example.myapp.Main".
    - If you have a compiled class file, you can find the class name by running `javap` on it.
    - If the class file is in a directory, you can infer the full class name from the directory names.
    - If the class file is in a JAR or ZIP file, you can infer the full class name from the file path in the JAR or ZIP file.
2. Look at the error message from the `java` command. The message should end with the full class name that `java` is trying to use.
    - Check that it exactly matches the full classname for the entry-point class.
    - It should not end with ".java" or ".class".
    - It should not contain slashes or any other character that is not legal in a Java identifier1.
    - The casing of the name should exactly match the full class name.
3. If you are using the correct classname, make sure that the class is actually on the classpath:
    - Work out the pathname that the classname maps to; see Mapping classnames to pathnames
    - Work out what the classpath is; see this example: Different ways to specify the classpath
    - Look at each of the JAR and ZIP files on the classpath to see if they contain a class with the required pathname.
    - Look at each directory to see if the pathname resolves to a file within the directory.

If checking the classpath by hand did not find the issue, you could add the `-Xdiag` and `-XshowSettings` options. The former lists all classes that are loaded, and the latter prints out settings that include the effective classpath for the JVM.

Finally, there are some *obscure* causes for this problem:

- An executable JAR file with a `Main-Class` attribute that specifies a class that does not exist.
- An executable JAR file with an incorrect `Class-Path` attribute.
- If you mess up2 the options before the classname, the `java` command may attempt to interpret one of them

as the classname.
- If someone has ignored Java style rules and used package or class identifiers that differ only in letter case, and you are running on a platform that treats letter case in filenames as non-significant.
- Problems with homoglyphs in class names in the code or on the command line.

"Main method not found in class <name>"

This problem happens when the `java` command is able to find and load the class that you nominated, but is then unable to find an entry-point method.

There are three possible explanations:

- If you are trying to run an executable JAR file, then the JAR's manifest has an incorrect "Main-Class" attribute that specifies a class that is not a valid entry point class.
- You have told the `java` command a class that is not an entry point class.
- The entry point class is incorrect; see Entry point classes for more information.

Other Resources

- What does "Could not find or load main class" mean?
- http://docs.oracle.com/javase/tutorial/getStarted/problems/index.html

1 - From Java 8 and later, the `java` command will helpfully map a filename separator ("/" or "") to a period (".") However, this behavior is not documented in the manual pages.

2 - A really obscure case is if you copy-and-paste a command from a formatted document where the text editor has used a "long hyphen" instead of a regular hyphen.

Section 8.3: Running a Java application with library dependencies

This is a continuation of the "main class" and "executable JAR" examples.

Typical Java applications consist of an application-specific code, and various reusable library code that you have implemented or that has been implemented by third parties. The latter are commonly referred to as library dependencies, and are typically packaged as JAR files.

Java is a dynamically bound language. When you run a Java application with library dependencies, the JVM needs to know where the dependencies are so that it can load classes as required. Broadly speaking, there are two ways to deal with this:

- The application and its dependencies can be repackaged into a single JAR file that contains all of the required classes and resources.
- The JVM can be told where to find the dependent JAR files via the runtime classpath.

For an executable JAR file, the runtime classpath is specified by the "Class-Path" manifest attribute. *(Editorial Note: This should be described in a separate Topic on the `jar` command.)* Otherwise, the runtime classpath needs to be supplied using the `-cp` option or using the `CLASSPATH` environment variable.

For example, suppose that we have a Java application in the "myApp.jar" file whose entry point class is `com.example.MyApp`. Suppose also that the application depends on library JAR files "lib/library1.jar" and "lib/library2.jar". We could launch the application using the `java` command as follows in a command line:

```
$ # Alternative 1 (preferred)
```

```
$ java -cp myApp.jar:lib/library1.jar:lib/library2.jar com.example.MyApp

$ # Alternative 2
$ export CLASSPATH=myApp.jar:lib/library1.jar:lib/library2.jar
$ java com.example.MyApp
```

(On Windows, you would use `;` instead of `:` as the classpath separator, and you would set the (local) `CLASSPATH` variable using `set` rather than `export`.)

While a Java developer would be comfortable with that, it is not "user friendly". So it is common practice to write a simple shell script (or Windows batch file) to hide the details that the user doesn't need to know about. For example, if you put the following shell script into a file called "myApp", made it executable, and put it into a directory on the command search path:

```
#!/bin/bash
# The 'myApp' wrapper script

export DIR=/usr/libexec/myApp
export CLASSPATH=$DIR/myApp.jar:$DIR/lib/library1.jar:$DIR/lib/library2.jar
java com.example.MyApp
```

then you could run it as follows:

```
$ myApp arg1 arg2 ...
```

Any arguments on the command line will be passed to the Java application via the `"$@"` expansion. (You can do something similar with a Windows batch file, though the syntax is different.)

Section 8.4: Java Options

The `java` command supports a wide range of options:

- All options start with a single hyphen or minus-sign (-): the GNU/Linux convention of using `--` for "long" options is not supported.
- Options must appear before the **`<classname>`** or the `-jar <jarfile>` argument to be recognized. Any arguments after them will be treated as arguments to be passed to Java app that is being run.
- Options that do not start with `-X` or `-XX` are standard options. You can rely on all Java implementations1 to support any standard option.
- Options that start with `-X` are non-standard options, and may be withdrawn from one Java version to the next.
- Options that start with `-XX` are advanced options, and may also be withdrawn.

Setting system properties with `-D`

The `-D<property>=<value>` option is used to set a property in the system `Properties` object. This parameter can be repeated to set different properties.

Memory, Stack and Garbage Collector options

The main options for controlling the heap and stack sizes are documented in Setting the Heap, PermGen and Stack sizes. *(Editorial note: Garbage Collector options should be described in the same topic.)*

Enabling and disabling assertions

The `-ea` and `-da` options respectively enable and disable Java **`assert`** checking:

- All assertion checking is disabled by default.
- The `-ea` option enables checking of all assertions
- The `-ea:<packagename>...` enables checking of assertions in a package *and all subpackages*.
- The `-ea:<classname>...` enables checking of assertions in a class.
- The `-da` option disables checking of all assertions
- The `-da:<packagename>...` disables checking of assertions in a package *and all subpackages*.
- The `-da:<classname>...` disables checking of assertions in a class.
- The `-esa` option enables checking for all system classes.
- The `-dsa` option disables checking for all system classes.

The options can be combined. For example.

```
$ # Enable all assertion checking in non-system classes
$ java -ea -dsa MyApp

$ # Enable assertions for all classes in a package except for one.
$ java -ea:com.wombat.fruitbat... -da:com.wombat.fruitbat.Brickbat MyApp
```

Note that enabling to assertion checking is liable to alter the behavior of a Java programming.

- It is liable make the application slower in general.
- It can cause specific methods to take longer to run, which could change timing of threads in a multi-threaded application.
- It can introduce serendipitous *happens-before* relations which can cause memory anomalies to disappear.
- An incorrectly implemented **`assert`** statement could have unwanted side-effects.

Selecting the VM type

The `-client` and `-server` options allow you to select between two different forms of the HotSpot VM:

- The "client" form is tuned for user applications and offers faster startup.
- The "server" form is tuned for long running applications. It takes longer capturing statistic during JVM "warm up" which allows the JIT compiler to do a better of job of optimizing the native code.

By default, the JVM will run in 64bit mode if possible, depending on the capabilities of the platform. The `-d32` and `-d64` options allow you to select the mode explicitly.

1 - Check the official manual for the `java` command. Sometimes a *standard* option is described as "subject to change".

Section 8.5: Spaces and other special characters in arguments

First of all, the problem of handling spaces in arguments is NOT actually a Java problem. Rather it is a problem that needs to be handled by the command shell that you are using when you run a Java program.

As an example, let us suppose that we have the following simple program that prints the size of a file:

```
import java.io.File;

public class PrintFileSizes {
```

```
    public static void main(String[] args) {
        for (String name: args) {
            File file = new File(name);
            System.out.println("Size of '" + file + "' is " + file.size());
        }
    }
}
```

Now suppose that we want print the size of a file whose pathname has spaces in it; e.g. `/home/steve/Test File.txt`. If we run the command like this:

```
$ java PrintFileSizes /home/steve/Test File.txt
```

the shell won't know that `/home/steve/Test File.txt` is actually one pathname. Instead, it will pass 2 distinct arguments to the Java application, which will attempt to find their respective file sizes, and fail because files with those paths (probably) do not exist.

Solutions using a POSIX shell

POSIX shells include `sh` as well derivatives such as `bash` and `ksh`. If you are using one of these shells, then you can solve the problem by *quoting* the argument.

```
$ java PrintFileSizes "/home/steve/Test File.txt"
```

The double-quotes around the pathname tell the shell that it should be passed as a single argument. The quotes will be removed when this happens. There are a couple of other ways to do this:

```
$ java PrintFileSizes '/home/steve/Test File.txt'
```

Single (straight) quotes are treated like double-quotes except that they also suppress various expansions within the argument.

```
$ java PrintFileSizes /home/steve/Test\ File.txt
```

A backslash escapes the following space, and causes it not to be interpreted as an argument separator.

For more comprehensive documentation, including descriptions of how to deal with other special characters in arguments, please refer to the quoting topic in the Bash documentation.

Solution for Windows

The fundamental problem for Windows is that at the OS level, the arguments are passed to a child process as a single string (source). This means that the ultimate responsibility of parsing (or re-parsing) the command line falls on either program or its runtime libraries. There is lots of inconsistency.

In the Java case, to cut a long story short:

- You can put double-quotes around an argument in a `java` command, and that will allow you to pass arguments with spaces in them.
- Apparently, the `java` command itself is parsing the command string, and it gets it more or less right
- However, when you try to combine this with the use of `SET` and variable substitution in a batch file, it gets really complicated as to whether double-quotes get removed.

- The `cmd.exe` shell apparently has other escaping mechanisms; e.g. doubling double-quotes, and using `^` escapes.

For more detail, please refer to the Batch-File documentation.

Section 8.6: Running an executable JAR file

Executable JAR files are the simplest way to assemble Java code into a single file that can be executed. *(Editorial Note: Creation of JAR files should be covered by a separate Topic.) *

Assuming that you have an executable JAR file with pathname **`<jar-path>`**, you should be able to run it as follows:

```
java -jar <jar-path>
```

If the command requires command-line arguments, add them after the **`<jar-path>`**. For example:

```
java -jar <jar-path> arg1 arg2 arg3
```

If you need to provide additional JVM options on the `java` command line, they need to go *before* the `-jar` option. Note that a `-cp` / `-classpath` option will be ignored if you use `-jar`. The application's classpath is determined by the JAR file manifest.

Section 8.7: Running a Java applications via a "main" class

When an application has not been packaged as an executable JAR, you need to provide the name of an entry-point class on the `java` command line.

Running the HelloWorld class

The "HelloWorld" example is described in Creating a new Java program . It consists of a single class called `HelloWorld` which satisfies the requirements for an entry-point.

Assuming that the (compiled) "HelloWorld.class" file is in the current directory, it can be launched as follows:

```
java HelloWorld
```

Some important things to note are:

- We must provide the name of the class: not the pathname for the ".class" file or the ".java" file.
- If the class is declared in a package (as most Java classes are), then the class name we supply to the `java` command must be the full classname. For instance if `SomeClass` is declared in the `com.example` package, then the full classname will be `com.example.SomeClass`.

Specifying a classpath

Unless we are using in the `java -jar` command syntax, the `java` command looks for the class to be loaded by searching the classpath; see The Classpath. The above command is relying on the default classpath being (or including) the current directory. We can be more explicit about this by specifying the classpath to be used using the `-cp` option.

```
java -cp . HelloWorld
```

This says to make the current directory (which is what "." refers to) the sole entry on the classpath.

The `-cp` is an option that is processed by the `java` command. All options that are intended for the `java` command should be before the classname. Anything after the class will be treated as an command line argument for the Java application, and will be passed to application in the `String[]` that is passed to the `main` method.

(If no `-cp` option is provided, the `java` will use the classpath that is given by the `CLASSPATH` environment variable. If that variable is unset or empty, `java` uses "." as the default classpath.)

Chapter 9: Literals

A Java literal is a syntactic element (i.e. something you find in the *source code* of a Java program) that represents a value. Examples are `1`, `0.333F`, `false`, `&#39;X&#39;` and `"Hello world\n"`.

Section 9.1: Using underscore to improve readability

Since Java 7 it has been possible to use one or more underscores (_) for separating groups of digits in a primitive number literal to improve their readability.

For instance, these two declarations are equivalent:

Version ≥ Java SE 7

```
int i1 = 123456;
int i2 = 123_456;
System.out.println(i1 == i2); // true
```

This can be applied to all primitive number literals as shown below:

Version ≥ Java SE 7

```
byte color = 1_2_3;
short yearsAnnoDomini= 2_016;
int socialSecurtyNumber = 999_99_9999;
long creditCardNumber = 1234_5678_9012_3456L;
float piFourDecimals =  3.14_15F;
double piTenDecimals =  3.14_15_92_65_35;
```

This also works using prefixes for binary, octal and hexadecimal bases:

Version ≥ Java SE 7

```
short binary= 0b0_1_0_1;
int octal = 07_7_7_7_7_7_7_0;
long hexBytes = 0xFF_EC_DE_5E;
```

There are a few rules about underscores which **forbid** their placement in the following places:

- At the beginning or end of a number (e.g. `_123` or `123_` are *not* valid)
- Adjacent to a decimal point in a floating point literal (e.g. `1._23` or `1_.23` are *not* valid)
- Prior to an F or L suffix (e.g. `1.23_F` or `9999999_L` are *not* valid)
- In positions where a string of digits is expected (e.g. `0_xFFFF` is *not* valid)

Section 9.2: Hexadecimal, Octal and Binary literals

A `hexadecimal` number is a value in base-16. There are 16 digits, `0-9` and the letters `A-F` (case does not matter). `A-F` represent `10-15`.

An `octal` number is a value in base-8, and uses the digits `0-7`.

A `binary` number is a value in base-2, and uses the digits `0` and `1`.

All of these numbers result in the same value, `110`:

```
int dec = 110;            // no prefix   --> decimal literal
int bin = 0b1101110;      // '0b' prefix --> binary literal
int oct = 0156;           // '0' prefix  --> octal literal
```

```
int hex = 0x6E;           // '0x' prefix --> hexadecimal literal
```

Note that binary literal syntax was introduced in Java 7.

The octal literal can easily be a trap for semantic errors. If you define a leading `'0'` to your decimal literals you will get the wrong value:

```
int a = 0100;        // Instead of 100, a == 64
```

Section 9.3: Boolean literals

Boolean literals are the simplest of the literals in the Java programming language. The two possible **`boolean`** values are represented by the literals **`true`** and **`false`**. These are case-sensitive. For example:

```
boolean flag = true;    // using the 'true' literal
flag = false;           // using the 'false' literal
```

Section 9.4: String literals

String literals provide the most convenient way to represent string values in Java source code. A String literal consists of:

- An opening double-quote (`"`) character.
- Zero or more other characters that are neither a double-quote or a line-break character. (A backslash (`\`) character alters the meaning of subsequent characters; see Escape sequences in literals.)
- A closing double-quote character.

For example:

```
"Hello world"   // A literal denoting an 11 character String
""              // A literal denoting an empty (zero length) String
"\""            // A literal denoting a String consisting of one
                //     double quote character
"1\t2\t3\n"     // Another literal with escape sequences
```

Note that a single string literal may not span multiple source code lines. It is a compilation error for a line-break (or the end of the source file) to occur before a literal's closing double-quote. For example:

```
"Jello world    // Compilation error (at the end of the line!)
```

Long strings

If you need a string that is too long to fit on a line, the conventional way to express it is to split it into multiple literals and use the concatenation operator (+) to join the pieces. For example

```
String typingPractice = "The quick brown fox " +
                        "jumped over " +
                        "the lazy dog"
```

An expression like the above consisting of string literals and + satisfies the requirements to be a Constant Expression. That means that the expression will be evaluated by the compiler and represented at runtime by a single `String` object.

Interning of string literals

When class file containing string literals is loaded by the JVM, the corresponding `String` objects are *interned* by the runtime system. This means that a string literal used in multiple classes occupies no more space than if it was used in one class.

For more information on interning and the string pool, refer to the String pool and heap storage example in the Strings topic.

Section 9.5: The Null literal

The Null literal (written as **`null`**) represents the one and only value of the null type. Here are some examples

```
MyClass object = null;
MyClass[] objects = new MyClass[]{new MyClass(), null, new MyClass()};

myMethod(null);

if (objects != null) {
     // Do something
}
```

The null type is rather unusual. It has no name, so you cannot express it in Java source code. (And it has no runtime representation either.)

The sole purpose of the null type is to be the type of **`null`**. It is assignment compatible with all reference types, and can be type cast to any reference type. (In the latter case, the cast does not entail a runtime type check.)

Finally, **`null`** has the property that **`null instanceof`** `<SomeReferenceType>` will evaluate to **`false`**, no matter what the type is.

Section 9.6: Escape sequences in literals

String and character literals provide an escape mechanism that allows express character codes that would otherwise not be allowed in the literal. An escape sequence consists of a backslash character (`\`) followed by one ore more other characters. The same sequences are valid in both character an string literals.

The complete set of escape sequences is as follows:

Escape sequence	Meaning
`\\`	Denotes an backslash (`\`) character
`\'`	Denotes a single-quote (`'`) character
`\"`	Denotes a double-quote (`"`) character
`\n`	Denotes a line feed (`LF`) character
`\r`	Denotes a carriage return (`CR`) character
`\t`	Denotes a horizontal tab (`HT`) character
`\f`	Denotes a form feed (`FF`) character
`\b`	Denotes a backspace (`BS`) character
`\<octal>`	Denotes a character code in the range 0 to 255.

The **`<octal>`** in the above consists of one, two or three octal digits ('0' through '7') which represent a number between 0 and 255 (decimal).

Note that a backslash followed by any other character is an invalid escape sequence. Invalid escape sequences are treated as compilation errors by the JLS.

Reference:

- JLS 3.10.6. Escape Sequences for Character and String Literals

Unicode escapes

In addition to the string and character escape sequences described above, Java has a more general Unicode escaping mechanism, as defined in JLS 3.3. Unicode Escapes. A Unicode escape has the following syntax:

```
'\' 'u' <hex-digit> <hex-digit> <hex-digit> <hex-digit>
```

where **<hex-digit>** is one of '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f', 'A', 'B', 'C', 'D', 'E', 'F'.

A Unicode escape is mapped by the Java compiler to a character (strictly speaking a 16-bit Unicode *code unit*), and can be used anywhere in the source code where the mapped character is valid. It is commonly used in character and string literals when you need to represent a non-ASCII character in a literal.

Escaping in regexes

TBD

Section 9.7: Character literals

Character literals provide the most convenient way to express **char** values in Java source code. A character literal consists of:

- An opening single-quote (') character.
- A representation of a character. This representation cannot be a single-quote or a line-break character, but it can be an escape sequence introduced by a backslash (\) character; see Escape sequences in literals.
- A closing single-quote (') character.

For example:

```
char a = 'a';
char doubleQuote = '"';
char singleQuote = '\'';
```

A line-break in a character literal is a compilation error:

```
char newline = '
// Compilation error in previous line
char newLine = '\n';  // Correct
```

Section 9.8: Decimal Integer literals

Integer literals provide values that can be used where you need a **byte**, **short**, **int**, **long** or **char** instance. (This example focuses on the simple decimal forms. Other examples explain how to literals in octal, hexadecimal and binary, and the use of underscores to improve readability.)

Ordinary integer literals

The simplest and most common form of integer literal is a decimal integer literal. For example:

```
0       // The decimal number zero       (type 'int')
```

```
1       // The decimal number one        (type 'int')
42      // The decimal number forty two  (type 'int')
```

You need to be careful with leading zeros. A leading zero causes an integer literal to be interpreted as *octal* not decimal.

```
077    // This literal actually means 7 x 8 + 7 ... or 63 decimal!
```

Integer literals are unsigned. If you see something like `-10` or `+10`, these are actually *expressions* using the unary `-` and unary `+` operators.

The range of integer literals of this form have an intrinsic type of **`int`**, and must fall in the range zero to 231 or 2,147,483,648.

Note that 231 is 1 greater than `Integer.MAX_VALUE`. Literals from 0 through to `2147483647` can be used anywhere, but it is a compilation error to use `2147483648` without a preceding unary `-` operator. (In other words, it is reserved for expressing the value of `Integer.MIN_VALUE`.)

```
int max = 2147483647;     // OK
int min = -2147483648;    // OK
int tooBig = 2147483648;  // ERROR
```

Long integer literals

Literals of type **`long`** are expressed by adding an `L` suffix. For example:

```
0L          // The decimal number zero       (type 'long')
1L          // The decimal number one        (type 'long')
2147483648L // The value of Integer.MAX_VALUE + 1

long big = 2147483648;    // ERROR
long big2 = 2147483648L;  // OK
```

Note that the distinction between **`int`** and **`long`** literals is significant in other places. For example

```
int i = 2147483647;
long l = i + 1;           // Produces a negative value because the operation is
                          // performed using 32 bit arithmetic, and the
                          // addition overflows
long l2 = i + 1L;         // Produces the (intuitively) correct value.
```

Reference: JLS 3.10.1 - Integer Literals

Section 9.9: Floating-point literals

Floating point literals provide values that can be used where you need a **`float`** or **`double`** instance. There are three kinds of floating point literal.

- Simple decimal forms
- Scaled decimal forms
- Hexadecimal forms

(The JLS syntax rules combine the two decimal forms into a single form. We treat them separately for ease of explanation.)

There are distinct literal types for **`float`** and **`double`** literals, expressed using suffixes. The various forms use letters to express different things. These letters are case insensitive.

Simple decimal forms

The simplest form of floating point literal consists of one or more decimal digits and a decimal point (`.`) and an optional suffix (`f`, `F`, `d` or `D`). The optional suffix allows you to specify that the literal is a **`float`** (`f` or `F`) or **`double`** (`d` or `D`) value. The default (when no suffix is specified) is **`double`**.

For example

```
0.0     // this denotes zero
.0      // this also denotes zero
0.      // this also denotes zero
3.14159 // this denotes Pi, accurate to (approximately!) 5 decimal places.
1.0F    // a `float` literal
1.0D    // a `double` literal.  (`double` is the default if no suffix is given)
```

In fact, decimal digits followed by a suffix is also a floating point literal.

```
1F      // means the same thing as 1.0F
```

The meaning of a decimal literal is the IEEE floating point number that is *closest* to the infinite precision mathematical Real number denoted by the decimal floating point form. This conceptual value is converted to IEEE binary floating point representation using *round to nearest*. (The precise semantics of decimal conversion are specified in the javadocs for `Double.valueOf(String)` and `Float.valueOf(String)`, bearing in mind that there are differences in the number syntaxes.)

Scaled decimal forms

Scaled decimal forms consist of simple decimal with an exponent part introduced by an `E` or `e`, and followed by a signed integer. The exponent part is a short hand for multiplying the decimal form by a power of ten, as shown in the examples below. There is also an optional suffix to distinguish **`float`** and **`double`** literals. Here are some examples:

```
1.0E1    // this means 1.0 x 10^1 ... or 10.0 (double)
1E-1D    // this means 1.0 x 10^(-1) ... or 0.1 (double)
1.0e10f  // this means 1.0 x 10^(10) ... or 10000000000.0 (float)
```

The size of a literal is limited by the representation (**`float`** or **`double`**). It is a compilation error if the scale factor results in a value that is too large or too small.

Hexadecimal forms

Starting with Java 6, it is possible to express floating point literals in hexadecimal. The hexadecimal form have an analogous syntax to the simple and scaled decimal forms with the following differences:

1. Every hexadecimal floating point literal starts with a zero (`0`) and then an `x` or `X`.
2. The digits of the number (but *not* the exponent part!) also include the hexadecimal digits `a` through `f` and their uppercase equivalents.
3. The exponent is *mandatory*, and is introduced by the letter `p` (or `P`) instead of an `e` or `E`. The exponent represents a scaling factor that is a power of 2 instead of a power of 10.

Here are some examples:

```
0x0.0p0f    // this is zero expressed in hexadecimal form (`float`)
0xff.0p19   // this is 255.0 x 2^19 (`double`)
```

Advice: since hexadecimal floating-point forms are unfamiliar to most Java programmers, it is advisable to use them sparingly.

Underscores

Starting with Java 7, underscores are permitted within the digit strings in all three forms of floating point literal. This applies to the "exponent" parts as well. See Using underscores to improve readability.

Special cases

It is a compilation error if a floating point literal denotes a number that is too large or too small to represent in the selected representation; i.e. if the number would overflow to +INF or -INF, or underflow to 0.0. However, it is legal for a literal to represent a non-zero denormalized number.

The floating point literal syntax does not provide literal representations for IEEE 754 special values such as the INF and NaN values. If you need to express them in source code, the recommended way is to use the constants defined by the `java.lang.Float` and `java.lang.Double`; e.g. `Float.NaN`, `Float.NEGATIVE_INFINITY` and `Float.POSITIVE_INFINITY`.

Chapter 10: Primitive Data Types

The 8 primitive data types `byte`, `short`, `int`, `long`, `char`, `boolean`, `float`, and `double` are the types that store most raw numerical data in Java programs.

Section 10.1: The char primitive

A `char` can store a single 16-bit Unicode character. A character literal is enclosed in single quotes

```
char myChar = 'u';
char myChar2 = '5';
char myChar3 = 65; // myChar3 == 'A'
```

It has a minimum value of `\u0000` (0 in the decimal representation, also called the *null character*) and a maximum value of `\uffff` (65,535).

The default value of a `char` is `\u0000`.

```
char defaultChar;    // defaultChar == \u0000
```

In order to define a char of `'` value an escape sequence (character preceded by a backslash) has to be used:

```
char singleQuote = '\'';
```

There are also other escape sequences:

```
char tab = '\t';
char backspace = '\b';
char newline = '\n';
char carriageReturn = '\r';
char formfeed = '\f';
char singleQuote = '\'';
char doubleQuote = '\"'; // escaping redundant here; '"' would be the same; however still allowed
char backslash = '\\';
char unicodeChar = '\uXXXX' // XXXX represents the Unicode-value of the character you want to
display
```

You can declare a `char` of any Unicode character.

```
char heart = '\u2764';
System.out.println(Character.toString(heart)); // Prints a line containing "❤".
```

It is also possible to add to a `char`. e.g. to iterate through every lower-case letter, you could do to the following:

```
for (int i = 0; i <= 26; i++) {
    char letter = (char) ('a' + i);
    System.out.println(letter);
}
```

Section 10.2: Primitive Types Cheatsheet

Table showing size and values range of all primitive types:

data type	numeric representation	range of values	default value

boolean	n/a	false and true	false
byte	8-bit signed	-27 to 27 - 1	0
		-128 to +127	
short	16-bit signed	-215 to 215 - 1	0
		-32,768 to +32,767	
int	32-bit signed	-231 to 231 - 1	0
		-2,147,483,648 to +2,147,483,647	
long	64-bit signed	-263 to 263 - 1	0L
		-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	
float	32-bit floating point	1.401298464e-45 to 3.402823466e+38 (positive or negative)	0.0F
double	64-bit floating point	4.94065645841246544e-324d to 1.79769313486231570e+308d (positive or negative)	0.0D
char	16-bit unsigned	0 to 216 - 1	0
		0 to 65,535	

Notes:

1. The Java Language Specification mandates that signed integral types (**`byte`** through **`long`**) use binary twos-complement representation, and the floating point types use standard IEE 754 binary floating point representations.
2. Java 8 and later provide methods to perform unsigned arithmetic operations on **`int`** and **`long`**. While these methods allow a program to *treat* values of the respective types as unsigned, the types remain signed types.
3. The smallest floating point shown above are *subnormal*; i.e. they have less precision than a *normal* value. The smallest normal numbers are 1.175494351e−38 and 2.2250738585072014e−308
4. A **`char`** conventionally represents a Unicode / UTF-16 *code unit*.
5. Although a **`boolean`** contains just one bit of information, its size in memory varies depending on the Java Virtual Machine implementation (see boolean type).

Section 10.3: The float primitive

A **`float`** is a single-precision 32-bit IEEE 754 floating point number. By default, decimals are interpreted as doubles. To create a **`float`**, simply append an `f` to the decimal literal.

```
double doubleExample = 0.5;       // without 'f' after digits = double
float floatExample = 0.5f;        // with 'f' after digits    = float

float myFloat = 92.7f;            // this is a float...
float positiveFloat = 89.3f;      // it can be positive,
float negativeFloat = -89.3f;     // or negative
float integerFloat = 43.0f;       // it can be a whole number (not an int)
float underZeroFloat = 0.0549f;   // it can be a fractional value less than 0
```

Floats handle the five common arithmetical operations: addition, subtraction, multiplication, division, and modulus.

Note: The following may vary slightly as a result of floating point errors. Some results have been rounded for clarity and readability purposes (i.e. the printed result of the addition example was actually 34.600002).

```
// addition
float result = 37.2f + -2.6f;  // result: 34.6

// subtraction
float result = 45.1f - 10.3f;    // result: 34.8
```

```
// multiplication
float result = 26.3f * 1.7f;   // result: 44.71

// division
float result = 37.1f / 4.8f;   // result: 7.729166

// modulus
float result = 37.1f % 4.8f;   // result: 3.4999971
```

Because of the way floating point numbers are stored (i.e. in binary form), many numbers don't have an exact representation.

```
float notExact = 3.1415926f;
System.out.println(notExact); // 3.1415925
```

While using **`float`** is fine for most applications, neither **`float`** nor **`double`** should be used to store exact representations of decimal numbers (like monetary amounts), or numbers where higher precision is required. Instead, the `BigDecimal` class should be used.

The default value of a **`float`** is *0.0f*.

```
float defaultFloat;    // defaultFloat == 0.0f
```

A **`float`** is precise to roughly an error of 1 in 10 million.

Note: `Float.POSITIVE_INFINITY`, `Float.NEGATIVE_INFINITY`, `Float.NaN` are **`float`** values. `NaN` stands for results of operations that cannot be determined, such as dividing 2 infinite values. Furthermore `0f` and `-0f` are different, but `==` yields true:

```
float f1 = 0f;
float f2 = -0f;
System.out.println(f1 == f2); // true
System.out.println(1f / f1); // Infinity
System.out.println(1f / f2); // -Infinity
System.out.println(Float.POSITIVE_INFINITY / Float.POSITIVE_INFINITY); // NaN
```

Section 10.4: The int primitive

A primitive data type such as **`int`** holds values directly into the variable that is using it, meanwhile a variable that was declared using `Integer` holds a reference to the value.

According to java API: "The Integer class wraps a value of the primitive type int in an object. An object of type Integer contains a single field whose type is int."

By default, **`int`** is a 32-bit signed integer. It can store a minimum value of -231, and a maximum value of 231 - 1.

```
int example = -42;
int myInt = 284;
int anotherInt = 73;

int addedInts = myInt + anotherInt; // 284 + 73 = 357
int subtractedInts = myInt - anotherInt; // 284 - 73 = 211
```

If you need to store a number outside of this range, **`long`** should be used instead. Exceeding the value range of **`int`** leads to an integer overflow, causing the value exceeding the range to be added to the opposite site of the range (positive becomes negative and vise versa). The value is `((value - MIN_VALUE) % RANGE) + MIN_VALUE`, or `((value`

```
+ 2147483648) % 4294967296) - 2147483648
```

```
int demo = 2147483647; //maximum positive integer
System.out.println(demo); //prints 2147483647
demo = demo + 1; //leads to an integer overflow
System.out.println(demo); // prints -2147483648
```

The maximum and minimum values of `int` can be found at:

```
int high = Integer.MAX_VALUE;    // high == 2147483647
int low = Integer.MIN_VALUE;     // low == -2147483648
```

The default value of an `int` is *0*

```
int defaultInt;    // defaultInt == 0
```

Section 10.5: Converting Primitives

In Java, we can convert between integer values and floating-point values. Also, since every character corresponds to a number in the Unicode encoding, `char` types can be converted to and from the integer and floating-point types. `boolean` is the only primitive datatype that cannot be converted to or from any other primitive datatype.

There are two types of conversions: *widening conversion* and *narrowing conversion*.

A *widening conversion* is when a value of one datatype is converted to a value of another datatype that occupies more bits than the former. There is no issue of data loss in this case.

Correspondingly, A *narrowing conversion* is when a value of one datatype is converted to a value of another datatype that occupies fewer bits than the former. Data loss can occur in this case.

Java performs *widening conversions* automatically. But if you want to perform a *narrowing conversion* (if you are sure that no data loss will occur), then you can force Java to perform the conversion using a language construct known as a `cast`.

Widening Conversion:

```
int a = 1;
double d = a;    // valid conversion to double, no cast needed (widening)
```

Narrowing Conversion:

```
double d = 18.96
int b = d;       // invalid conversion to int, will throw a compile-time error
int b = (int) d; // valid conversion to int, but result is truncated (gets rounded down)
                 // This is type-casting
                 // Now, b = 18
```

Section 10.6: Memory consumption of primitives vs. boxed primitives

Primitive	Boxed Type	Memory Size of primitive / boxed
boolean	Boolean	1 byte / 16 bytes
byte	Byte	1 byte / 16 bytes
short	Short	2 bytes / 16 bytes

char	Char	2 bytes / 16 bytes
int	Integer	4 bytes / 16 bytes
long	Long	8 bytes / 16 bytes
float	Float	4 bytes / 16 bytes
double	Double	8 bytes / 16 bytes

Boxed objects always require 8 bytes for type and memory management, and because the size of objects is always a multiple of 8, boxed types *all require 16 bytes total*. In *addition*, each usage of a boxed object entails storing a reference which accounts for another 4 or 8 bytes, depending on the JVM and JVM options.

In data-intensive operations, memory consumption can have a major impact on performance. Memory consumption grows even more when using arrays: a `float[5]` array will require only 32 bytes; whereas a `Float[5]` storing 5 distinct non-null values will require 112 bytes total (on 64 bit without compressed pointers, this increases to 152 bytes).

Boxed value caches

The space overheads of the boxed types can be mitigated to a degree by the boxed value caches. Some of the boxed types implement a cache of instances. For example, by default, the `Integer` class will cache instances to represent numbers in the range `-128` to `+127`. This does not, however, reduce the additional cost arising from the additional memory indirection.

If you create an instance of a boxed type either by autoboxing or by calling the static `valueOf(primitive)` method, the runtime system will attempt to use a cached value. If your application uses a lot of values in the range that is cached, then this can substantially reduce the memory penalty of using boxed types. Certainly, if you are creating boxed value instances "by hand", it is better to use `valueOf` rather than **`new`**. (The **`new`** operation always creates a new instance.) If, however, the majority of your values are *not* in the cached range, it can be faster to call **`new`** and save the cache lookup.

Section 10.7: The double primitive

A **`double`** is a double-precision 64-bit IEEE 754 floating point number.

```
double example = -7162.37;
double myDouble = 974.21;
double anotherDouble = 658.7;

double addedDoubles = myDouble + anotherDouble; // 315.51
double subtractedDoubles = myDouble - anotherDouble; // 1632.91

double scientificNotationDouble = 1.2e-3;    // 0.0012
```

Because of the way floating point numbers are stored, many numbers don't have an exact representation.

```
double notExact = 1.32 - 0.42; // result should be 0.9
System.out.println(notExact); // 0.9000000000000001
```

While using **`double`** is fine for most applications, neither **`float`** nor **`double`** should be used to store precise numbers such as currency. Instead, the `BigDecimal` class should be used

The default value of a **`double`** is *0.0d*

```
public double defaultDouble;    // defaultDouble == 0.0
```

Note: `Double.POSITIVE_INFINITY`, `Double.NEGATIVE_INFINITY`, `Double.NaN` are **`double`** values. `NaN` stands for results of operations that cannot be determined, such as dividing 2 infinite values. Furthermore `0d` and `-0d` are different, but `==` yields true:

```
double d1 = 0d;
double d2 = -0d;
System.out.println(d1 == d2); // true
System.out.println(1d / d1); // Infinity
System.out.println(1d / d2); // -Infinity
System.out.println(Double.POSITIVE_INFINITY / Double.POSITIVE_INFINITY); // NaN
```

Section 10.8: The long primitive

By default, **`long`** is a 64-bit signed integer (in Java 8, it can be either signed or unsigned). Signed, it can store a minimum value of -263, and a maximum value of 263 - 1, and unsigned it can store a minimum value of 0 and a maximum value of 264 - 1

```
long example = -42;
long myLong = 284;
long anotherLong = 73;

//an "L" must be appended to the end of the number, because by default,
//numbers are assumed to be the int type. Appending an "L" makes it a long
//as 549755813888 (2 ^ 39) is larger than the maximum value of an int (2^31 - 1),
//"L" must be appended
long bigNumber = 549755813888L;

long addedLongs = myLong + anotherLong; // 284 + 73 = 357
long subtractedLongs = myLong - anotherLong; // 284 - 73 = 211
```

The maximum and minimum values of **`long`** can be found at:

```
long high = Long.MAX_VALUE;     // high == 9223372036854775807L
long low = Long.MIN_VALUE;      // low == -9223372036854775808L
```

The default value of a **`long`** is *0L*

```
long defaultLong;     // defaultLong == 0L
```

Note: letter "L" appended at the end of **`long`** literal is case insensitive, however it is good practice to use capital as it is easier to distinct from digit one:

```
2L == 2l;             // true
```

Warning: Java caches Integer objects instances from the range -128 to 127. The reasoning is explained here: https://blogs.oracle.com/darcy/entry/boxing_and_caches_integer_valueof

The following results can be found:

```
Long val1 = 127L;
Long val2 = 127L;

System.out.println(val1 == val2); // true

Long val3 = 128L;
Long val4 = 128L;
```

```
System.out.println(val3 == val4); // false
```

To properly compare 2 Object Long values, use the following code(From Java 1.7 onward):

```
Long val3 = 128L;
Long val4 = 128L;

System.out.println(Objects.equal(val3, val4)); // true
```

Comparing a primitive long to an Object long will not result in a false negative like comparing 2 objects with == does.

Section 10.9: The boolean primitive

A **`boolean`** can store one of two values, either **`true`** or **`false`**

```
boolean foo = true;
System.out.println("foo = " + foo);                // foo = true

boolean bar = false;
System.out.println("bar = " + bar);                // bar = false

boolean notFoo = !foo;
System.out.println("notFoo = " + notFoo);          // notFoo = false

boolean fooAndBar = foo && bar;
System.out.println("fooAndBar = " + fooAndBar);    // fooAndBar = false

boolean fooOrBar = foo || bar;
System.out.println("fooOrBar = " + fooOrBar);      // fooOrBar = true

boolean fooXorBar = foo ^ bar;
System.out.println("fooXorBar = " + fooXorBar);    // fooXorBar = true
```

The default value of a **`boolean`** is *false*

```
boolean defaultBoolean;    // defaultBoolean == false
```

Section 10.10: The byte primitive

A **`byte`** is a 8-bit signed integer. It can store a minimum value of -27 (-128), and a maximum value of 27 - 1 (127)

```
byte example = -36;
byte myByte = 96;
byte anotherByte = 7;

byte addedBytes = (byte) (myByte + anotherByte); // 103
byte subtractedBytes = (byte) (myBytes - anotherByte); // 89
```

The maximum and minimum values of **`byte`** can be found at:

```
byte high = Byte.MAX_VALUE;        // high == 127
byte low = Byte.MIN_VALUE;         // low == -128
```

The default value of a **`byte`** is *0*

```
byte defaultByte;    // defaultByte == 0
```

Section 10.11: Negative value representation

Java and most other languages store negative integral numbers in a representation called *2's complement* notation.

For a unique binary representation of a data type using `n` bits, values are encoded like this:

The least significant `n-1` bits store a positive integral number `x` in integral representation. Most significant value stores a bit vith value `s`. The value repesented by those bits is

x - s * 2n-1

i.e. if the most significant bit is 1, then a value that is just by 1 larger than the number you could represent with the other bits (`2n-2 + 2n-3 + ... + 21 + 20 = 2n-1 - 1`) is subtracted allowing a unique binary representation for each value from - 2n-1 (s = 1; x = 0) to 2n-1 - 1 (s = 0; x = 2n-1 - 1).

This also has the nice side effect, that you can add the binary representations as if they were positive binary numbers:

v1 = x1 - s1 * 2n-1 v2 = x2 - s2 * 2n-1

s1	s2	x1 + x2 overflow	addition result
0	0	No	x1 + x2 = v1 + v2
0	0	Yes	too large to be represented with data type (overflow)
0	1	No	x1 + x2 - 2n-1 = x1 + x2 - s2 * 2n-1 = v1 + v2
0	1	Yes	(x1 + x2) **mod** 2n-1 = x1 + x2 - 2n-1 = v1 + v2
1	0	*	see above (swap summands)
1	1	No	too small to be represented with data type (x1 + x2 - 2n < -2n-1 ; underflow)
1	1	Yes	(x1 + x2) **mod** 2n-1 - 2n-1 = (x1 + x2 - 2n-1) - 2n-1 = (x1 - s1 * 2n-1) + (x2 - s2 * 2n-1) = v1 + v2

Note that this fact makes finding binary representation of the additive inverse (i.e. the negative value) easy:

Observe that adding the bitwise complement to the number results in all bits being 1. Now add 1 to make value overflow and you get the neutral element 0 (all bits 0).

So the negative value of a number `i` can be calculated using (ignoring possible promotion to **`int`** here)

```
(~i) + 1
```

Example: taking the negative value of 0 (**`byte`**):

The result of negating `0`, is `11111111`. Adding 1 gives a value of `100000000` (9 bits). Because a **`byte`** can only store 8 bits, the leftmost value is truncated, and the result is `00000000`

Original	Process	Result
0 (00000000)	Negate	-0 (11111111)
11111111	Add 1 to binary	100000000
100000000	Truncate to 8 bits	00000000 (-0 equals 0)

Section 10.12: The short primitive

A `short` is a 16-bit signed integer. It has a minimum value of -215 (-32,768), and a maximum value of 215 -1 (32,767)

```
short example = -48;
short myShort = 987;
short anotherShort = 17;

short addedShorts = (short) (myShort + anotherShort); // 1,004
short subtractedShorts = (short) (myShort - anotherShort); // 970
```

The maximum and minimum values of `short` can be found at:

```
short high = Short.MAX_VALUE;        // high == 32767
short low = Short.MIN_VALUE;         // low == -32768
```

The default value of a `short` is *0*

```
short defaultShort;    // defaultShort == 0
```

Chapter 11: Strings

Strings (`java.lang.String`) are pieces of text stored in your program. Strings are **not** a primitive data type in Java, however, they are very common in Java programs.

In Java, Strings are immutable, meaning that they cannot be changed. (Click here for a more thorough explanation of immutability.)

Section 11.1: Comparing Strings

In order to compare Strings for equality, you should use the String object's `equals` or `equalsIgnoreCase` methods.

For example, the following snippet will determine if the two instances of `String` are equal on all characters:

```
String firstString = "Test123";
String secondString = "Test" + 123;

if (firstString.equals(secondString)) {
   // Both Strings have the same content.
}
```

Live demo

This example will compare them, independent of their case:

```
String firstString = "Test123";
String secondString = "TEST123";

if (firstString.equalsIgnoreCase(secondString)) {
    // Both Strings are equal, ignoring the case of the individual characters.
}
```

Live demo

Note that `equalsIgnoreCase` does not let you specify a `Locale`. For instance, if you compare the two words `"Taki"` and `"TAKI"` in English they are equal; however, in Turkish they are different (in Turkish, the lowercase `I` is `ı`). For cases like this, converting both strings to lowercase (or uppercase) with `Locale` and then comparing with `equals` is the solution.

```
String firstString = "Taki";
String secondString = "TAKI";

System.out.println(firstString.equalsIgnoreCase(secondString)); //prints true

Locale locale = Locale.forLanguageTag("tr-TR");

System.out.println(firstString.toLowerCase(locale).equals(
                   secondString.toLowerCase(locale))); //prints false
```

Live demo

Do not use the == operator to compare Strings

Unless you can guarantee that all strings have been interned (see below), you **should not** use the `==` or `!=`

operators to compare Strings. These operators actually test references, and since multiple `String` objects can represent the same String, this is liable to give the wrong answer.

Instead, use the `String.equals(Object)` method, which will compare the String objects based on their values. For a detailed explanation, please refer to Pitfall: using == to compare strings.

Comparing Strings in a switch statement

Version ≥ Java SE 7

As of Java 1.7, it is possible to compare a String variable to literals in a **`switch`** statement. Make sure that the String is not null, otherwise it will always throw a `NullPointerException`. Values are compared using `String.equals`, i.e. case sensitive.

```
String stringToSwitch = "A";

switch (stringToSwitch) {
    case "a":
        System.out.println("a");
        break;
    case "A":
        System.out.println("A"); //the code goes here
        break;
    case "B":
        System.out.println("B");
        break;
    default:
        break;
}
```

Live demo

Comparing Strings with constant values

When comparing a `String` to a constant value, you can put the constant value on the left side of `equals` to ensure that you won't get a `NullPointerException` if the other String is **`null`**.

```
"baz".equals(foo)
```

While `foo.equals("baz")` will throw a `NullPointerException` if `foo` is **`null`**, `"baz".equals(foo)` will evaluate to **`false`**.

Version ≥ Java SE 7

A more readable alternative is to use `Objects.equals()`, which does a null check on both parameters: `Objects.equals(foo, "baz")`.

(**Note:** It is debatable as to whether it is better to avoid `NullPointerExceptions` in general, or let them happen and then fix the root cause; see here and here. Certainly, calling the avoidance strategy "best practice" is not justifiable.)

String orderings

The `String` class implements `Comparable<String>` with the `String.compareTo` method (as described at the start of this example). This makes the natural ordering of `String` objects case-sensitive order. The `String` class provide a `Comparator<String>` constant called `CASE_INSENSITIVE_ORDER` suitable for case-insensitive sorting.

Comparing with interned Strings

The Java Language Specification (JLS 3.10.6) states the following:

> "Moreover, a string literal always refers to the same instance of class `String`. This is because string literals - or, more generally, strings that are the values of constant expressions - are *interned* so as to share unique instances, using the method `String.intern`."

This means it is safe to compare references to two string *literals* using ==. Moreover, the same is true for references to `String` objects that have been produced using the `String.intern()` method.

For example:

```
String strObj = new String("Hello!");
String str = "Hello!";

// The two string references point two strings that are equal
if (strObj.equals(str)) {
    System.out.println("The strings are equal");
}

// The two string references do not point to the same object
if (strObj != str) {
    System.out.println("The strings are not the same object");
}

// If we intern a string that is equal to a given literal, the result is
// a string that has the same reference as the literal.
String internedStr = strObj.intern();

if (internedStr == str) {
    System.out.println("The interned string and the literal are the same object");
}
```

Behind the scenes, the interning mechanism maintains a hash table that contains all interned strings that are still *reachable*. When you call `intern()` on a `String`, the method looks up the object in the hash table:

- If the string is found, then that value is returned as the interned string.
- Otherwise, a copy of the string is added to the hash table and that string is returned as the interned string.

It is possible to use interning to allow strings to be compared using ==. However, there are significant problems with doing this; see Pitfall - Interning strings so that you can use == is a bad idea for details. It is not recommended in most cases.

Section 11.2: Changing the case of characters within a String

The `String` type provides two methods for converting strings between upper case and lower case:

- `toUpperCase` to convert all characters to upper case
- `toLowerCase` to convert all characters to lower case

These methods both return the converted strings as new `String` instances: the original `String` objects are not modified because `String` is immutable in Java. See this for more on immutability : Immutability of Strings in Java

```
String string = "This is a Random String";
```

```
String upper = string.toUpperCase();
String lower = string.toLowerCase();

System.out.println(string);   // prints "This is a Random String"
System.out.println(lower);    // prints "this is a random string"
System.out.println(upper);    // prints "THIS IS A RANDOM STRING"
```

Non-alphabetic characters, such as digits and punctuation marks, are unaffected by these methods. Note that these methods may also incorrectly deal with certain Unicode characters under certain conditions.

Note: These methods are *locale-sensitive*, and may produce unexpected results if used on strings that are intended to be interpreted independent of the locale. Examples are programming language identifiers, protocol keys, and `HTML` tags.

For instance, `"TITLE".toLowerCase()` in a Turkish locale returns "`tıtle`", where `ı (\u0131)` is the LATIN SMALL LETTER DOTLESS I character. To obtain correct results for locale insensitive strings, pass `Locale.ROOT` as a parameter to the corresponding case converting method (e.g. `toLowerCase(Locale.ROOT)` or `toUpperCase(Locale.ROOT)`).

Although using `Locale.ENGLISH` is also correct for most cases, the **language invariant** way is `Locale.ROOT`.

A detailed list of Unicode characters that require special casing can be found on the Unicode Consortium website.

Changing case of a specific character within an ASCII string:

To change the case of a specific character of an ASCII string following algorithm can be used:

Steps:

1. Declare a string.
2. Input the string.
3. Convert the string into a character array.
4. Input the character that is to be searched.
5. Search for the character into the character array.
6. If found,check if the character is lowercase or uppercase.
    - If Uppercase, add 32 to the ASCII code of the character.
    - If Lowercase, subtract 32 from the ASCII code of the character.
7. Change the original character from the Character array.
8. Convert the character array back into the string.

Voila, the Case of the character is changed.

An example of the code for the algorithm is:

```
Scanner scanner = new Scanner(System.in);
System.out.println("Enter the String");
String s = scanner.next();
char[] a = s.toCharArray();
System.out.println("Enter the character you are looking for");
System.out.println(s);
String c = scanner.next();
char d = c.charAt(0);

for (int i = 0; i <= s.length(); i++) {
    if (a[i] == d) {
        if (d >= 'a' && d <= 'z') {
            d -= 32;
```

```
        } else if (d >= 'A' && d <= 'Z') {
            d += 32;
        }
        a[i] = d;
        break;
    }
}
s = String.valueOf(a);
System.out.println(s);
```

Section 11.3: Finding a String Within Another String

To check whether a particular String `a` is being contained in a String `b` or not, we can use the method `String.contains()` with the following syntax:

```
b.contains(a); // Return true if a is contained in b, false otherwise
```

The `String.contains()` method can be used to verify if a `CharSequence` can be found in the String. The method looks for the String `a` in the String `b` in a case-sensitive way.

```
String str1 = "Hello World";
String str2 = "Hello";
String str3 = "helLO";

System.out.println(str1.contains(str2)); //prints true
System.out.println(str1.contains(str3)); //prints false
```

Live Demo on Ideone

To find the exact position where a String starts within another String, use `String.indexOf()`:

```
String s = "this is a long sentence";
int i = s.indexOf('i');    // the first 'i' in String is at index 2
int j = s.indexOf("long"); // the index of the first occurrence of "long" in s is 10
int k = s.indexOf('z');    // k is -1 because 'z' was not found in String s
int h = s.indexOf("LoNg"); // h is -1 because "LoNg" was not found in String s
```

Live Demo on Ideone

The `String.indexOf()` method returns the first index of a **`char`** or `String` in another `String`. The method returns `-1` if it is not found.

Note: The `String.indexOf()` method is case sensitive.

Example of search ignoring the case:

```
String str1 = "Hello World";
String str2 = "wOr";
str1.indexOf(str2);                                 // -1
str1.toLowerCase().contains(str2.toLowerCase());    // true
str1.toLowerCase().indexOf(str2.toLowerCase());     // 6
```

Live Demo on Ideone

Section 11.4: String pool and heap storage

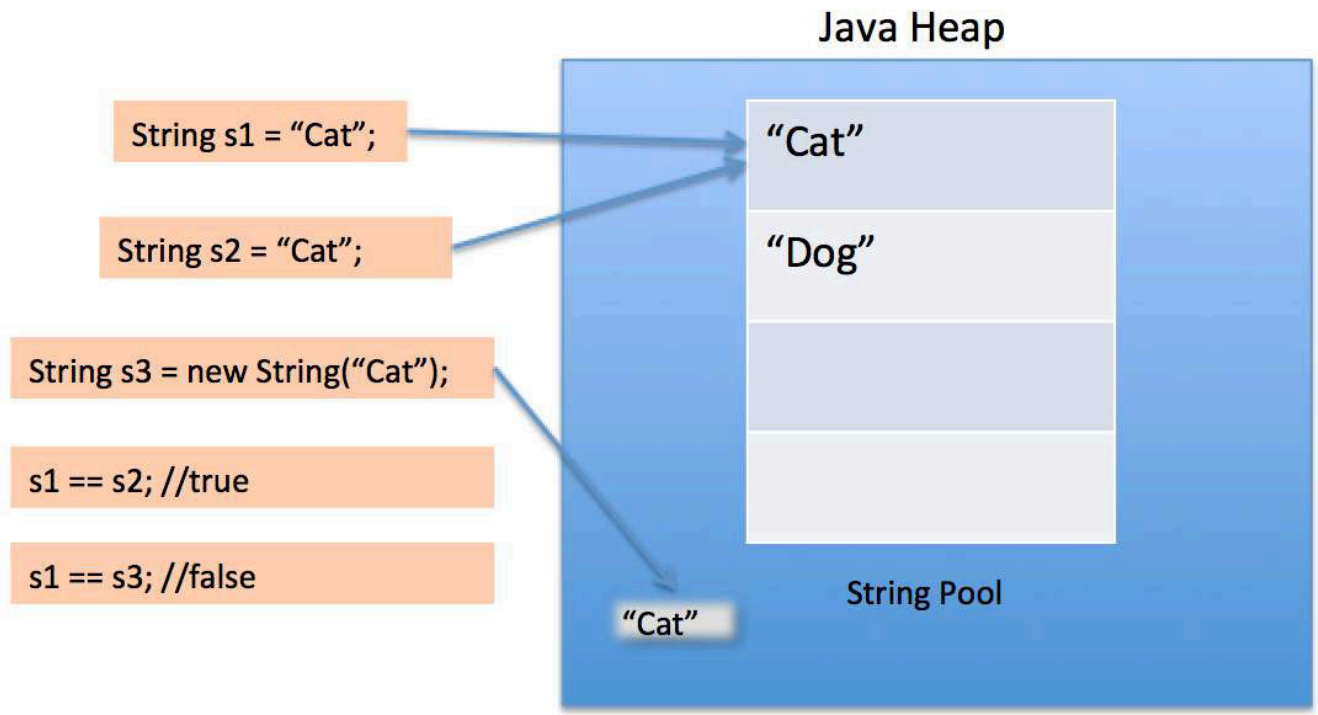
Like many Java objects, **all** `String` instances are created on the heap, even literals. When the JVM finds a `String` literal that has no equivalent reference in the heap, the JVM creates a corresponding `String` instance on the heap **and** it also stores a reference to the newly created `String` instance in the String pool. Any other references to the same `String` literal are replaced with the previously created `String` instance in the heap.

Let's look at the following example:

```
class Strings
{
    public static void main (String[] args)
    {
        String a = "alpha";
        String b = "alpha";
        String c = new String("alpha");

        //All three strings are equivalent
        System.out.println(a.equals(b) && b.equals(c));

        //Although only a and b reference the same heap object
        System.out.println(a == b);
        System.out.println(a != c);
        System.out.println(b != c);
    }
}
```

The output of the above is:

```
true
true
true
true
```



When we use double quotes to create a String, it first looks for String with same value in the String pool, if found it just returns the reference else it creates a new String in the pool and then returns the reference.

However using new operator, we force String class to create a new String object in heap space. We can use intern() method to put it into the pool or refer to other String object from string pool having same value.

The String pool itself is also created on the heap.

`Version < Java SE 7`

Before Java 7, `String` **literals** were stored in the runtime constant pool in the method area of `PermGen`, that had a fixed size.

The String pool also resided in `PermGen`.

`Version ≥ Java SE 7`

RFC: 6962931

> In JDK 7, interned strings are no longer allocated in the permanent generation of the Java heap, but are instead allocated in the main part of the Java heap (known as the young and old generations), along with the other objects created by the application. This change will result in more data residing in the main Java heap, and less data in the permanent generation, and thus may require heap sizes to be adjusted. Most applications will see only relatively small differences in heap usage due to this change, but larger applications that load many classes or make heavy use of the `String.intern()` method will see more significant differences.

Section 11.5: Splitting Strings

You can split a `String` on a particular delimiting character or a Regular Expression, you can use the `String.split()` method that has the following signature:

```
public String[] split(String regex)
```

Note that delimiting character or regular expression gets removed from the resulting String Array.

Example using delimiting character:

```
String lineFromCsvFile = "Mickey;Bolton;12345;121216";
String[] dataCells = lineFromCsvFile.split(";");
// Result is dataCells = { "Mickey", "Bolton", "12345", "121216"};
```

Example using regular expression:

```
String lineFromInput = "What    do you need    from me?";
String[] words = lineFromInput.split("\\s+"); // one or more space chars
// Result is words = {"What", "do", "you", "need", "from", "me?"};
```

You can even directly split a `String` literal:

```
String[] firstNames = "Mickey, Frank, Alicia, Tom".split(", ");
// Result is firstNames = {"Mickey", "Frank", "Alicia", "Tom"};
```

Warning: Do not forget that the parameter is always treated as a regular expression.

```
"aaa.bbb".split("."); // This returns an empty array
```

In the previous example `.` is treated as the regular expression wildcard that matches any character, and since every character is a delimiter, the result is an empty array.

Splitting based on a delimiter which is a regex meta-character

The following characters are considered special (aka meta-characters) in regex

```
< > - = ! ( ) [ ] { } \ ^ $ | ? * + .
```

To split a string based on one of the above delimiters, you need to either *escape* them using `\\` or use `Pattern.quote()`:

- Using `Pattern.quote()`:

```
String s = "a|b|c";
String regex = Pattern.quote("|");
String[] arr = s.split(regex);
```

- Escaping the special characters:

```
String s = "a|b|c";
String[] arr = s.split("\\|");
```

Split removes empty values

`split(delimiter)` by default removes trailing empty strings from result array. To turn this mechanism off we need to use overloaded version of `split(delimiter, limit)` with limit set to negative value like

```
String[] split = data.split("\\|", -1);
```

`split(regex)` internally returns result of `split(regex, 0)`.

The limit parameter controls the number of times the pattern is applied and therefore affects the length of the resulting array.
If the limit *n* is greater than zero then the pattern will be applied at most *n - 1* times, the array's length will be no greater than *n*, and the array's last entry will contain all input beyond the last matched delimiter.
If *n* is negative, then the pattern will be applied as many times as possible and the array can have any length.
If *n* is zero then the pattern will be applied as many times as possible, the array can have any length, and trailing empty strings will be discarded.

Splitting with a `StringTokenizer`

Besides the `split()` method Strings can also be split using a `StringTokenizer`.

`StringTokenizer` is even more restrictive than `String.split()`, and also a bit harder to use. It is essentially designed for pulling out tokens delimited by a fixed set of characters (given as a `String`). Each character will act as a separator. Because of this restriction, it's about twice as fast as `String.split()`.

Default set of characters are empty spaces (`\t\n\r\f`). The following example will print out each word separately.

```
String str = "the lazy fox jumped over the brown fence";
StringTokenizer tokenizer = new StringTokenizer(str);
while (tokenizer.hasMoreTokens()) {
    System.out.println(tokenizer.nextToken());
}
```

This will print out:

```
the
lazy
fox
jumped
over
the
brown
fence
```

You can use different character sets for separation.

```
String str = "jumped over";
// In this case character `u` and `e` will be used as delimiters
StringTokenizer tokenizer = new StringTokenizer(str, "ue");
while (tokenizer.hasMoreTokens()) {
    System.out.println(tokenizer.nextToken());
}
```

This will print out:

```
j
mp
d ov
r
```

Section 11.6: Joining Strings with a delimiter

Version ≥ Java SE 8

An array of strings can be joined using the static method `String.join()`:

```
String[] elements = { "foo", "bar", "foobar" };
String singleString = String.join(" + ", elements);

System.out.println(singleString); // Prints "foo + bar + foobar"
```

Similarly, there's an overloaded `String.join()` method for `Iterables`.

To have a fine-grained control over joining, you may use StringJoiner class:

```
StringJoiner sj = new StringJoiner(", ", "[", "]");
    // The last two arguments are optional,
    // they define prefix and suffix for the result string

sj.add("foo");
sj.add("bar");
sj.add("foobar");

System.out.println(sj); // Prints "[foo, bar, foobar]"
```

To join a stream of strings, you may use the joining collector:

```
Stream<String> stringStream = Stream.of("foo", "bar", "foobar");
String joined = stringStream.collect(Collectors.joining(", "));
System.out.println(joined); // Prints "foo, bar, foobar"
```

There's an option to define prefix and suffix here as well:

```
Stream<String> stringStream = Stream.of("foo", "bar", "foobar");
String joined = stringStream.collect(Collectors.joining(", ", "{", "}"));
System.out.println(joined); // Prints "{foo, bar, foobar}"
```

Section 11.7: String concatenation and StringBuilders

String concatenation can be performed using the `+` operator. For example:

```
String s1 = "a";
String s2 = "b";
String s3 = "c";
String s = s1 + s2 + s3; // abc
```

Normally a compiler implementation will perform the above concatenation using methods involving a `StringBuilder` under the hood. When compiled, the code would look similar to the below:

```
StringBuilder sb = new StringBuilder("a");
String s = sb.append("b").append("c").toString();
```

`StringBuilder` has several overloaded methods for appending different types, for example, to append an `int` instead of a `String`. For example, an implementation can convert:

```
String s1 = "a";
String s2 = "b";
String s = s1 + s2 + 2; // ab2
```

to the following:

```
StringBuilder sb = new StringBuilder("a");
String s = sb.append("b").append(2).toString();
```

The above examples illustrate a simple concatenation operation that is effectively done in a single place in the code. The concatenation involves a single instance of the `StringBuilder`. In some cases, a concatenation is carried out in a cumulative way such as in a loop:

```
String result = "";
for(int i = 0; i < array.length; i++) {
    result += extractElement(array[i]);
}
return result;
```

In such cases, the compiler optimization is usually not applied, and each iteration will create a new `StringBuilder` object. This can be optimized by explicitly transforming the code to use a single `StringBuilder`:

```
StringBuilder result = new StringBuilder();
for(int i = 0; i < array.length; i++) {
    result.append(extractElement(array[i]));
```

```
}
return result.toString();
```

A `StringBuilder` will be initialized with an empty space of only 16 characters. If you know in advance that you will be building larger strings, it can be beneficial to initialize it with sufficient size in advance, so that the internal buffer does not need to be resized:

```
StringBuilder buf = new StringBuilder(30); // Default is 16 characters
buf.append("0123456789");
buf.append("0123456789"); // Would cause a reallocation of the internal buffer otherwise
String result = buf.toString(); // Produces a 20-chars copy of the string
```

If you are producing many strings, it is advisable to reuse `StringBuilders`:

```
StringBuilder buf = new StringBuilder(100);
for (int i = 0; i < 100; i++) {
    buf.setLength(0); // Empty buffer
    buf.append("This is line ").append(i).append('\n');
    outputfile.write(buf.toString());
}
```

If (and only if) multiple threads are writing to the *same* buffer, use StringBuffer, which is a **`synchronized`** version of `StringBuilder`. But because usually only a single thread writes to a buffer, it is usually faster to use `StringBuilder` without synchronization.

Using concat() method:

```
String string1 = "Hello ";
String string2 = "world";
String string3 = string1.concat(string2);  // "Hello world"
```

This returns a new string that is string1 with string2 added to it at the end. You can also use the concat() method with string literals, as in:

```
"My name is ".concat("Buyya");
```

Section 11.8: Substrings

```
String s = "this is an example";
String a = s.substring(11); // a will hold the string starting at character 11 until the end
("example")
String b = s.substring(5, 10); // b will hold the string starting at character 5 and ending right
before character 10 ("is an")
String b = s.substring(5, b.length()-3); // b will hold the string starting at character 5 ending
right before b' s lenght is out of 3  ("is an exam")
```

Substrings may also be applied to slice and add/replace character into its original String. For instance, you faced a Chinese date containing Chinese characters but you want to store it as a well format Date String.

```
String datestring = "2015年11月17日"
datestring = datestring.substring(0, 4) + "-" + datestring.substring(5,7) + "-" +
datestring.substring(8,10);
//Result will be 2015-11-17
```

The substring method extracts a piece of a `String`. When provided one parameter, the parameter is the start and

the piece extends until the end of the `String`. When given two parameters, the first parameter is the starting character and the second parameter is the index of the character right after the end (the character at the index is not included). An easy way to check is the subtraction of the first parameter from the second should yield the expected length of the string.

Version < Java SE 7

In JDK <7u6 versions the `substring` method instantiates a `String` that shares the same backing `char[]` as the original `String` and has the internal `offset` and `count` fields set to the result start and length. Such sharing may cause memory leaks, that can be prevented by calling `new String(s.substring(...))` to force creation of a copy, after which the `char[]` can be garbage collected.

Version ≥ Java SE 7

From JDK 7u6 the `substring` method always copies the entire underlying `char[]` array, making the complexity linear compared to the previous constant one but guaranteeing the absence of memory leaks at the same time.

Section 11.9: Platform independent new line separator

Since the new line separator varies from platform to platform (e.g. `\n` on Unix-like systems or `\r\n` on Windows) it is often necessary to have a platform-independent way of accessing it. In Java it can be retrieved from a system property:

```
System.getProperty("line.separator")
```

Version ≥ Java SE 7

Because the new line separator is so commonly needed, from Java 7 on a shortcut method returning exactly the same result as the code above is available:

```
System.lineSeparator()
```

Note: Since it is very unlikely that the new line separator changes during the program's execution, it is a good idea to store it in in a static final variable instead of retrieving it from the system property every time it is needed.

When using `String.format`, use `%n` rather than `\n` or '\r\n' to output a platform independent new line separator.

```
System.out.println(String.format('line 1: %s.%nline 2: %s%n', lines[0],lines[1]));
```

Section 11.10: Reversing Strings

There are a couple ways you can reverse a string to make it backwards.

1. StringBuilder/StringBuffer:

```
String code = "code";
System.out.println(code);

StringBuilder sb = new StringBuilder(code);
code = sb.reverse().toString();

System.out.println(code);
```

2. Char array:

```
String code = "code";
System.out.println(code);

char[] array = code.toCharArray();
for (int index = 0, mirroredIndex = array.length - 1; index < mirroredIndex; index++,
mirroredIndex--) {
    char temp = array[index];
    array[index] = array[mirroredIndex];
    array[mirroredIndex] = temp;
}

// print reversed
System.out.println(new String(array));
```

Section 11.11: Adding toString() method for custom objects

Suppose you have defined the following `Person` class:

```
public class Person {

    String name;
    int age;

    public Person (int age, String name) {
        this.age = age;
        this.name = name;
    }
}
```

If you instantiate a new `Person` object:

```
Person person = new Person(25, "John");
```

and later in your code you use the following statement in order to print the object:

```
System.out.println(person.toString());
```

Live Demo on Ideone

you'll get an output similar to the following:

```
Person@7ab89d
```

This is the result of the implementation of the `toString()` method defined in the `Object` class, a superclass of `Person`. The documentation of `Object.toString()` states:

> The toString method for class Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character `@', and the unsigned hexadecimal representation of the hash code of the object. In other words, this method returns a string equal to the value of:
>
> ```
> getClass().getName() + '@' + Integer.toHexString(hashCode())
> ```

So, for meaningful output, you'll have to **override** the `toString()` method:

```
@Override
```

```
public String toString() {
    return "My name is " + this.name + " and my age is " + this.age;
}
```

Now the output will be:

```
My name is John and my age is 25
```

You can also write

```
System.out.println(person);
```

Live Demo on Ideone

In fact, `println()` implicitly invokes the `toString` method on the object.

Section 11.12: Remove Whitespace from the Beginning and End of a String

The `trim()` method returns a new String with the leading and trailing whitespace removed.

```
String s = new String("   Hello World!!  ");
String t = s.trim();  // t = "Hello World!!"
```

If you `trim` a String that doesn't have any whitespace to remove, you will be returned the same String instance.

Note that the `trim()` method has its own notion of whitespace, which differs from the notion used by the `Character.isWhitespace()` method:

- All ASCII control characters with codes `U+0000` to `U+0020` are considered whitespace and are removed by `trim()`. This includes `U+0020 'SPACE'`, `U+0009 'CHARACTER TABULATION'`, `U+000A 'LINE FEED'` and `U+000D 'CARRIAGE RETURN'` characters, but also the characters like `U+0007 'BELL'`.
- Unicode whitespace like `U+00A0 'NO-BREAK SPACE'` or `U+2003 'EM SPACE'` are *not* recognized by `trim()`.

Section 11.13: Case insensitive switch

`Version ≥ Java SE 7`

`switch` itself can not be parameterised to be case insensitive, but if absolutely required, can behave insensitive to the input string by using `toLowerCase()` or `toUpperCase`:

```
switch (myString.toLowerCase()) {
     case "case1" :
        ...
     break;
     case "case2" :
        ...
     break;
}
```

Beware

- `Locale` might affect how changing cases happen!

- Care must be taken not to have any uppercase characters in the labels - those will never get executed!

Section 11.14: Replacing parts of Strings

Two ways to replace: by regex or by exact match.

Note: the original String object will be unchanged, the return value holds the changed String.

Exact match

Replace single character with another single character:

```
String replace(char oldChar, char newChar)
```

> Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.

```
String s = "popcorn";
System.out.println(s.replace('p','W'));
```

Result:

```
WoWcorn
```

Replace sequence of characters with another sequence of characters:

```
String replace(CharSequence target, CharSequence replacement)
```

> Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence.

```
String s = "metal petal et al.";
System.out.println(s.replace("etal","etallica"));
```

Result:

```
metallica petallica et al.
```

Regex

Note: the grouping uses the $ character to reference the groups, like $1.

Replace all matches:

```
String replaceAll(String regex, String replacement)
```

> Replaces each substring of this string that matches the given regular expression with the given replacement.

```
String s = "spiral metal petal et al.";
System.out.println(s.replaceAll("(\\w*etal)","$1lica"));
```

Result:

```
spiral metallica petallica et al.
```

Replace first match only:

```
String replaceFirst(String regex, String replacement)
```

> Replaces the first substring of this string that matches the given regular expression with the given replacement

```
String s = "spiral metal petal et al.";
System.out.println(s.replaceAll("(\\w*etal)","$1lica"));
```

Result:

```
spiral metallica petal et al.
```

Section 11.15: Getting the length of a String

In order to get the length of a `String` object, call the `length()` method on it. The length is equal to the number of UTF-16 code units (chars) in the string.

```
String str = "Hello, World!";
System.out.println(str.length()); // Prints out 13
```

Live Demo on Ideone

A **`char`** in a String is UTF-16 value. Unicode codepoints whose values are ≥ 0x1000 (for example, most emojis) use two char positions. To count the number of Unicode codepoints in a String, regardless of whether each codepoint fits in a UTF-16 **`char`** value, you can use the `codePointCount` method:

```
int length = str.codePointCount(0, str.length());
```

You can also use a Stream of codepoints, as of Java 8:

```
int length = str.codePoints().count();
```

Section 11.16: Getting the nth character in a String

```
String str = "My String";

System.out.println(str.charAt(0)); // "M"
System.out.println(str.charAt(1)); // "y"
System.out.println(str.charAt(2)); // " "
System.out.println(str.charAt(str.length-1)); // Last character "g"
```

To get the nth character in a string, simply call `charAt(n)` on a `String`, where `n` is the index of the character you would like to retrieve

NOTE: index `n` is starting at `0`, so the first element is at n=0.

Section 11.17: Counting occurrences of a substring or

character in a string

`countMatches` method from org.apache.commons.lang3.StringUtils is typically used to count occurrences of a substring or character in a `String`:

```
import org.apache.commons.lang3.StringUtils;

String text = "One fish, two fish, red fish, blue fish";

// count occurrences of a substring
String stringTarget = "fish";
int stringOccurrences = StringUtils.countMatches(text, stringTarget); // 4

// count occurrences of a char
char charTarget = ',';
int charOccurrences = StringUtils.countMatches(text, charTarget); // 3
```

Otherwise for does the same with standard Java API's you could use Regular Expressions:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

String text = "One fish, two fish, red fish, blue fish";
System.out.println(countStringInString("fish", text)); // prints 4
System.out.println(countStringInString(",", text)); // prints 3


public static int countStringInString(String search, String text) {
    Pattern pattern = Pattern.compile(search);
    Matcher matcher = pattern.matcher(text);

    int stringOccurrences = 0;
    while (matcher.find()) {
      stringOccurrences++;
    }
    return stringOccurrences;
}
```

Chapter 12: StringBuffer

Introduction to Java StringBuffer class.

Section 12.1: String Buffer class

Key Points:

- used to created mutable (modifiable) string.
- **Mutable**: Which can be changed.
- is thread-safe i.e. multiple threads cannot access it simultaneously.

Methods:

- public synchronized StringBuffer append(String s)
- public synchronized StringBuffer insert(int offset, String s)
- public synchronized StringBuffer replace(int startIndex, int endIndex, String str)
- public synchronized StringBuffer delete(int startIndex, int endIndex)
- public synchronized StringBuffer reverse()
- public int capacity()
- public void ensureCapacity(int minimumCapacity)
- public char charAt(int index)
- public int length()
- public String substring(int beginIndex)
- public String substring(int beginIndex, int endIndex)

Example Showing difference between String and String Buffer implementation:

```
class Test {
 public static void main(String args[])
 {
  String str = "study";
  str.concat("tonight");
  System.out.println(str);      // Output: study

  StringBuffer strB = new StringBuffer("study");
  strB.append("tonight");
  System.out.println(strB);    // Output: studytonight
 }
}
```

Chapter 13: StringBuilder

Java StringBuilder class is used to create mutable (modifiable) string. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK 1.5.

Section 13.1: Comparing StringBuffer, StringBuilder, Formatter and StringJoiner

The `StringBuffer`, `StringBuilder`, `Formatter` and `StringJoiner` classes are Java SE utility classes that are primarily used for assembling strings from other information:

- The `StringBuffer` class has been present since Java 1.0, and provides a variety of methods for building and modifying a "buffer" containing a sequence of characters.
- The `StringBuilder` class was added in Java 5 to address performance issues with the original `StringBuffer` class. The APIs for the two clases are essentially the same. The main difference between `StringBuffer` and `StringBuilder` is that the former is thread-safe and synchronized and the latter is not.

This example shows how `StringBuilder` is can be used:

```
int one = 1;
String color = "red";
StringBuilder sb = new StringBuilder();
sb.append("One=").append(one).append(", Color=").append(color).append('\n');
System.out.print(sb);
// Prints "One=1, Colour=red" followed by an ASCII newline.
```

(The `StringBuffer` class is used the same way: just change `StringBuilder` to `StringBuffer` in the above)

The `StringBuffer` and `StringBuilder` classes are suitable for both assembling and modifying strings; i.e they provide methods for replacing and removing characters as well as adding them in various. The remining two classes are specific to the task of assembling strings.

- The `Formatter` class was added in Java 5, and is loosely modeled on the `sprintf` function in the C standard library. It takes a *format* string with embedded *format specifiers* and a sequences of other arguments, and generates a string by converting the arguments into text and substituting them in place of the format specifiers. The details of the format specifiers say how the arguments are converted into text.
- The `StringJoiner` class was added in Java 8. It is a special purpose formatter that succinctly formats a sequence of strings with separators between them. It is designed with a *fluent* API, and can be used with Java 8 streams.

Here are some typical examples of `Formatter` usage:

```
// This does the same thing as the StringBuilder example above
int one = 1;
String color = "red";
Formatter f = new Formatter();
System.out.print(f.format("One=%d, colour=%s%n", one, color));
// Prints "One=1, Colour=red" followed by the platform's line separator

// The same thing using the `String.format` convenience method
System.out.print(String.format("One=%d, color=%s%n", one, color));
```

The `StringJoiner` class is not ideal for the above task, so here is an example of a formatting an array of strings.

```
StringJoiner sj = new StringJoiner(", ", "[", "]");
for (String s : new String[]{"A", "B", "C"}) {
    sj.add(s);
}
System.out.println(sj);
// Prints "[A, B, C]"
```

The use-cases for the 4 classes can be summarized:

- `StringBuilder` suitable for any string assembly OR string modification task.
- `StringBuffer` use (only) when you require a thread-safe version of `StringBuilder`.
- `Formatter` provides much richer string formatting functionality, but is not as efficient as `StringBuilder`. This is because each call to `Formatter.format(...)` entails:
    - parsing the `format` string,
    - creating and populate a *varargs* array, and
    - autoboxing any primitive type arguments.
- `StringJoiner` provides succinct and efficient formatting of a sequence of strings with separators, but is not suitable for other formatting tasks.

Section 13.2: Repeat a String n times

Problem: Create a `String` containing `n` repetitions of a `String s`.

The trivial approach would be repeatedly concatenating the `String`

```
final int n = ...
final String s = ...
String result = "";

for (int i = 0; i < n; i++) {
    result += s;
}
```

This creates `n` new string instances containing 1 to `n` repetitions of `s` resulting in a runtime of $O(s.length() * n^2) = O(s.length() * (1+2+...+(n-1)+n))$.

To avoid this `StringBuilder` should be used, which allows creating the `String` in $O(s.length() * n)$ instead:

```
final int n = ...
final String s = ...

StringBuilder builder = new StringBuilder();

for (int i = 0; i < n; i++) {
    builder.append(s);
}

String result = builder.toString();
```

Chapter 14: String Tokenizer

The java.util.StringTokenizer class allows you to break a string into tokens. It is simple way to break string.

The set of delimiters (the characters that separate tokens) may be specified either at creation time or on a per-token basis.

Section 14.1: StringTokenizer Split by space

```
import java.util.StringTokenizer;
public class Simple{
 public static void main(String args[]){
   StringTokenizer st = new StringTokenizer("apple ball cat dog"," ");
     while (st.hasMoreTokens()) {
         System.out.println(st.nextToken());
     }
   }
}
```

Output:

apple

ball

cat

dog

Section 14.2: StringTokenizer Split by comma ','

```
public static void main(String args[]) {
        StringTokenizer st = new StringTokenizer("apple,ball cat,dog", ",");
        while (st.hasMoreTokens()) {
            System.out.println(st.nextToken());
        }
    }
```

Output:

apple

ball cat

dog

Chapter 15: Splitting a string into fixed length parts

Section 15.1: Break a string up into substrings all of a known length

The trick is to use a look-behind with the regex `\G`, which means "end of previous match":

```
String[] parts = str.split("(?<=\\G.{8})");
```

The regex matches 8 characters after the end of the last match. Since in this case the match is zero-width, we could more simply say "8 characters after the last match".

Conveniently, `\G` is initialized to start of input, so it works for the first part of the input too.

Section 15.2: Break a string up into substrings all of variable length

Same as the known length example, but insert the length into regex:

```
int length = 5;
String[] parts = str.split("(?<=\\G.{" + length + "})");
```

Chapter 16: Date Class

Parameter	Explanation
No parameter	Creates a new Date object using the allocation time (to the nearest millisecond)
long date	Creates a new Date object with the time set to the number of milliseconds since "the epoch" (January 1, 1970, 00:00:00 GMT)

Section 16.1: Convert java.util.Date to java.sql.Date

`java.util.Date` to `java.sql.Date` conversion is usually necessary when a Date object needs to be written in a database.

`java.sql.Date` is a wrapper around millisecond value and is used by `JDBC` to identify an `SQL DATE` type

In the below example, we use the `java.util.Date()` constructor, that creates a Date object and initializes it to represent time to the nearest millisecond. This date is used in the `convert(java.util.Date utilDate)` method to return a `java.sql.Date` object

Example

```
public class UtilToSqlConversion {

    public static void main(String args[])
    {
        java.util.Date utilDate = new java.util.Date();
        System.out.println("java.util.Date is : " + utilDate);
        java.sql.Date sqlDate = convert(utilDate);
        System.out.println("java.sql.Date is : " + sqlDate);
        DateFormat df = new SimpleDateFormat("dd/MM/YYYY - hh:mm:ss");
        System.out.println("dateFormated date is : " + df.format(utilDate));
    }

    private static java.sql.Date convert(java.util.Date uDate) {
        java.sql.Date sDate = new java.sql.Date(uDate.getTime());
        return sDate;
    }

}
```

Output

```
java.util.Date is : Fri Jul 22 14:40:35 IST 2016
java.sql.Date is : 2016-07-22
dateFormated date is : 22/07/2016 - 02:40:35
```

`java.util.Date` has both date and time information, whereas `java.sql.Date` only has date information

Section 16.2: A basic date output

Using the following code with the format string `yyyy/MM/dd hh:mm.ss`, we will receive the following output

> 2016/04/19 11:45.36

```
// define the format to use
String formatString = "yyyy/MM/dd hh:mm.ss";

// get a current date object
Date date = Calendar.getInstance().getTime();

// create the formatter
SimpleDateFormat simpleDateFormat = new SimpleDateFormat(formatString);

// format the date
String formattedDate = simpleDateFormat.format(date);

// print it
System.out.println(formattedDate);


// single-line version of all above code
System.out.println(new SimpleDateFormat("yyyy/MM/dd
hh:mm.ss").format(Calendar.getInstance().getTime()));
```

Section 16.3: Java 8 LocalDate and LocalDateTime objects

Date and LocalDate objects **cannot** be *exactly* converted between each other since a Date object represents both a specific day and time, while a LocalDate object does not contain time or timezone information. However, it can be useful to convert between the two if you only care about the actual date information and not the time information.

Creates a LocalDate

```
// Create a default date
LocalDate lDate = LocalDate.now();

// Creates a date from values
lDate = LocalDate.of(2017, 12, 15);

// create a date from string
lDate = LocalDate.parse("2017-12-15");

// creates a date from zone
LocalDate.now(ZoneId.systemDefault());
```

Creates a LocalDateTime

```
// Create a default date time
LocalDateTime lDateTime = LocalDateTime.now();

// Creates a date time from values
lDateTime = LocalDateTime.of(2017, 12, 15, 11, 30);

// create a date time from string
lDateTime = LocalDateTime.parse("2017-12-05T11:30:30");

// create a date time from zone
LocalDateTime.now(ZoneId.systemDefault());
```

LocalDate to Date and vice-versa

```
Date date = Date.from(Instant.now());
ZoneId defaultZoneId = ZoneId.systemDefault();
```

```
// Date to LocalDate
LocalDate localDate = date.toInstant().atZone(defaultZoneId).toLocalDate();

// LocalDate to Date
Date.from(localDate.atStartOfDay(defaultZoneId).toInstant());
```

LocalDateTime to Date and vice-versa

```
Date date = Date.from(Instant.now());
ZoneId defaultZoneId = ZoneId.systemDefault();

// Date to LocalDateTime
LocalDateTime localDateTime =  date.toInstant().atZone(defaultZoneId).toLocalDateTime();

// LocalDateTime to Date
Date out = Date.from(localDateTime.atZone(defaultZoneId).toInstant());
```

Section 16.4: Creating a Specific Date

While the Java Date class has several constructors, you'll notice that most are deprecated. The only acceptable way of creating a Date instance directly is either by using the empty constructor or passing in a long (number of milliseconds since standard base time). Neither are handy unless you're looking for the current date or have another Date instance already in hand.

To create a new date, you will need a Calendar instance. From there you can set the Calendar instance to the date that you need.

```
Calendar c = Calendar.getInstance();
```

This returns a new Calendar instance set to the current time. Calendar has many methods for mutating it's date and time or setting it outright. In this case, we'll set it to a specific date.

```
c.set(1974, 6, 2, 8, 0, 0);
Date d = c.getTime();
```

The `getTime` method returns the Date instance that we need. Keep in mind that the Calendar set methods only set one or more fields, they do not set them all. That is, if you set the year, the other fields remain unchanged.

PITFALL

In many cases, this code snippet fulfills its purpose, but keep in mind that two important parts of the date/time are not defined.

- the `(1974, 6, 2, 8, 0, 0)` parameters are interpreted within the default timezone, defined somewhere else,
- the milliseconds are not set to zero, but filled from the system clock at the time the Calendar instance is created.

Section 16.5: Converting Date to a certain String format

`format()` from `SimpleDateFormat` class helps to convert a `Date` object into certain format `String` object by using the supplied *pattern string*.

```
Date today = new Date();
```

```
SimpleDateFormat dateFormat = new SimpleDateFormat("dd-MMM-yy"); //pattern is specified here
System.out.println(dateFormat.format(today)); //25-Feb-16
```

Patterns can be applied again by using `applyPattern()`

```
dateFormat.applyPattern("dd-MM-yyyy");
System.out.println(dateFormat.format(today)); //25-02-2016

dateFormat.applyPattern("dd-MM-yyyy HH:mm:ss E");
System.out.println(dateFormat.format(today)); //25-02-2016 06:14:33 Thu
```

Note: Here `mm` (small letter m) denotes minutes and `MM` (capital M) denotes month. Pay careful attention when formatting years: capital "Y" (`Y`) indicates the "week in the year" while lower-case "y" (`y`) indicates the year.

Section 16.6: LocalTime

To use just the time part of a Date use LocalTime. You can instantiate a LocalTime object in a couple ways

1. `LocalTime time = LocalTime.now();`
2. `time = LocalTime.MIDNIGHT;`
3. `time = LocalTime.NOON;`
4. `time = LocalTime.of(12, 12, 45);`

`LocalTime` also has a built in toString method that displays the format very nicely.

```
System.out.println(time);
```

you can also get, add and subtract hours, minutes, seconds, and nanoseconds from the LocalTime object i.e.

```
time.plusMinutes(1);
time.getMinutes();
time.minusMinutes(1);
```

You can turn it into a Date object with the following code:

```
LocalTime lTime = LocalTime.now();
Instant instant = lTime.atDate(LocalDate.of(A_YEAR, A_MONTH, A_DAY)).
        atZone(ZoneId.systemDefault()).toInstant();
Date time = Date.from(instant);
```

this class works very nicely within a timer class to simulate an alarm clock.

Section 16.7: Convert formatted string representation of date to Date object

This method can be used to convert a formatted string representation of a date into a `Date` object.

```
    /**
     * Parses the date using the given format.
     *
     * @param formattedDate the formatted date string
     * @param dateFormat the date format which was used to create the string.
     * @return the date
     */
    public static Date parseDate(String formattedDate, String dateFormat) {
        Date date = null;
```

```
        SimpleDateFormat objDf = new SimpleDateFormat(dateFormat);
        try {
          date = objDf.parse(formattedDate);
        } catch (ParseException e) {
          // Do what ever needs to be done with exception.
        }
        return date;
    }
```

Section 16.8: Creating Date objects

```
Date date = new Date();
System.out.println(date); // Thu Feb 25 05:03:59 IST 2016
```

Here this `Date` object contains the current date and time when this object was created.

```
Calendar calendar = Calendar.getInstance();
calendar.set(90, Calendar.DECEMBER, 11);
Date myBirthDate = calendar.getTime();
System.out.println(myBirthDate); // Mon Dec 31 00:00:00 IST 1990
```

`Date` objects are best created through a `Calendar` instance since the use of the data constructors is deprecated and discouraged. To do se we need to get an instance of the `Calendar` class from the factory method. Then we can set year, month and day of month by using numbers or in case of months constants provided py the Calendar class to improve readability and reduce errors.

```
calendar.set(90, Calendar.DECEMBER, 11, 8, 32, 35);
Date myBirthDatenTime = calendar.getTime();
System.out.println(myBirthDatenTime); // Mon Dec 31 08:32:35 IST 1990
```

Along with date, we can also pass time in the order of hour, minutes and seconds.

Section 16.9: Comparing Date objects

Calendar, Date, and LocalDate

Version < Java SE 8

before, after, compareTo and equals methods

```
//Use of Calendar and Date objects
final Date today = new Date();
final Calendar calendar = Calendar.getInstance();
calendar.set(1990, Calendar.NOVEMBER, 1, 0, 0, 0);
Date birthdate = calendar.getTime();

final Calendar calendar2 = Calendar.getInstance();
calendar2.set(1990, Calendar.NOVEMBER, 1, 0, 0, 0);
Date samebirthdate = calendar2.getTime();

//Before example
System.out.printf("Is %1$tF before %2$tF? %3$b%n", today, birthdate,
Boolean.valueOf(today.before(birthdate)));
System.out.printf("Is %1$tF before %1$tF? %3$b%n", today, today,
Boolean.valueOf(today.before(today)));
System.out.printf("Is %2$tF before %1$tF? %3$b%n", today, birthdate,
Boolean.valueOf(birthdate.before(today)));

//After example
System.out.printf("Is %1$tF after %2$tF? %3$b%n", today, birthdate,
```

```
Boolean.valueOf(today.after(birthdate)));
System.out.printf("Is %1$tF after %1$tF? %3$b%n", today, birthdate,
Boolean.valueOf(today.after(today)));
System.out.printf("Is %2$tF after %1$tF? %3$b%n", today, birthdate,
Boolean.valueOf(birthdate.after(today)));

//Compare example
System.out.printf("Compare %1$tF to %2$tF: %3$d%n", today, birthdate,
Integer.valueOf(today.compareTo(birthdate)));
System.out.printf("Compare %1$tF to %1$tF: %3$d%n", today, birthdate,
Integer.valueOf(today.compareTo(today)));
System.out.printf("Compare %2$tF to %1$tF: %3$d%n", today, birthdate,
Integer.valueOf(birthdate.compareTo(today)));

//Equal example
System.out.printf("Is %1$tF equal to %2$tF? %3$b%n", today, birthdate,
Boolean.valueOf(today.equals(birthdate)));
System.out.printf("Is %1$tF equal to %2$tF? %3$b%n", birthdate, samebirthdate,
            Boolean.valueOf(birthdate.equals(samebirthdate)));
System.out.printf(
            "Because birthdate.getTime() -> %1$d is different from samebirthdate.getTime() -> %2$d,
there are millisecondes!%n",
            Long.valueOf(birthdate.getTime()), Long.valueOf(samebirthdate.getTime()));

//Clear ms from calendars
calendar.clear(Calendar.MILLISECOND);
calendar2.clear(Calendar.MILLISECOND);
birthdate = calendar.getTime();
samebirthdate = calendar2.getTime();

System.out.printf("Is %1$tF equal to %2$tF after clearing ms? %3$b%n", birthdate, samebirthdate,
            Boolean.valueOf(birthdate.equals(samebirthdate)));
```

Version ≥ Java SE 8

isBefore, isAfter, compareTo and equals methods

```
//Use of LocalDate
final LocalDate now = LocalDate.now();
final LocalDate birthdate2 = LocalDate.of(2012, 6, 30);
final LocalDate birthdate3 = LocalDate.of(2012, 6, 30);

//Hours, minutes, second and nanoOfsecond can also be configured with an other class LocalDateTime
//LocalDateTime.of(year, month, dayOfMonth, hour, minute, second, nanoOfSecond);

//isBefore example
System.out.printf("Is %1$tF before %2$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.isBefore(birthdate2)));
System.out.printf("Is %1$tF before %1$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.isBefore(now)));
System.out.printf("Is %2$tF before %1$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(birthdate2.isBefore(now)));

//isAfter example
System.out.printf("Is %1$tF after %2$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.isAfter(birthdate2)));
System.out.printf("Is %1$tF after %1$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.isAfter(now)));
System.out.printf("Is %2$tF after %1$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(birthdate2.isAfter(now)));

//compareTo example
System.out.printf("Compare %1$tF to %2$tF %3$d%n", now, birthdate2,
Integer.valueOf(now.compareTo(birthdate2)));
```

```
System.out.printf("Compare %1$tF to %1$tF %3$d%n", now, birthdate2,
Integer.valueOf(now.compareTo(now)));
System.out.printf("Compare %2$tF to %1$tF %3$d%n", now, birthdate2,
Integer.valueOf(birthdate2.compareTo(now)));

//equals example
System.out.printf("Is %1$tF equal to %2$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.equals(birthdate2)));
System.out.printf("Is %1$tF to %2$tF? %3$b%n", birthdate2, birthdate3,
Boolean.valueOf(birthdate2.equals(birthdate3)));

//isEqual example
System.out.printf("Is %1$tF equal to %2$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.isEqual(birthdate2)));
System.out.printf("Is %1$tF to %2$tF? %3$b%n", birthdate2, birthdate3,
Boolean.valueOf(birthdate2.isEqual(birthdate3)));
```

Date comparison before Java 8

Before Java 8, dates could be compared using java.util.Calendar and java.util.Date classes. Date class offers 4 methods to compare dates :

- after(Date when)
- before(Date when)
- compareTo(Date anotherDate)
- equals(Object obj)

`after`, `before`, `compareTo` and `equals` methods compare the values returned by getTime() method for each date.

`compareTo` method returns positive integer.

- Value greater than 0 : when the Date is after the Date argument
- Value greater than 0 : when the Date is before the Date argument
- Value equals to 0 : when the Date is equal to the Date argument

`equals` results can be surprising as shown in the example because values, like milliseconds, are not initialize with the same value if not explicitly given.

Since Java 8

With Java 8 a new Object to work with Date is available java.time.LocalDate. `LocalDate` implements ChronoLocalDate, the abstract representation of a date where the Chronology, or calendar system, is pluggable.

To have the date time precision the Object java.time.LocalDateTime has to be used. `LocalDate` and `LocalDateTime` use the same methods name for comparing.

Comparing dates using a `LocalDate` is different from using `ChronoLocalDate` because the chronology, or calendar system are not taken in account the first one.

Because most application should use `LocalDate`, `ChronoLocalDate` is not included in examples. Further reading here.

> Most applications should declare method signatures, fields and variables as LocalDate, not this[ChronoLocalDate] interface.

`LocalDate` has 5 methods to compare dates :

- isAfter(ChronoLocalDate other)
- isBefore(ChronoLocalDate other)
- isEqual(ChronoLocalDate other)
- compareTo(ChronoLocalDate other)
- equals(Object obj)

In case of `LocalDate` parameter, `isAfter`, `isBefore`, `isEqual`, `equals` and `compareTo` now use this method:

```
int compareTo0(LocalDate otherDate) {
    int cmp = (year - otherDate.year);
    if (cmp == 0) {
        cmp = (month - otherDate.month);
        if (cmp == 0) {
            cmp = (day - otherDate.day);
        }
    }
    return cmp;
}
```

`equals` method check if the parameter reference equals the date first whereas `isEqual` directly calls `compareTo0`.

In case of an other class instance of `ChronoLocalDate` the dates are compared using the `Epoch Day`. The Epoch Day count is a simple incrementing count of days where day 0 is 1970-01-01 (ISO).

Section 16.10: Converting String into Date

`parse()` from `SimpleDateFormat` class helps to convert a `String` pattern into a `Date` object.

```
DateFormat dateFormat = DateFormat.getDateInstance(DateFormat.SHORT, Locale.US);
String dateStr = "02/25/2016"; // input String
Date date = dateFormat.parse(dateStr);
System.out.println(date.getYear()); // 116
```

There are 4 different styles for the text format, `SHORT`, `MEDIUM` (this is the default), `LONG` and `FULL`, all of which depend on the locale. If no locale is specified, the system default locale is used.

Style	Locale.US	Locale.France
SHORT	6/30/09	30/06/09
MEDIUM	Jun 30, 2009	30 juin 2009
LONG	June 30, 2009	30 juin 2009
FULL	Tuesday, June 30, 2009	mardi 30 juin 2009

Section 16.11: Time Zones and java.util.Date

A `java.util.Date` object *does not* have a concept of time zone.

- There is no way to **set** a timezone for a Date
- There is no way to **change** the timezone of a Date object
- A Date object created with the `new Date()` default constructor will be initialised with the current time in the system default timezone

However, it is possible to display the date represented by the point in time described by the Date object in a different time zone using e.g. `java.text.SimpleDateFormat`:

```
Date date = new Date();
```

```
//print default time zone
System.out.println(TimeZone.getDefault().getDisplayName());
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss"); //note: time zone not in
format!
//print date in the original time zone
System.out.println(sdf.format(date));
//current time in London
sdf.setTimeZone(TimeZone.getTimeZone("Europe/London"));
System.out.println(sdf.format(date));
```

Output:

```
Central European Time
2016-07-21 22:50:56
2016-07-21 21:50:56
```

Chapter 17: Dates and Time (java.time.*)

Section 17.1: Calculate Difference between 2 LocalDates

Use `LocalDate` and `ChronoUnit`:

```
LocalDate d1 = LocalDate.of(2017, 5, 1);
LocalDate d2 = LocalDate.of(2017, 5, 18);
```

now, since the method `between` of the `ChronoUnit` enumerator takes 2 `Temporals` as parameters so you can pass without a problem the `LocalDate` instances

```
long days = ChronoUnit.DAYS.between(d1, d2);
System.out.println( days );
```

Section 17.2: Date and time

Date and time without time zone information

```
LocalDateTime dateTime = LocalDateTime.of(2016, Month.JULY, 27, 8, 0);
LocalDateTime now = LocalDateTime.now();
LocalDateTime parsed = LocalDateTime.parse("2016-07-27T07:00:00");
```

Date and time with time zone information

```
ZoneId zoneId = ZoneId.of("UTC+2");
ZonedDateTime dateTime = ZonedDateTime.of(2016, Month.JULY, 27, 7, 0, 0, 235, zoneId);
ZonedDateTime composition = ZonedDateTime.of(localDate, localTime, zoneId);
ZonedDateTime now = ZonedDateTime.now(); // Default time zone
ZonedDateTime parsed = ZonedDateTime.parse("2016-07-27T07:00:00+01:00[Europe/Stockholm]");
```

Date and time with offset information (i.e. no DST changes taken into account)

```
ZoneOffset zoneOffset = ZoneOffset.ofHours(2);
OffsetDateTime dateTime = OffsetDateTime.of(2016, 7, 27, 7, 0, 0, 235, zoneOffset);
OffsetDateTime composition = OffsetDateTime.of(localDate, localTime, zoneOffset);
OffsetDateTime now = OffsetDateTime.now(); // Offset taken from the default ZoneId
OffsetDateTime parsed = OffsetDateTime.parse("2016-07-27T07:00:00+02:00");
```

Section 17.3: Operations on dates and times

```
LocalDate tomorrow = LocalDate.now().plusDays(1);
LocalDateTime anHourFromNow = LocalDateTime.now().plusHours(1);
Long daysBetween = java.time.temporal.ChronoUnit.DAYS.between(LocalDate.now(),
LocalDate.now().plusDays(3)); // 3
Duration duration = Duration.between(Instant.now(),
ZonedDateTime.parse("2016-07-27T07:00:00+01:00[Europe/Stockholm]"))
```

Section 17.4: Instant

Represents an instant in time. Can be thought of as a wrapper around a Unix timestamp.

```
Instant now = Instant.now();
Instant epoch1 = Instant.ofEpochMilli(0);
```

```
Instant epoch2 = Instant.parse("1970-01-01T00:00:00Z");
java.time.temporal.ChronoUnit.MICROS.between(epoch1, epoch2); // 0
```

Section 17.5: Usage of various classes of Date Time API

Following example also have explanation required for understanding example within it.

```
import java.time.Clock;
import java.time.Duration;
import java.time.Instant;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.util.TimeZone;
public class SomeMethodsExamples {

/**
 * Has the methods of the class {@link LocalDateTime}
 */
public static void checkLocalDateTime() {
    LocalDateTime localDateTime = LocalDateTime.now();
    System.out.println("Local Date time using static now() method ::: >>> "
            + localDateTime);

    LocalDateTime ldt1 = LocalDateTime.now(ZoneId.of(ZoneId.SHORT_IDS
            .get("AET")));
    System.out
            .println("LOCAL TIME USING now(ZoneId zoneId) method ::: >>>>"
                    + ldt1);

    LocalDateTime ldt2 = LocalDateTime.now(Clock.system(ZoneId
            .of(ZoneId.SHORT_IDS.get("PST"))));
    System.out
            .println("Local TIME USING now(Clock.system(ZoneId.of())) ::: >>>> "
                    + ldt2);

    System.out
            .println("Following is a static map in ZoneId class which has mapping of short timezone
names to their Actual timezone names");
    System.out.println(ZoneId.SHORT_IDS);

}

/**
 * This has the methods of the class {@link LocalDate}
 */
public static void checkLocalDate() {
    LocalDate localDate = LocalDate.now();
    System.out.println("Gives date without Time using now() method. >> "
            + localDate);
    LocalDate localDate2 = LocalDate.now(ZoneId.of(ZoneId.SHORT_IDS
            .get("ECT")));
    System.out
            .println("now() is overridden to take ZoneID as parametere using this we can get the
same date under different timezones. >> "
                    + localDate2);
}

/**
```

```java
 * This has the methods of abstract class {@link Clock}. Clock can be used
 * for time which has time with {@link TimeZone}.
 */
public static void checkClock() {
    Clock clock = Clock.systemUTC();
    // Represents time according to ISO 8601
    System.out.println("Time using Clock class : " + clock.instant());
}

/**
 * This has the {@link Instant} class methods.
 */
public static void checkInstant() {
    Instant instant = Instant.now();

    System.out.println("Instant using now() method :: " + instant);

    Instant ins1 = Instant.now(Clock.systemUTC());

    System.out.println("Instants using now(Clock clock) :: " + ins1);

}

/**
 * This class checks the methods of the {@link Duration} class.
 */
public static void checkDuration() {
    // toString() converts the duration to PTnHnMnS format according to ISO
    // 8601 standard. If a field is zero its ignored.

    // P is the duration designator (historically called "period") placed at
    // the start of the duration representation.
    // Y is the year designator that follows the value for the number of
    // years.
    // M is the month designator that follows the value for the number of
    // months.
    // W is the week designator that follows the value for the number of
    // weeks.
    // D is the day designator that follows the value for the number of
    // days.
    // T is the time designator that precedes the time components of the
    // representation.
    // H is the hour designator that follows the value for the number of
    // hours.
    // M is the minute designator that follows the value for the number of
    // minutes.
    // S is the second designator that follows the value for the number of
    // seconds.

    System.out.println(Duration.ofDays(2));
}

/**
 * Shows Local time without date. It doesn't store or represenet a date and
 * time. Instead its a representation of Time like clock on the wall.
 */
public static void checkLocalTime() {
    LocalTime localTime = LocalTime.now();
    System.out.println("LocalTime :: " + localTime);
}

/**
```

```
 * A date time with Time zone details in ISO-8601 standards.
 */
public static void checkZonedDateTime() {
    ZonedDateTime zonedDateTime = ZonedDateTime.now(ZoneId
            .of(ZoneId.SHORT_IDS.get("CST")));
    System.out.println(zonedDateTime);

}
}
```

Section 17.6: Date Time Formatting

Before Java 8, there was `DateFormat` and `SimpleDateFormat` classes in the package `java.text` and this legacy code will be continued to be used for sometime.

But, Java 8 offers a modern approach to handling Formatting and Parsing.

In formatting and parsing first you pass a `String` object to `DateTimeFormatter`, and in turn use it for formatting or parsing.

```
import java.time.*;
import java.time.format.*;

class DateTimeFormat
{
    public static void main(String[] args) {

        //Parsing
        String pattern = "d-MM-yyyy HH:mm";
        DateTimeFormatter dtF1 = DateTimeFormatter.ofPattern(pattern);

        LocalDateTime ldp1 = LocalDateTime.parse("2014-03-25T01:30"), //Default format
                      ldp2 = LocalDateTime.parse("15-05-2016 13:55",dtF1); //Custom format

        System.out.println(ldp1 + "\n" + ldp2); //Will be printed in Default format

        //Formatting
        DateTimeFormatter dtF2 = DateTimeFormatter.ofPattern("EEE d, MMMM, yyyy HH:mm");

        DateTimeFormatter dtF3 = DateTimeFormatter.ISO_LOCAL_DATE_TIME;

        LocalDateTime ldtf1 = LocalDateTime.now();

        System.out.println(ldtf1.format(dtF2) +"\n"+ldtf1.format(dtF3));
    }
}
```

An important notice, instead of using Custom patterns, it is good practice to use predefined formatters. Your code look more clear and usage of ISO8061 will definitely help you in the long run.

Section 17.7: Simple Date Manipulations

Get the current date.

```
LocalDate.now()
```

Get yesterday's date.

```
LocalDate y = LocalDate.now().minusDays(1);
```

Get tomorrow's date

```
LocalDate t = LocalDate.now().plusDays(1);
```

Get a specific date.

```
LocalDate t = LocalDate.of(1974, 6, 2, 8, 30, 0, 0);
```

In addition to the `plus` and `minus` methods, there are a set of "with" methods that can be used to set a particular field on a `LocalDate` instance.

```
LocalDate.now().withMonth(6);
```

The example above returns a new instance with the month set to June (this differs from `java.util.Date` where `setMonth` was indexed a 0 making June 5).

Because LocalDate manipulations return immutable LocalDate instances, these methods may also be chained together.

```
LocalDate ld = LocalDate.now().plusDays(1).plusYears(1);
```

This would give us tomorrow's date one year from now.

Chapter 18: LocalTime

Method	Output
`LocalTime.of(13, 12, 11)`	13:12:11
`LocalTime.MIDNIGHT`	00:00
`LocalTime.NOON`	12:00
`LocalTime.now()`	Current time from system clock
`LocalTime.MAX`	The maximum supported local time 23:59:59.999999999
`LocalTime.MIN`	The minimum supported local time 00:00
`LocalTime.ofSecondOfDay(84399)`	23:59:59 , Obtains Time from second-of-day value
`LocalTime.ofNanoOfDay(2000000000)`	00:00:02 , Obtains Time from nanos-of-day value

Section 18.1: Amount of time between two LocalTime

There are two equivalent ways to calculate the amount of time unit between two `LocalTime`: (1) through `until(Temporal, TemporalUnit)` method and through (2) `TemporalUnit.between(Temporal, Temporal)`.

```
import java.time.LocalTime;
import java.time.temporal.ChronoUnit;

public class AmountOfTime {

    public static void main(String[] args) {

        LocalTime start = LocalTime.of(1, 0, 0); // hour, minute, second
        LocalTime end = LocalTime.of(2, 10, 20); // hour, minute, second

        long halfDays1 = start.until(end, ChronoUnit.HALF_DAYS); // 0
        long halfDays2 = ChronoUnit.HALF_DAYS.between(start, end); // 0

        long hours1 = start.until(end, ChronoUnit.HOURS); // 1
        long hours2 = ChronoUnit.HOURS.between(start, end); // 1

        long minutes1 = start.until(end, ChronoUnit.MINUTES); // 70
        long minutes2 = ChronoUnit.MINUTES.between(start, end); // 70

        long seconds1 = start.until(end, ChronoUnit.SECONDS); // 4220
        long seconds2 = ChronoUnit.SECONDS.between(start, end); // 4220

        long millisecs1 = start.until(end, ChronoUnit.MILLIS); // 4220000
        long millisecs2 = ChronoUnit.MILLIS.between(start, end); // 4220000

        long microsecs1 = start.until(end, ChronoUnit.MICROS); // 4220000000
        long microsecs2 = ChronoUnit.MICROS.between(start, end); // 4220000000

        long nanosecs1 = start.until(end, ChronoUnit.NANOS); // 4220000000000
        long nanosecs2 = ChronoUnit.NANOS.between(start, end); // 4220000000000

        // Using others ChronoUnit will be thrown UnsupportedTemporalTypeException.
        // The following methods are examples thereof.
        long days1 = start.until(end, ChronoUnit.DAYS);
        long days2 = ChronoUnit.DAYS.between(start, end);
    }
}
```

Section 18.2: Intro

LocalTime is an immutable class and thread-safe, used to represent time, often viewed as hour-min-sec. Time is represented to nanosecond precision. For example, the value "13:45.30.123456789" can be stored in a LocalTime.

This class does not store or represent a date or time-zone. Instead, it is a description of the local time as seen on a wall clock. It cannot represent an instant on the time-line without additional information such as an offset or time-zone. This is a value based class, equals method should be used for comparisons.

Fields

MAX - The maximum supported LocalTime, '23:59:59.999999999'. MIDNIGHT, MIN, NOON

Important Static Methods

now(), now(Clock clock), now(ZoneId zone), parse(CharSequence text)

Important Instance Methods

isAfter(LocalTime other), isBefore(LocalTime other), minus(TemporalAmount amountToSubtract), minus(long amountToSubtract, TemporalUnit unit), plus(TemporalAmount amountToAdd), plus(long amountToAdd, TemporalUnit unit)

```
ZoneId zone = ZoneId.of("Asia/Kolkata");
LocalTime now = LocalTime.now();
LocalTime now1 = LocalTime.now(zone);
LocalTime then = LocalTime.parse("04:16:40");
```

Difference in time can be calculated in any of following ways

```
long timeDiff = Duration.between(now, now1).toMinutes();
long timeDiff1 = java.time.temporal.ChronoUnit.MINUTES.between(now2, now1);
```

You can also add/subtract hours, minutes or seconds from any object of LocalTime.

minusHours(long hoursToSubtract), minusMinutes(long hoursToMinutes), minusNanos(long nanosToSubtract), minusSeconds(long secondsToSubtract), plusHours(long hoursToSubtract), plusMinutes(long hoursToMinutes), plusNanos(long nanosToSubtract), plusSeconds(long secondsToSubtract)

```
now.plusHours(1L);
now1.minusMinutes(20L);
```

Section 18.3: Time Modification

You can add hours, minutes, seconds and nanoseconds:

```
LocalTime time = LocalTime.now();
LocalTime addHours = time.plusHours(5); // Add 5 hours
LocaLTime addMinutes = time.plusMinutes(15) // Add 15 minutes
LocalTime addSeconds = time.plusSeconds(30) // Add 30 seconds
LocalTime addNanoseconds = time.plusNanos(150_000_000) // Add 150.000.000ns (150ms)
```

Section 18.4: Time Zones and their time difference

```
import java.time.LocalTime;
```

```java
import java.time.ZoneId;
import java.time.temporal.ChronoUnit;

public class Test {
    public static void main(String[] args)
    {
            ZoneId zone1 = ZoneId.of("Europe/Berlin");
            ZoneId zone2 = ZoneId.of("Brazil/East");

            LocalTime now = LocalTime.now();
            LocalTime now1 = LocalTime.now(zone1);
            LocalTime now2 = LocalTime.now(zone2);

            System.out.println("Current Time : " + now);
            System.out.println("Berlin Time : " + now1);
            System.out.println("Brazil Time : " + now2);

            long minutesBetween = ChronoUnit.MINUTES.between(now2, now1);
            System.out.println("Minutes Between Berlin and Brazil : " + minutesBetween +"mins");

    }
}
```

Chapter 19: BigDecimal

The BigDecimal class provides operations for arithmetic (add, subtract, multiply, divide), scale manipulation, rounding, comparison, hashing, and format conversion. The BigDecimal represents immutable, arbitrary-precision signed decimal numbers. This class shall be used in a necessity of high-precision calculation.

Section 19.1: Comparing BigDecimals

The method `compareTo` should be used to compare `BigDecimals`:

```
BigDecimal a = new BigDecimal(5);
a.compareTo(new BigDecimal(0));    // a is greater, returns 1
a.compareTo(new BigDecimal(5));    // a is equal, returns 0
a.compareTo(new BigDecimal(10));   // a is less, returns -1
```

Commonly you should **not** use the `equals` method since it considers two `BigDecimals` equal only if they are equal in value and also **scale**:

```
BigDecimal a = new BigDecimal(5);
a.equals(new BigDecimal(5));       // value and scale are equal, returns true
a.equals(new BigDecimal(5.00));    // value is equal but scale is not, returns false
```

Section 19.2: Using BigDecimal instead of float

Due to way that the float type is represented in computer memory, results of operations using this type can be inaccurate - some values are stored as approximations. Good examples of this are monetary calculations. If high precision is necessary, other types should be used. e.g. Java 7 provides BigDecimal.

```
import java.math.BigDecimal;

public class FloatTest {

public static void main(String[] args) {
    float accountBalance = 10000.00f;
    System.out.println("Operations using float:");
    System.out.println("1000 operations for 1.99");
    for(int i = 0; i<1000; i++){
        accountBalance -= 1.99f;
    }
    System.out.println(String.format("Account balance after float operations: %f",
accountBalance));

    BigDecimal accountBalanceTwo = new BigDecimal("10000.00");
    System.out.println("Operations using BigDecimal:");
    System.out.println("1000 operations for 1.99");
    BigDecimal operation = new BigDecimal("1.99");
    for(int i = 0; i<1000; i++){
        accountBalanceTwo = accountBalanceTwo.subtract(operation);
    }
    System.out.println(String.format("Account balance after BigDecimal operations: %f",
accountBalanceTwo));
}
```

Output of this program is:

```
Operations using float:
1000 operations for 1.99
Account balance after float operations: 8009,765625
Operations using BigDecimal:
1000 operations for 1.99
Account balance after BigDecimal operations: 8010,000000
```

For a starting balance of 10000.00, after 1000 operations for 1.99, we expect the balance to be 8010.00. Using the float type gives us an answer around 8009.77, which is unacceptably imprecise in the case of monetary calculations. Using BigDecimal gives us the proper result.

Section 19.3: BigDecimal.valueOf()

The BigDecimal class contains an internal cache of frequently used numbers e.g. 0 to 10. The BigDecimal.valueOf() methods are provided in preference to constructors with similar type parameters i.e. in the below example a is preferred to b.

```
BigDecimal a = BigDecimal.valueOf(10L); //Returns cached Object reference
BigDecimal b = new BigDecimal(10L); //Does not return cached Object reference

BigDecimal a = BigDecimal.valueOf(20L); //Does not return cached Object reference
BigDecimal b = new BigDecimal(20L); //Does not return cached Object reference


BigDecimal a = BigDecimal.valueOf(15.15); //Preferred way to convert a double (or float) into a
BigDecimal, as the value returned is equal to that resulting from constructing a BigDecimal from the
result of using Double.toString(double)
BigDecimal b = new BigDecimal(15.15); //Return unpredictable result
```

Section 19.4: Mathematical operations with BigDecimal

This example shows how to perform basic mathematical operations using BigDecimals.

1.Addition

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

//Equivalent to result = a + b
BigDecimal result = a.add(b);
System.out.println(result);
```

> Result : 12

2.Subtraction

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

//Equivalent to result = a - b
BigDecimal result = a.subtract(b);
System.out.println(result);
```

> Result : -2

3.Multiplication

When multiplying two `BigDecimal`s the result is going to have scale equal to the sum of the scales of operands.

```
BigDecimal a = new BigDecimal("5.11");
BigDecimal b = new BigDecimal("7.221");

//Equivalent to result = a * b
BigDecimal result = a.multiply(b);
System.out.println(result);
```

> Result : 36.89931

To change the scale of the result use the overloaded multiply method which allows passing `MathContext` - an object describing the rules for operators, in particular the precision and rounding mode of the result. For more information about available rounding modes please refer to the Oracle Documentation.

```
BigDecimal a = new BigDecimal("5.11");
BigDecimal b = new BigDecimal("7.221");

MathContext returnRules = new MathContext(4, RoundingMode.HALF_DOWN);

//Equivalent to result = a * b
BigDecimal result = a.multiply(b, returnRules);
System.out.println(result);
```

> Result : 36.90

4.Division

Division is a bit more complicated than the other arithmetic operations, for instance consider the below example:

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

BigDecimal result = a.divide(b);
System.out.println(result);
```

We would expect this to give something similar to : 0.7142857142857143, but we would get:

> Result: **java.lang.ArithmeticException: Non-terminating decimal expansion; no exact representable decimal result.**

This would work perfectly well when the result would be a terminating decimal say if I wanted to divide 5 by 2, but for those numbers which upon dividing would give a non terminating result we would get an `ArithmeticException`. In the real world scenario, one cannot predict the values that would be encountered during the division, so we need to specify the **Scale** and the **Rounding Mode** for BigDecimal division. For more information on the Scale and Rounding Mode, refer the Oracle Documentation.

For example, I could do:

```
BigDecimal a = new BigDecimal("5");
```

```
BigDecimal b = new BigDecimal("7");

//Equivalent to result = a / b (Upto 10 Decimal places and Round HALF_UP)
BigDecimal result = a.divide(b,10,RoundingMode.HALF_UP);
System.out.println(result);
```

> Result : 0.7142857143

5.Remainder or Modulus

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

//Equivalent to result = a % b
BigDecimal result = a.remainder(b);
System.out.println(result);
```

> Result : 5

6.Power

```
BigDecimal a = new BigDecimal("5");

//Equivalent to result = a^10
BigDecimal result = a.pow(10);
System.out.println(result);
```

> Result : 9765625

7.Max

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

//Equivalent to result = MAX(a,b)
BigDecimal result = a.max(b);
System.out.println(result);
```

> Result : 7

8.Min

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

//Equivalent to result = MIN(a,b)
BigDecimal result = a.min(b);
System.out.println(result);
```

> Result : 5

9.Move Point To Left

```
BigDecimal a = new BigDecimal("5234.49843776");

//Moves the decimal point to 2 places left of current position
BigDecimal result = a.movePointLeft(2);
System.out.println(result);
```

> Result : 52.3449843776

10.Move Point To Right

```
BigDecimal a = new BigDecimal("5234.49843776");

//Moves the decimal point to 3 places right of current position
BigDecimal result = a.movePointRight(3);
System.out.println(result);
```

> Result : 5234498.43776

There are many more options and combination of parameters for the above mentioned examples (For instance, there are 6 variations of the divide method), this set is a non-exhaustive list and covers a few basic examples.

Section 19.5: Initialization of BigDecimals with value zero, one or ten

BigDecimal provides static properties for the numbers zero, one and ten. It's good practise to use these instead of using the actual numbers:

- `BigDecimal.ZERO`
- `BigDecimal.ONE`
- `BigDecimal.TEN`

By using the static properties, you avoid an unnecessary instantiation, also you've got a literal in your code instead of a 'magic number'.

```
//Bad example:
BigDecimal bad0 = new BigDecimal(0);
BigDecimal bad1 = new BigDecimal(1);
BigDecimal bad10 = new BigDecimal(10);

//Good Example:
BigDecimal good0 = BigDecimal.ZERO;
BigDecimal good1 = BigDecimal.ONE;
BigDecimal good10 = BigDecimal.TEN;
```

Section 19.6: BigDecimal objects are immutable

If you want to calculate with BigDecimal you have to use the returned value because BigDecimal objects are immutable:

```
BigDecimal a = new BigDecimal("42.23");
BigDecimal b = new BigDecimal("10.001");

a.add(b); // a will still be 42.23
```

```
BigDecimal c = a.add(b); // c will be 52.231
```

Chapter 20: BigInteger

The `BigInteger` class is used for mathematical operations involving large integers with magnitudes too large for primitive data types. For example 100-factorial is 158 digits - much larger than a **`long`** can represent. `BigInteger` provides analogues to all of Java's primitive integer operators, and all relevant methods from `java.lang.Math` as well as few other operations.

Section 20.1: Initialization

The `java.math.BigInteger` class provides operations analogues to all of Java's primitive integer operators and for all relevant methods from `java.lang.Math`. As the `java.math` package is not automatically made available you may have to import `java.math.BigInteger` before you can use the simple class name.

To convert **`long`** or **`int`** values to `BigInteger` use:

```
long longValue = Long.MAX_VALUE;
BigInteger valueFromLong = BigInteger.valueOf(longValue);
```

or, for integers:

```
int intValue = Integer.MIN_VALUE; // negative
BigInteger valueFromInt = BigInteger.valueOf(intValue);
```

which will *widen* the `intValue` integer to long, using sign bit extension for negative values, so that negative values will stay negative.

To convert a numeric `String` to `BigInteger` use:

```
String decimalString = "-1";
BigInteger valueFromDecimalString = new BigInteger(decimalString);
```

Following constructor is used to translate the String representation of a `BigInteger` in the specified radix into a `BigInteger`.

```
String binaryString = "10";
int binaryRadix = 2;
BigInteger valueFromBinaryString = new BigInteger(binaryString , binaryRadix);
```

Java also supports direct conversion of bytes to an instance of `BigInteger`. Currently only signed and unsigned big endian encoding may be used:

```
byte[] bytes = new byte[] { (byte) 0x80 };
BigInteger valueFromBytes = new BigInteger(bytes);
```

This will generate a `BigInteger` instance with value -128 as the first bit is interpreted as the sign bit.

```
byte[] unsignedBytes = new byte[] { (byte) 0x80 };
int sign = 1; // positive
BigInteger valueFromUnsignedBytes = new BigInteger(sign, unsignedBytes);
```

This will generate a `BigInteger` instance with value 128 as the bytes are interpreted as unsigned number, and the sign is explicitly set to 1, a positive number.

There are predefined constants for common values:

- `BigInteger.ZERO` — value of "0".
- `BigInteger.ONE` — value of "1".
- `BigInteger.TEN` — value of "10".

There's also `BigInteger.TWO` (value of "2"), but you can't use it in your code because it's **`private`**.

Section 20.2: BigInteger Mathematical Operations Examples

BigInteger is in an immutable object, so you need to assign the results of any mathematical operation, to a new BigInteger instance.

Addition: *10 + 10 = 20*

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("10");

BigInteger sum = value1.add(value2);
System.out.println(sum);
```

> output: 20

Subtraction: *10 - 9 = 1*

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("9");

BigInteger sub = value1.subtract(value2);
System.out.println(sub);
```

> output: 1

Division: *10 / 5 = 2*

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("5");

BigInteger div = value1.divide(value2);
System.out.println(div);
```

> output: 2

Division: *17/4 = 4*

```
BigInteger value1 = new BigInteger("17");
BigInteger value2 = new BigInteger("4");

BigInteger div = value1.divide(value2);
System.out.println(div);
```

> output: 4

Multiplication: *10 * 5 = 50*

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("5");

BigInteger mul = value1.multiply(value2);
System.out.println(mul);
```

> output: 50

Power: *10 ^ 3 = 1000*

```
BigInteger value1 = new BigInteger("10");
BigInteger power = value1.pow(3);
System.out.println(power);
```

> output: 1000

Remainder: *10 % 6 = 4*

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("6");

BigInteger power = value1.remainder(value2);
System.out.println(power);
```

> output: 4

GCD: Greatest Common Divisor (GCD) for `12`and `18` is `6`.

```
BigInteger value1 = new BigInteger("12");
BigInteger value2 = new BigInteger("18");

System.out.println(value1.gcd(value2));
```

> Output: 6

Maximum of two BigIntegers:

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("11");

System.out.println(value1.max(value2));
```

> Output: 11

Minimum of two BigIntegers:

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("11");

System.out.println(value1.min(value2));
```

> Output: 10

Section 20.3: Comparing BigIntegers

You can compare `BigIntegers` same as you compare `String` or other objects in Java.

For example:

```
BigInteger one = BigInteger.valueOf(1);
BigInteger two = BigInteger.valueOf(2);

if(one.equals(two)){
    System.out.println("Equal");
}
else{
    System.out.println("Not Equal");
}
```

Output:

```
Not Equal
```

Note:

In general, do **not** use use the `==` operator to compare BigIntegers

- `==` operator: compares references; i.e. whether two values refer to the same object
- `equals()` method: compares the content of two BigIntegers.

For example, BigIntegers should **not** be compared in the following way:

```
if (firstBigInteger == secondBigInteger) {
  // Only checks for reference equality, not content equality!
}
```

Doing so may lead to unexpected behavior, as the `==` operator only checks for reference equality. If both BigIntegers contain the same content, but do not refer to the same object, **this will fail.** Instead, compare BigIntegers using the `equals` methods, as explained above.

You can also compare your `BigInteger` to constant values like 0,1,10.

for example:

```
BigInteger reallyBig = BigInteger.valueOf(1);
if(BigInteger.ONE.equals(reallyBig)){
    //code when they are equal.
}
```

You can also compare two BigIntegers by using `compareTo()` method, as following: `compareTo()` returns 3 values.

- **0:** When both are **equal**.
- **1:** When first is **greater than** second (the one in brackets).
- **-1:** When first is **less than** second.

```
BigInteger reallyBig = BigInteger.valueOf(10);
BigInteger reallyBig1 = BigInteger.valueOf(100);

if(reallyBig.compareTo(reallyBig1) == 0){
    //code when both are equal.
}
else if(reallyBig.compareTo(reallyBig1) == 1){
    //code when reallyBig is greater than reallyBig1.
}
else if(reallyBig.compareTo(reallyBig1) == -1){
    //code when reallyBig is less than reallyBig1.
}
```

Section 20.4: Binary Logic Operations on BigInteger

BigInteger supports the binary logic operations that are available to `Number` types as well. As with all operations they are implemented by calling a method.

Binary Or:

```
BigInteger val1 = new BigInteger("10");
BigInteger val2 = new BigInteger("9");

val1.or(val2);
```

> Output: 11 (which is equivalent to `10 | 9`)

Binary And:

```
BigInteger val1 = new BigInteger("10");
BigInteger val2 = new BigInteger("9");

val1.and(val2);
```

> Output: 8 (which is equivalent to `10 & 9`)

Binary Xor:

```
BigInteger val1 = new BigInteger("10");
BigInteger val2 = new BigInteger("9");

val1.xor(val2);
```

> Output: 3 (which is equivalent to `10 ^ 9`)

RightShift:

```
BigInteger val1 = new BigInteger("10");
```

```
val1.shiftRight(1);   // the argument be an Integer
```

> Output: 5 (equivalent to `10 >> 1`)

LeftShift:

```
BigInteger val1 = new BigInteger("10");

val1.shiftLeft(1);   // here parameter should be Integer
```

> Output: 20 (equivalent to `10 << 1`)

Binary Inversion (Not):

```
BigInteger val1 = new BigInteger("10");

val1.not();
```

> Output: 5

*NAND (And-Not):**

```
BigInteger val1 = new BigInteger("10");
BigInteger val2 = new BigInteger("9");

val1.andNot(val2);
```

> Output: 7

Section 20.5: Generating random BigIntegers

The `BigInteger` class has a constructor dedicated to generate random `BigIntegers`, given an instance of `java.util.Random` and an **`int`** that specifies how many bits will the `BigInteger` have. Its usage is quite simple - when you call the constructor `BigInteger(int, Random)` like this:

```
BigInteger randomBigInt = new BigInteger(bitCount, sourceOfRandomness);
```

then you'll end up with a `BigInteger` whose value is between 0 (inclusive) and 2`bitCount` (exclusive).

This also means that **`new`** `BigInteger(2147483647, sourceOfRandomness)` may return all positive `BigInteger`s given enough time.

What will the `sourceOfRandomness` be is up to you. For example, a **`new`** `Random()` is good enough in most cases:

```
new BigInteger(32, new Random());
```

If you're willing to give up speed for higher-quality random numbers, you can use a **new** ="https://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html" rel="nofollow noreferrer">SecureRandom() instead:

```java
import java.security.SecureRandom;

// somewhere in the code...
new BigInteger(32, new SecureRandom());
```

You can even implement an algorithm on-the-fly with an anonymous class! Note that **rolling out your own RNG algorithm *will* end you up with low quality randomness**, so always be sure to use an algorithm that is proven to be decent unless you want the resulting `BigInteger`(s) to be predictable.

```java
new BigInteger(32, new Random() {
    int seed = 0;

    @Override
    protected int next(int bits) {
        seed = ((22695477 * seed) + 1) & 2147483647; // Values shamelessly stolen from
```

="https://en.wikipedia.org/wiki/Linear_congruential_generator#Parameters_in_common_use" rel="nofollow noreferrer">Wikipedia return seed; } });

Chapter 21: NumberFormat

Section 21.1: NumberFormat

Different countries have different number formats and considering this we can have different formats using Locale of java. Using locale can help in formatting

```
Locale locale = new Locale("en", "IN");
NumberFormat numberFormat = NumberFormat.getInstance(locale);
```

using above format you can perform various tasks

1. Format Number

   ```
   numberFormat.format(10000000.99);
   ```

2. Format Currency

   ```
   NumberFormat currencyFormat = NumberFormat.getCurrencyInstance(locale);
   currencyFormat.format(10340.999);
   ```

3. Format Percentage

   ```
   NumberFormat percentageFormat = NumberFormat.getPercentInstance(locale);
   percentageFormat.format(10929.999);
   ```

4. Control Number of Digits

```
numberFormat.setMinimumIntegerDigits(int digits)
numberFormat.setMaximumIntegerDigits(int digits)
numberFormat.setMinimumFractionDigits(int digits)
numberFormat.setMaximumFractionDigits(int digits)
```

Chapter 22: Bit Manipulation

Section 22.1: Checking, setting, clearing, and toggling individual bits. Using long as bit mask

Assuming we want to modify bit `n` of an integer primitive, `i` (byte, short, char, int, or long):

```
(i & 1 << n) != 0 // checks bit 'n'
i |= 1 << n;      // sets bit 'n' to 1
i &= ~(1 << n);   // sets bit 'n' to 0
i ^= 1 << n;      // toggles the value of bit 'n'
```

Using long/int/short/byte as a bit mask:

```
public class BitMaskExample {
    private static final long FIRST_BIT = 1L << 0;
    private static final long SECOND_BIT = 1L << 1;
    private static final long THIRD_BIT = 1L << 2;
    private static final long FOURTH_BIT = 1L << 3;
    private static final long FIFTH_BIT = 1L << 4;
    private static final long BIT_55 = 1L << 54;

    public static void main(String[] args) {
        checkBitMask(FIRST_BIT | THIRD_BIT | FIFTH_BIT | BIT_55);
    }

    private static void checkBitMask(long bitmask) {
        System.out.println("FIRST_BIT: " + ((bitmask & FIRST_BIT) != 0));
        System.out.println("SECOND_BIT: " + ((bitmask & SECOND_BIT) != 0));
        System.out.println("THIRD_BIT: " + ((bitmask & THIRD_BIT) != 0));
        System.out.println("FOURTh_BIT: " + ((bitmask & FOURTH_BIT) != 0));
        System.out.println("FIFTH_BIT: " + ((bitmask & FIFTH_BIT) != 0));
        System.out.println("BIT_55: " + ((bitmask & BIT_55) != 0));
    }
}
```

Prints

```
FIRST_BIT: true
SECOND_BIT: false
THIRD_BIT: true
FOURTh_BIT: false
FIFTH_BIT: true
BIT_55: true
```

which matches that mask we passed as `checkBitMask` parameter: `FIRST_BIT | THIRD_BIT | FIFTH_BIT | BIT_55`.

Section 22.2: java.util.BitSet class

Since 1.7 there's a java.util.BitSet class that provides simple and user-friendly bit storage and manipulation interface:

```
final BitSet bitSet = new BitSet(8); // by default all bits are unset

IntStream.range(0, 8).filter(i -> i % 2 == 0).forEach(bitSet::set); // {0, 2, 4, 6}

bitSet.set(3); // {0, 2, 3, 4, 6}
```

```
bitSet.set(3, false); // {0, 2, 4, 6}

final boolean b = bitSet.get(3); // b = false

bitSet.flip(6); // {0, 2, 4}

bitSet.set(100); // {0, 2, 4, 100} - expands automatically
```

`BitSet` implements `Clonable` and `Serializable`, and under the hood all bit values are stored in `long[] words` field, that expands automatically.

It also supports whole-set logical operations `and`, `or`, `xor`, `andNot`:

```
bitSet.and(new BitSet(8));
bitSet.or(new BitSet(8));
bitSet.xor(new BitSet(8));
bitSet.andNot(new BitSet(8));
```

Section 22.3: Checking if a number is a power of 2

If an integer `x` is a power of 2, only one bit is set, whereas `x-1` has all bits set after that. For example: `4` is `100` and `3` is `011` as binary number, which satisfies the aforementioned condition. Zero is not a power of 2 and has to be checked explicitly.

```
boolean isPowerOfTwo(int x)
{
    return (x != 0) && ((x & (x - 1)) == 0);
}
```

Usage for Left and Right Shift

Let's suppose, we have three kind of permissions, **READ**, **WRITE** and **EXECUTE**. Each permission can range from 0 to 7. (Let's assume 4 bit number system)

> RESOURCE = READ WRITE EXECUTE (12 bit number)
>
> RESOURCE = 0100 0110 0101 = 4 6 5 (12 bit number)

How can we get the (12 bit number) permissions, set on above (12 bit number)?

> 0100 0110 0101
>
> 0000 0000 0111 (&)
>
> 0000 0000 0101 = 5

So, this is how we can get the **EXECUTE** permissions of the **RESOURCE**. Now, what if we want to get **READ** permissions of the **RESOURCE**?

> 0100 0110 0101
>
> 0111 0000 0000 (&)

> 0100 0000 0000 = 1024

Right? You are probably assuming this? But, permissions are resulted in 1024. We want to get only READ permissions for the resource. Don't worry, that's why we had the shift operators. If we see, READ permissions are 8 bits behind the actual result, so if apply some shift operator, which will bring READ permissions to the very right of the result? What if we do:

> 0100 0000 0000 >> 8 => 0000 0000 0100 (Because it's a positive number so replaced with 0's, if you don't care about sign, just use unsigned right shift operator)

We now actually have the **READ** permissions which is 4.

Now, for example, we are given **READ**, **WRITE**, **EXECUTE** permissions for a **RESOURCE**, what can we do to make permissions for this **RESOURCE**?

Let's first take the example of binary permissions. (Still assuming 4 bit number system)

> READ = 0001
>
> WRITE = 0100
>
> EXECUTE = 0110

If you are thinking that we will simply do:

`READ | WRITE | EXECUTE`, you are somewhat right but not exactly. See, what will happen if we will perform READ | WRITE | EXECUTE

0001 | 0100 | 0110 => 0111

But permissions are actually being represented (in our example) as 0001 0100 0110

So, in order to do this, we know that **READ** is placed 8 bits behind, **WRITE** is placed 4 bits behind and **PERMISSIONS** is placed at the last. The number system being used for **RESOURCE** permissions is actually 12 bit (in our example). It can(will) be different in different systems.

> (READ << 8) | (WRITE << 4) | (EXECUTE)
>
> 0000 0000 0001 << 8 (READ)
>
> 0001 0000 0000 (Left shift by 8 bits)
>
> 0000 0000 0100 << 4 (WRITE)
>
> 0000 0100 0000 (Left shift by 4 bits)
>
> 0000 0000 0001 (EXECUTE)

Now if we add the results of above shifting, it will be something like;

> 0001 0000 0000 (READ)
>
> 0000 0100 0000 (WRITE)
>
> 0000 0000 0001 (EXECUTE)
>
> 0001 0100 0001 (PERMISSIONS)

Section 22.4: Signed vs unsigned shift

In Java, all number primitives are signed. For example, an int always represent values from [-2^31 - 1, 2^31], keeping the first bit to sign the value - 1 for negative value, 0 for positive.

Basic shift operators `>>` and `<<` are signed operators. They will conserve the sign of the value.

But it is common for programmers to use numbers to store *unsigned values*. For an int, it means shifting the range to [0, 2^32 - 1], to have twice as much value as with a signed int.

For those power users, the bit for sign as no meaning. That's why Java added `>>>`, a left-shift operator, disregarding that sign bit.

```
               initial value:              4 (                                100)
    signed left-shift: 4 << 1              8 (                               1000)
   signed right-shift: 4 >> 1              2 (                                 10)
unsigned right-shift: 4 >>> 1              2 (                                 10)
               initial value:             -4 (  11111111111111111111111111111100)
   signed left-shift: -4 << 1             -8 (  11111111111111111111111111111000)
  signed right-shift: -4 >> 1             -2 (  11111111111111111111111111111110)
unsigned right-shift: -4 >>> 1     2147483646 (   1111111111111111111111111111110)
```

Why is there no `<<<` ?

This comes from the intended definition of right-shift. As it fills the emptied places on the left, there are no decision to take regarding the bit of sign. As a consequence, there is no need for 2 different operators.

See this question for a more detailled answer.

Section 22.5: Expressing the power of 2

For expressing the power of 2 (2^n) of integers, one may use a bitshift operation that allows to explicitly specify the n.

The syntax is basically:

```
int pow2 = 1<<n;
```

Examples:

```
int twoExp4 = 1<<4; //2^4
int twoExp5 = 1<<5; //2^5
int twoExp6 = 1<<6; //2^6
...
int twoExp31 = 1<<31; //2^31
```

This is especially useful when defining constant values that should make it apparent, that a power of 2 is used,

instead of using hexadecimal or decimal values.

```
int twoExp4 = 0x10; //hexadecimal
int twoExp5 = 0x20; //hexadecimal
int twoExp6 = 64; //decimal
...
int twoExp31 = -2147483648; //is that a power of 2?
```

A simple method to calculate the int power of 2 would be

```
int pow2(int exp){
    return 1<<exp;
}
```

Section 22.6: Packing / unpacking values as bit fragments

It is common for memory performance to compress multiple values into a single primitive value. This may be useful to pass various information into a single variable.

For example, one can pack 3 bytes - such as color code in RGB - into an single int.

Packing the values

```
// Raw bytes as input
byte[] b = {(byte)0x65, (byte)0xFF, (byte)0x31};

// Packed in big endian: x == 0x65FF31
int x = (b[0] & 0xFF) << 16  // Red
      | (b[1] & 0xFF) <<  8  // Green
      | (b[2] & 0xFF) <<  0; // Blue

// Packed in little endian: y == 0x31FF65
int y = (b[0] & 0xFF) <<  0
      | (b[1] & 0xFF) <<  8
      | (b[2] & 0xFF) << 16;
```

Unpacking the values

```
// Raw int32 as input
int x = 0x31FF65;

// Unpacked in big endian: {0x65, 0xFF, 0x31}
byte[] c = {
    (byte)(x >> 16),
    (byte)(x >>  8),
    (byte)(x & 0xFF)
};

// Unpacked in little endian: {0x31, 0xFF, 0x65}
byte[] d = {
    (byte)(x & 0xFF),
    (byte)(x >>  8),
    (byte)(x >> 16)
};
```

Chapter 23: Arrays

Parameter	Details
`ArrayType`	Type of the array. This can be primitive (**`int`**, **`long`**, **`byte`**) or Objects (`String`, `MyObject`, etc).
index	Index refers to the position of a certain Object in an array.
length	Every array, when being created, needs a set length specified. This is either done when creating an empty array (**`new int`**`[3]`) or implied when specifying values (`{1, 2, 3}`).

Arrays allow for the storage and retrieval of an arbitrary quantity of values. They are analogous to vectors in mathematics. Arrays of arrays are analogous to matrices, and act as multidimensional arrays. Arrays can store any data of any type: primitives such as **`int`** or reference types such as `Object`.

Section 23.1: Creating and Initializing Arrays

Basic cases

```
int[]   numbers1 = new int[3];                // Array for 3 int values, default value is 0
int[]   numbers2 = { 1, 2, 3 };               // Array literal of 3 int values
int[]   numbers3 = new int[] { 1, 2, 3 };     // Array of 3 int values initialized
int[][] numbers4 = { { 1, 2 }, { 3, 4, 5 } }; // Jagged array literal
int[][] numbers5 = new int[5][];              // Jagged array, one dimension 5 long
int[][] numbers6 = new int[5][4];             // Multidimensional array: 5x4
```

Arrays may be created using any primitive or reference type.

```
float[]  boats = new float[5];          // Array of five 32-bit floating point numbers.
double[] header = new double[] { 4.56, 332.267, 7.0, 0.3367, 10.0 };
                                        // Array of five 64-bit floating point numbers.
String[] theory = new String[] { "a", "b", "c" };
                                        // Array of three strings (reference type).
Object[] dArt = new Object[] { new Object(), "We love Stack Overflow.", new Integer(3) };
                                        // Array of three Objects (reference type).
```

For the last example, note that subtypes of the declared array type are allowed in the array.

Arrays for user defined types can also be built similar to primitive types

```
UserDefinedClass[] udType = new UserDefinedClass[5];
```

Arrays, Collections, and Streams

Version ≥ Java SE 1.2

```
// Parameters require objects, not primitives

// Auto-boxing happening for int 127 here
Integer[]       initial        = { 127, Integer.valueOf( 42 ) };
List<Integer>   toList         = Arrays.asList( initial );  // Fixed size!

// Note: Works with all collections
Integer[]       fromCollection = toList.toArray( new Integer[toList.size()] );

//Java doesn't allow you to create an array of a parameterized type
List<String>[]  list = new ArrayList<String>[2];  // Compilation error!
```

Version ≥ Java SE 8

```
// Streams - JDK 8+
Stream<Integer> toStream       = Arrays.stream( initial );
Integer[]       fromStream     = toStream.toArray( Integer[]::new );
```

Intro

An *array* is a data structure that holds a fixed number of primitive values **or** references to object instances.

Each item in an array is called an element, and each element is accessed by its numerical index. The length of an array is established when the array is created:

```
int size = 42;
int[] array = new int[size];
```

The size of an array is fixed at runtime when initialized. It cannot be changed after initialization. If the size must be mutable at runtime, a `Collection` class such as `ArrayList` should be used instead. `ArrayList` stores elements in an array and supports resizing by allocating a new array and copying elements from the old array.

If the array is of a primitive type, i.e.

```
int[] array1 = { 1,2,3 };
int[] array2 = new int[10];
```

the values are stored in the array itself. In the absence of an initializer (as in `array2` above), the default value assigned to each element is `0` (zero).

If the array type is an object reference, as in

```
SomeClassOrInterface[] array = new SomeClassOrInterface[10];
```

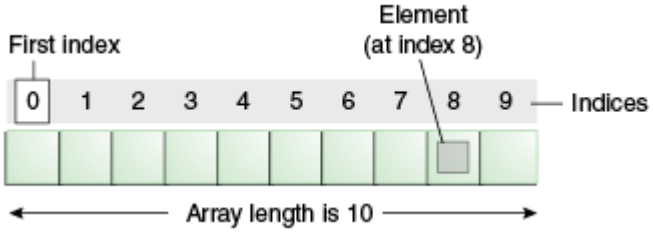
then the array contains *references* to objects of type `SomeClassOrInterface`. Those references can refer to an instance of `SomeClassOrInterface` *or any subclass (for classes) or implementing class (for interfaces) of* `SomeClassOrInterface`. If the array declaration has no initializer then the default value of **`null`** is assigned to each element.

Because all arrays are **`int`**-indexed, the size of an array must be specified by an **`int`**. The size of the array cannot be specified as a **`long`**:

```
long size = 23L;
int[] array = new int[size]; // Compile-time error:
                             // incompatible types: possible lossy conversion from
                             // long to int
```

Arrays use a **zero-based index** system, which means indexing starts at `0` and ends at `length - 1`.

For example, the following image represents an array with size `10`. Here, the first element is at index `0` and the last element is at index `9`, instead of the first element being at index `1` and the last element at index `10` (see figure below).



Accesses to elements of arrays are done in **constant time**. That means accessing to the first element of the array has the same cost (in time) of accessing the second element, the third element and so on.

Java offers several ways of defining and initializing arrays, including **literal** and **constructor** notations. When

declaring arrays using the `new Type[length]` constructor, each element will be initialized with the following default values:

- `0` for primitive numerical types: **`byte`**, **`short`**, **`int`**, **`long`**, **`float`**, and **`double`**.
- `'\u0000'` (null character) for the **`char`** type.
- **`false`** for the **`boolean`** type.
- **`null`** for reference types.

Creating and initializing primitive type arrays

```
int[] array1 = new int[] { 1, 2, 3 }; // Create an array with new operator and
                                      // array initializer.
int[] array2 = { 1, 2, 3 };           // Shortcut syntax with array initializer.
int[] array3 = new int[3];            // Equivalent to { 0, 0, 0 }
int[] array4 = null;                  // The array itself is an object, so it
                                      // can be set as null.
```

When declaring an array, [ ] will appear as part of the type at the beginning of the declaration (after the type name), or as part of the declarator for a particular variable (after variable name), or both:

```
int array5[];       /* equivalent to */  int[] array5;
int a, b[], c[][];  /* equivalent to */  int a; int[] b; int[][] c;
int[] a, b[];       /* equivalent to */  int[] a; int[][] b;
int a, []b, c[][];  /* Compilation Error, because [] is not part of the type at beginning
                       of the declaration, rather it is before 'b'. */
// The same rules apply when declaring a method that returns an array:
int foo()[] { ... } /* equivalent to */  int[] foo() { ... }
```

In the following example, both declarations are correct and can compile and run without any problems. However, both the Java Coding Convention and the Google Java Style Guide discourage the form with brackets after the variable name—the brackets identify the array type and should appear with the type designation. The same should be used for method return signatures.

```
float array[]; /* and */ int foo()[] { ... } /* are discouraged */
float[] array; /* and */ int[] foo() { ... } /* are encouraged */
```

The discouraged type is meant to accommodate transitioning C users, who are familiar with the syntax for C which has the brackets after the variable name.

In Java, it is possible to have arrays of size `0`:

```
int[] array = new int[0]; // Compiles and runs fine.
int[] array2 = {};        // Equivalent syntax.
```

However, since it's an empty array, no elements can be read from it or assigned to it:

```
array[0] = 1;     // Throws java.lang.ArrayIndexOutOfBoundsException.
int i = array2[0]; // Also throws ArrayIndexOutOfBoundsException.
```

Such empty arrays are typically useful as return values, so that the calling code only has to worry about dealing with an array, rather than a potential **`null`** value that may lead to a `NullPointerException`.

The length of an array must be a non-negative integer:

```
int[] array = new int[-1]; // Throws java.lang.NegativeArraySizeException
```

The array size can be determined using a public final field called `length`:

```
System.out.println(array.length); // Prints 0 in this case.
```

Note: `array.length` returns the actual size of the array and not the number of array elements which were assigned a value, unlike `ArrayList.size()` which returns the number of array elements which were assigned a value.

Creating and initializing multi-dimensional arrays

The simplest way to create a multi-dimensional array is as follows:

```
int[][] a = new int[2][3];
```

It will create two three-length **`int`** arrays—`a[0]` and `a[1]`. This is very similar to the classical, C-style initialization of rectangular multi-dimensional arrays.

You can create and initialize at the same time:

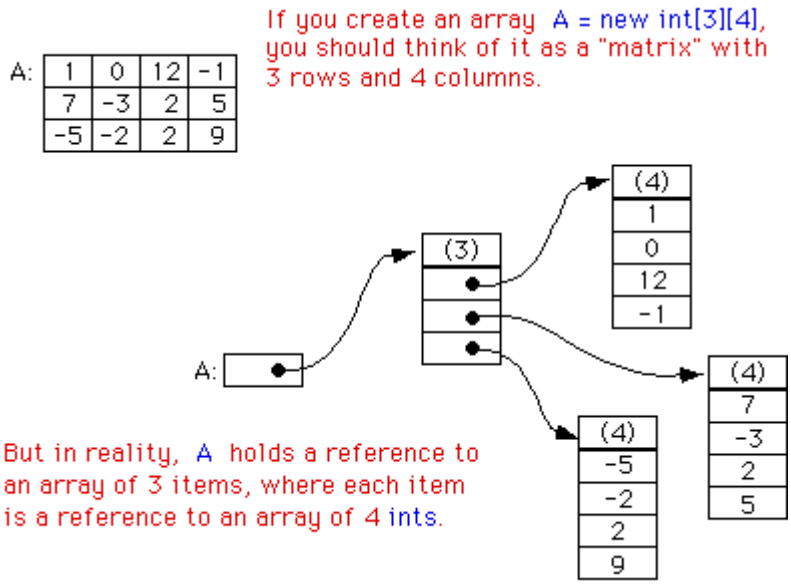
```
int[][] a = { {1, 2}, {3, 4}, {5, 6} };
```

Unlike C, where only rectangular multi-dimensional arrays are supported, inner arrays do not need to be of the same length, or even defined:

```
int[][] a = { {1}, {2, 3}, null };
```

Here, `a[0]` is a one-length **`int`** array, whereas `a[1]` is a two-length **`int`** array and `a[2]` is **`null`**. Arrays like this are called jagged arrays or ragged arrays, that is, they are arrays of arrays. Multi-dimensional arrays in Java are implemented as arrays of arrays, i.e. `array[i][j][k]` is equivalent to `((array[i])[j])[k]`. Unlike C#, the syntax `array[i,j]` is not supported in Java.

Multidimensional array representation in Java



Source - Live on Ideone

Creating and initializing reference type arrays

```
String[] array6 = new String[] { "Laurel", "Hardy" }; // Create an array with new
                                                      // operator and array initializer.
String[] array7 = { "Laurel", "Hardy" };              // Shortcut syntax with array
```

```
                                                    // initializer.
String[] array8 = new String[3];                    // { null, null, null }
String[] array9 = null;                             // null
```

Live on Ideone

In addition to the `String` literals and primitives shown above, the shortcut syntax for array initialization also works with canonical `Object` types:

```
Object[] array10 = { new Object(), new Object() };
```

Because arrays are covariant, a reference type array can be initialized as an array of a subclass, although an `ArrayStoreException` will be thrown if you try to set an element to something other than a `String`:

```
Object[] array11 = new String[] { "foo", "bar", "baz" };
array11[1] = "qux"; // fine
array11[1] = new StringBuilder(); // throws ArrayStoreException
```

The shortcut syntax cannot be used for this because the shortcut syntax would have an implicit type of `Object[]`.

An array can be initialized with zero elements by using `String[] emptyArray = new String[0]`. For example, an array with zero length like this is used for Creating an `Array` from a `Collection` when the method needs the runtime type of an object.

In both primitive and reference types, an empty array initialization (for example `String[] array8 = new String[3]`) will initialize the array with the default value for each data type.

Creating and initializing generic type arrays

In generic classes, arrays of generic types **cannot** be initialized like this due to type erasure:

```
public class MyGenericClass<T> {
    private T[] a;

    public MyGenericClass() {
        a = new T[5]; // Compile time error: generic array creation
    }
}
```

Instead, they can be created using one of the following methods: (note that these will generate unchecked warnings)

1. By creating an `Object` array, and casting it to the generic type:

   ```
   a = (T[]) new Object[5];
   ```

   This is the simplest method, but since the underlying array is still of type `Object[]`, this method does not provide type safety. Therefore, this method of creating an array is best used only within the generic class - not exposed publicly.

2. By using `Array.newInstance` with a class parameter:

   ```
   public MyGenericClass(Class<T> clazz) {
       a = (T[]) Array.newInstance(clazz, 5);
   ```

```
}
```

Here the class of `T` has to be explicitly passed to the constructor. The return type of `Array.newInstance` is always `Object`. However, this method is safer because the newly created array is always of type `T[]`, and therefore can be safely externalized.

Filling an array after initialization

Version ≥ Java SE 1.2

`Arrays.fill()` can be used to fill an array with **the same value** after initialization:

```
Arrays.fill(array8, "abc");        // { "abc", "abc", "abc" }
```

Live on Ideone

`fill()` can also assign a value to each element of the specified range of the array:

```
Arrays.fill(array8, 1, 2, "aaa");  // Placing "aaa" from index 1 to 2.
```

Live on Ideone

Version ≥ Java SE 8

Since Java version 8, the method `setAll`, and its `Concurrent` equivalent `parallelSetAll`, can be used to set every element of an array to generated values. These methods are passed a generator function which accepts an index and returns the desired value for that position.

The following example creates an integer array and sets all of its elements to their respective index value:

```
int[] array = new int[5];
Arrays.setAll(array, i -> i); // The array becomes { 0, 1, 2, 3, 4 }.
```

Live on Ideone

Separate declaration and initialization of arrays

The value of an index for an array element must be a whole number (0, 1, 2, 3, 4, ...) and less than the length of the array (indexes are zero-based). Otherwise, an ArrayIndexOutOfBoundsException will be thrown:

```
int[] array9;              // Array declaration - uninitialized
array9 = new int[3];       // Initialize array  - { 0, 0, 0 }
array9[0] = 10;            // Set index 0 value - { 10, 0, 0 }
array9[1] = 20;            // Set index 1 value - { 10, 20, 0 }
array9[2] = 30;            // Set index 2 value - { 10, 20, 30 }
```

Arrays may not be re-initialized with array initializer shortcut syntax

It is not possible to re-initialize an array via a shortcut syntax with an array initializer since an array initializer can only be specified in a field declaration or local variable declaration, or as a part of an array creation expression.

However, it is possible to create a new array and assign it to the variable being used to reference the old array. While this results in the array referenced by that variable being re-initialized, the variable contents are a completely new array. To do this, the **new** operator can be used with an array initializer and assigned to the array variable:

```
// First initialization of array
int[] array = new int[] { 1, 2, 3 };
```

```
// Prints "1 2 3 ".
for (int i : array) {
    System.out.print(i + " ");
}

// Re-initializes array to a new int[] array.
array = new int[] { 4, 5, 6 };

// Prints "4 5 6 ".
for (int i : array) {
    System.out.print(i + " ");
}

array = { 1, 2, 3, 4 }; // Compile-time error! Can't re-initialize an array via shortcut
                        // syntax with array initializer.
```

Live on Ideone

Section 23.2: Creating a List from an Array

The `Arrays.asList()` method can be used to return a fixed-size `List` containing the elements of the given array. The resulting `List` will be of the same parameter type as the base type of the array.

```
String[] stringArray = {"foo", "bar", "baz"};
List<String> stringList = Arrays.asList(stringArray);
```

Note: This list is backed by (*a view* of) the original array, meaning that any changes to the list will change the array and vice versa. However, changes to the list that would change its size (and hence the array length) will throw an exception.

To create a copy of the list, use the constructor of `java.util.ArrayList` taking a `Collection` as an argument:

Version ≥ Java SE 5

```
String[] stringArray = {"foo", "bar", "baz"};
List<String> stringList = new ArrayList<String>(Arrays.asList(stringArray));
```

Version ≥ Java SE 7

In Java SE 7 and later, a pair of angle brackets `<>` (empty set of type arguments) can be used, which is called the Diamond. The compiler can determine the type arguments from the context. This means the type information can be left out when calling the constructor of `ArrayList` and it will be inferred automatically during compilation. This is called Type Inference which is a part of Java Generics.

```
// Using Arrays.asList()

String[] stringArray = {"foo", "bar", "baz"};
List<String> stringList = new ArrayList<>(Arrays.asList(stringArray));

// Using ArrayList.addAll()

String[] stringArray = {"foo", "bar", "baz"};
ArrayList<String> list = new ArrayList<>();
list.addAll(Arrays.asList(stringArray));

// Using Collections.addAll()

String[] stringArray = {"foo", "bar", "baz"};
ArrayList<String> list = new ArrayList<>();
Collections.addAll(list, stringArray);
```

A point worth noting about the Diamond is that it cannot be used with Anonymous Classes.

Version ≥ Java SE 8

```
// Using Streams

int[] ints = {1, 2, 3};
List<Integer> list = Arrays.stream(ints).boxed().collect(Collectors.toList());

String[] stringArray = {"foo", "bar", "baz"};
List<Object> list = Arrays.stream(stringArray).collect(Collectors.toList());
```

Important notes related to using Arrays.asList() method

- This method returns `List`, which is an instance of `Arrays$ArrayList`(static inner class of `Arrays`) and not `java.util.ArrayList`. The resulting `List` is of fixed-size. That means, adding or removing elements is not supported and will throw an `UnsupportedOperationException`:

  ```
  stringList.add("something"); // throws java.lang.UnsupportedOperationException
  ```

- A new `List` can be created by passing an array-backed `List` to the constructor of a new `List`. This creates a new copy of the data, which has changeable size and that is not backed by the original array:

  ```
  List<String> modifiableList = new ArrayList<>(Arrays.asList("foo", "bar"));
  ```

- Calling `<T> List<T> asList(T... a)` on a primitive array, such as an `int[]`, will produce a `List<int[]>` whose only element is the source primitive array instead of the actual elements of the source array.

  The reason for this behavior is that primitive types cannot be used in place of generic type parameters, so the entire primitive array replaces the generic type parameter in this case. In order to convert a primitive array to a `List`, first of all, convert the primitive array to an array of the corresponding wrapper type (i.e. call `Arrays.asList` on an `Integer[]` instead of an `int[]`).

  Therefore, this will print `false`:

  ```
  int[] arr = {1, 2, 3};      // primitive array of int
  System.out.println(Arrays.asList(arr).contains(1));
  ```

  View Demo

  On the other hand, this will print `true`:

  ```
  Integer[] arr = {1, 2, 3};  // object array of Integer (wrapper for int)
  System.out.println(Arrays.asList(arr).contains(1));
  ```

  View Demo

  This will also print `true`, because the array will be interpreted as an `Integer[]`):

  ```
  System.out.println(Arrays.asList(1,2,3).contains(1));
  ```

  View Demo

Section 23.3: Creating an Array from a Collection

Two methods in `java.util.Collection` create an array from a collection:

- `Object[] toArray()`
- `<T> T[] toArray(T[] a)`

`Object[] toArray()` can be used as follows:

Version ≥ Java SE 5

```
Set<String> set = new HashSet<String>();
set.add("red");
set.add("blue");

// although set is a Set<String>, toArray() returns an Object[] not a String[]
Object[] objectArray = set.toArray();
```

`<T> T[] toArray(T[] a)` can be used as follows:

Version ≥ Java SE 5

```
Set<String> set = new HashSet<String>();
set.add("red");
set.add("blue");

// The array does not need to be created up front with the correct size.
// Only the array type matters. (If the size is wrong, a new array will
// be created with the same type.)
String[] stringArray = set.toArray(new String[0]);

// If you supply an array of the same size as collection or bigger, it
// will be populated with collection values and returned (new array
// won't be allocated)
String[] stringArray2 = set.toArray(new String[set.size()]);
```

The difference between them is more than just having untyped vs typed results. Their performance can differ as well (for details please read this performance analysis section):

- `Object[] toArray()` uses vectorized `arraycopy`, which is much faster than the type-checked `arraycopy` used in `T[] toArray(T[] a)`.
- `T[] toArray(new T[non-zero-size])` needs to zero-out the array at runtime, while `T[] toArray(new T[0])` does not. Such avoidance makes the latter call faster than the former. Detailed analysis here : Arrays of Wisdom of the Ancients.

Version ≥ Java SE 8

Starting from Java SE 8+, where the concept of `Stream` has been introduced, it is possible to use the `Stream` produced by the collection in order to create a new Array using the `Stream.toArray` method.

```
String[] strings = list.stream().toArray(String[]::new);
```

Examples taken from two answers (1, 2) to Converting 'ArrayList to 'String[]' in Java on Stack Overflow.

Section 23.4: Multidimensional and Jagged Arrays

It is possible to define an array with more than one dimension. Instead of being accessed by providing a single index, a multidimensional array is accessed by specifying an index for each dimension.

The declaration of multidimensional array can be done by adding [ ] for each dimension to a regular array declaration. For instance, to make a 2-dimensional `int` array, add another set of brackets to the declaration, such as `int[][]`. This continues for 3-dimensional arrays (`int[][][]`) and so forth.

To define a 2-dimensional array with three rows and three columns:

```
int rows = 3;
int columns = 3;
int[][] table = new int[rows][columns];
```

The array can be indexed and assign values to it with this construct. Note that the unassigned values are the default values for the type of an array, in this case `0` for `int`.

```
table[0][0] = 0;
table[0][1] = 1;
table[0][2] = 2;
```

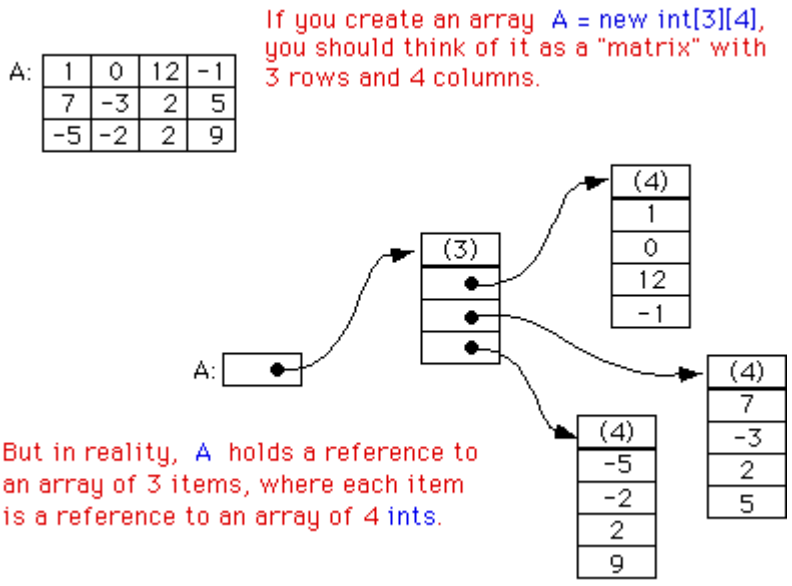
It is also possible to instantiate a dimension at a time, and even make non-rectangular arrays. These are more commonly referred to as **jagged arrays**.

```
int[][] nonRect = new int[4][];
```

It is important to note that although it is possible to define any dimension of jagged array, it's preceding level ***must*** be defined.

```
// valid
String[][] employeeGraph = new String[30][];

// invalid
int[][] unshapenMatrix = new int[][10];

// also invalid
int[][][] misshapenGrid = new int[100][][10];
```

How Multidimensional Arrays are represented in Java

> 
>
>
> *Image source: http://math.hws.edu/eck/cs124/javanotes3/c8/s5.html*

Jagged array literal intialization

Multidimensional arrays and jagged arrays can also be initialized with a literal expression. The following declares and populates a 2x3 `int` array:

```
int[][] table = {
    {1, 2, 3},
    {4, 5, 6}
};
```

Note: Jagged subarrays may also be `null`. For instance, the following code declares and populates a two dimensional `int` array whose first subarray is `null`, second subarray is of zero length, third subarray is of one length and the last subarray is a two length array:

```
int[][] table = {
    null,
    {},
    {1},
    {1,2}
};
```

For multidimensional array it is possible to extract arrays of lower-level dimension by their indices:

```
int[][][] arr = new int[3][3][3];
int[][] arr1 = arr[0]; // get first 3x3-dimensional array from arr
int[] arr2 = arr1[0]; // get first 3-dimensional array from arr1
int[] arr3 = arr[0]; // error: cannot convert from int[][] to int[]
```

Section 23.5: ArrayIndexOutOfBoundsException

The `ArrayIndexOutOfBoundsException` is thrown when a non-existing index of an array is being accessed.

Arrays are zero-based indexed, so the index of the first element is `0` and the index of the last element is the array capacity minus 1 (i.e. `array.length - 1`).

Therefore, any request for an array element by the index `i` has to satisfy the condition `0 <= i < array.length`, otherwise the `ArrayIndexOutOfBoundsException` will be thrown.

The following code is a simple example where an `ArrayIndexOutOfBoundsException` is thrown.

```
String[] people = new String[] { "Carol", "Andy" };

// An array will be created:
// people[0]: "Carol"
// people[1]: "Andy"

// Notice: no item on index 2. Trying to access it triggers the exception:
System.out.println(people[2]);  // throws an ArrayIndexOutOfBoundsException.
```

Output:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2
at your.package.path.method(YourClass.java:15)
```

Note that the illegal index that is being accessed is also included in the exception (2 in the example); this information could

be useful to find the cause of the exception.

To avoid this, simply check that the index is within the limits of the array:

```
int index = 2;
if (index >= 0 && index < people.length) {
    System.out.println(people[index]);
}
```

Section 23.6: Array Covariance

Object arrays are covariant, which means that just as `Integer` is a subclass of `Number`, `Integer[]` is a subclass of `Number[]`. This may seem intuitive, but can result in surprising behavior:

```
Integer[] integerArray = {1, 2, 3};
Number[] numberArray = integerArray;  // valid
Number firstElement = numberArray[0]; // valid
numberArray[0] = 4L;                  // throws ArrayStoreException at runtime
```

Although `Integer[]` is a subclass of `Number[]`, it can only hold `Integer`s, and trying to assign a `Long` element throws a runtime exception.

Note that this behavior is unique to arrays, and can be avoided by using a generic `List` instead:

```
List<Integer> integerList = Arrays.asList(1, 2, 3);
//List<Number> numberList = integerList;  // compile error
List<? extends Number> numberList = integerList;
Number firstElement = numberList.get(0);
//numberList.set(0, 4L);                  // compile error
```

It's not necessary for all of the array elements to share the same type, as long as they are a subclass of the array's type:

```
interface I {}

class A implements I {}
class B implements I {}
class C implements I {}

I[] array10 = new I[] { new A(), new B(), new C() }; // Create an array with new
                                                     // operator and array initializer.

I[] array11 = { new A(), new B(), new C() };         // Shortcut syntax with array
                                                     // initializer.

I[] array12 = new I[3];                              // { null, null, null }

I[] array13 = new A[] { new A(), new A() };          // Works because A implements I.

Object[] array14 = new Object[] { "Hello, World!", 3.14159, 42 }; // Create an array with
                                                     // new operator and array initializer.

Object[] array15 = { new A(), 64, "My String" };     // Shortcut syntax
                                                     // with array initializer.
```

Section 23.7: Arrays to Stream

Version ≥ Java SE 8

Converting an array of objects to `Stream`:

```
String[] arr = new String[] {"str1", "str2", "str3"};
Stream<String> stream = Arrays.stream(arr);
```

Converting an array of primitives to `Stream` using `Arrays.stream()` will transform the array to a primitive specialization of Stream:

```
int[] intArr = {1, 2, 3};
IntStream intStream = Arrays.stream(intArr);
```

You can also limit the `Stream` to a range of elements in the array. The start index is inclusive and the end index is exclusive:

```
int[] values = {1, 2, 3, 4};
IntStream intStream = Arrays.stream(values, 2, 4);
```

A method similar to `Arrays.stream()` appears in the `Stream` class: `Stream.of()`. The difference is that `Stream.of()` uses a varargs parameter, so you can write something like:

```
Stream<Integer> intStream = Stream.of(1, 2, 3);
Stream<String> stringStream = Stream.of("1", "2", "3");
Stream<Double> doubleStream = Stream.of(new Double[]{1.0, 2.0});
```

Section 23.8: Iterating over arrays

You can iterate over arrays either by using enhanced for loop (aka foreach) or by using array indices:

```
int[] array = new int[10];

// using indices: read and write
for (int i = 0; i < array.length; i++) {
    array[i] = i;
}
```

Version ≥ Java SE 5

```
// extended for: read only
for (int e : array) {
    System.out.println(e);
}
```

It is worth noting here that there is no direct way to use an Iterator on an Array, but through the Arrays library it can be easily converted to a list to obtain an `Iterable` object.

For boxed arrays use Arrays.asList:

```
Integer[] boxed = {1, 2, 3};
Iterable<Integer> boxedIt = Arrays.asList(boxed); // list-backed iterable
Iterator<Integer> fromBoxed1 = boxedIt.iterator();
```

For primitive arrays (using java 8) use streams (specifically in this example - Arrays.stream -> IntStream):

```
int[] primitives = {1, 2, 3};
```

```
IntStream primitiveStream = Arrays.stream(primitives); // list-backed iterable
PrimitiveIterator.OfInt fromPrimitive1 = primitiveStream.iterator();
```

If you can't use streams (no java 8), you can choose to use google's guava library:

```
Iterable<Integer> fromPrimitive2 = Ints.asList(primitives);
```

In two-dimensional arrays or more, both techniques can be used in a slightly more complex fashion.

Example:

```
int[][] array = new int[10][10];

for (int indexOuter = 0; indexOuter < array.length; indexOuter++) {
    for (int indexInner = 0; indexInner < array[indexOuter].length; indexInner++) {
        array[indexOuter][indexInner] = indexOuter + indexInner;
    }
}
```

Version ≥ Java SE 5

```
for (int[] numbers : array) {
    for (int value : numbers) {
        System.out.println(value);
    }
}
```

It is impossible to set an Array to any non-uniform value without using an index based loop.

Of course you can also use `while` or `do-while` loops when iterating using indices.

One note of caution: when using array indices, make sure the index is between `0` and `array.length - 1` (both inclusive). Don't make hard coded assumptions on the array length otherwise you might break your code if the array length changes but your hard coded values don't.

Example:

```
int[] numbers = {1, 2, 3, 4};

public void incrementNumbers() {
    // DO THIS :
    for (int i = 0; i < numbers.length; i++) {
        numbers[i] += 1; //or this: numbers[i] = numbers[i] + 1; or numbers[i]++;
    }

    // DON'T DO THIS :
    for (int i = 0; i < 4; i++) {
        numbers[i] += 1;
    }
}
```

It's also best if you don't use fancy calculations to get the index but use the index to iterate and if you need different values calculate those.

Example:

```
public void fillArrayWithDoubleIndex(int[] array) {
    // DO THIS :
    for (int i = 0; i < array.length; i++) {
        array[i] = i * 2;
```

```
    }

    // DON'T DO THIS :
    int doubleLength = array.length * 2;
    for (int i = 0; i < doubleLength; i += 2) {
        array[i / 2] = i;
    }
}
```

Accessing Arrays in reverse order

```
int[] array = {0, 1, 1, 2, 3, 5, 8, 13};
for (int i = array.length - 1; i >= 0; i--) {
   System.out.println(array[i]);
}
```

Using temporary Arrays to reduce code repetition

Iterating over a temporary array instead of repeating code can make your code cleaner. It can be used where the same operation is performed on multiple variables.

```
// we want to print out all of these
String name = "Margaret";
int eyeCount = 16;
double height = 50.2;
int legs = 9;
int arms = 5;


// copy-paste approach:
System.out.println(name);
System.out.println(eyeCount);
System.out.println(height);
System.out.println(legs);
System.out.println(arms);


// temporary array approach:
for(Object attribute : new Object[]{name, eyeCount, height, legs, arms})
    System.out.println(attribute);

// using only numbers
for(double number : new double[]{eyeCount, legs, arms, height})
    System.out.println(Math.sqrt(number));
```

Keep in mind that this code should not be used in performance-critical sections, as an array is created every time the loop is entered, and that primitive variables will be copied into the array and thus cannot be modified.

Section 23.9: Arrays to a String

Version ≥ Java SE 5

Since Java 1.5 you can get a `String` representation of the contents of the specified array without iterating over its every element. Just use `Arrays.toString(Object[])` or `Arrays.deepToString(Object[])` for multidimentional arrays:

```
int[] arr = {1, 2, 3, 4, 5};
System.out.println(Arrays.toString(arr));      // [1, 2, 3, 4, 5]
```

```
int[][] arr = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
System.out.println(Arrays.deepToString(arr));  // [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

`Arrays.toString()` method uses `Object.toString()` method to produce `String` values of every item in the array, beside primitive type array, it can be used for all type of arrays. For instance:

```
public class Cat { /* implicitly extends Object */
    @Override
    public String toString() {
      return "CAT!";
    }
}

Cat[] arr = { new Cat(), new Cat() };
System.out.println(Arrays.toString(arr));        // [CAT!, CAT!]
```

If no overridden `toString()` exists for the class, then the inherited `toString()` from `Object` will be used. Usually the output is then not very useful, for example:

```
public class Dog {
    /* implicitly extends Object */
}

Dog[] arr = { new Dog() };
System.out.println(Arrays.toString(arr));        // [Dog@17ed40e0]
```

Section 23.10: Sorting arrays

Sorting arrays can be easily done with the Arrays api.

```
import java.util.Arrays;

// creating an array with integers
int[] array = {7, 4, 2, 1, 19};
// this is the sorting part just one function ready to be used
Arrays.sort(array);
// prints [1, 2, 4, 7, 19]
System.out.println(Arrays.toString(array));
```

Sorting String arrays:

`String` is not a numeric data, it defines it's own order which is called lexicographic order, also known as alphabetic order. When you sort an array of String using `sort()` method, it sorts array into natural order defined by Comparable interface, as shown below :

Increasing Order

```
String[] names = {"John", "Steve", "Shane", "Adam", "Ben"};
System.out.println("String array before sorting : " + Arrays.toString(names));
Arrays.sort(names);
System.out.println("String array after sorting in ascending order : " + Arrays.toString(names));
```

Output:

```
String array before sorting : [John, Steve, Shane, Adam, Ben]
String array after sorting in ascending order : [Adam, Ben, John, Shane, Steve]
```

Decreasing Order

```
Arrays.sort(names, 0, names.length, Collections.reverseOrder());
System.out.println("String array after sorting in descending order : " + Arrays.toString(names));
```

Output:

```
String array after sorting in descending order : [Steve, Shane, John, Ben, Adam]
```

Sorting an Object array

In order to sort an object array, all elements must implement either `Comparable` or `Comparator` interface to define the order of the sorting.

We can use either `sort(Object[])` method to sort an object array on its natural order, but you must ensure that all elements in the array must implement `Comparable`.

Furthermore, they must be mutually comparable as well, for example `e1.compareTo(e2)` must not throw a *`ClassCastException`* for any elements e1 and e2 in the array. Alternatively you can sort an Object array on custom order using `sort(T[], Comparator)` method as shown in following example.

```
// How to Sort Object Array in Java using Comparator and Comparable
Course[] courses = new Course[4];
courses[0] = new Course(101, "Java", 200);
courses[1] = new Course(201, "Ruby", 300);
courses[2] = new Course(301, "Python", 400);
courses[3] = new Course(401, "Scala", 500);

System.out.println("Object array before sorting : " + Arrays.toString(courses));

Arrays.sort(courses);
System.out.println("Object array after sorting in natural order : " + Arrays.toString(courses));

Arrays.sort(courses, new Course.PriceComparator());
System.out.println("Object array after sorting by price : " + Arrays.toString(courses));

Arrays.sort(courses, new Course.NameComparator());
System.out.println("Object array after sorting by name : " + Arrays.toString(courses));
```

Output:

```
Object array before sorting : [#101 Java@200 , #201 Ruby@300 , #301 Python@400 , #401 Scala@500 ]
Object array after sorting in natural order : [#101 Java@200 , #201 Ruby@300 , #301 Python@400 ,
#401 Scala@500 ]
Object array after sorting by price : [#101 Java@200 , #201 Ruby@300 , #301 Python@400 , #401
Scala@500 ]
Object array after sorting by name : [#101 Java@200 , #301 Python@400 , #201 Ruby@300 , #401
Scala@500 ]
```

Section 23.11: Getting the Length of an Array

Arrays are objects which provide space to store up to its size of elements of specified type. An array's size can not be modified after the array is created.

```
int[] arr1 = new int[0];
int[] arr2 = new int[2];
int[] arr3 = new int[]{1, 2, 3, 4};
int[] arr4 = {1, 2, 3, 4, 5, 6, 7};

int len1 = arr1.length; // 0
int len2 = arr2.length; // 2
int len3 = arr3.length; // 4
int len4 = arr4.length; // 7
```

The `length` field in an array stores the size of an array. It is a **`final`** field and cannot be modified.

This code shows the difference between the `length` of an array and amount of objects an array stores.

```
public static void main(String[] args) {
    Integer arr[] = new Integer[] {1,2,3,null,5,null,7,null,null,null,11,null,13};

    int arrayLength = arr.length;
    int nonEmptyElementsCount = 0;

    for (int i=0; i<arrayLength; i++) {
        Integer arrElt = arr[i];
        if (arrElt != null) {
            nonEmptyElementsCount++;
        }
    }

    System.out.println("Array 'arr' has a length of "+arrayLength+"\n"
                           + "and it contains "+nonEmptyElementsCount+" non-empty values");
}
```

Result:

```
Array 'arr' has a length of 13
and it contains 7 non-empty values
```

Section 23.12: Finding an element in an array

There are many ways find the location of a value in an array. The following example snippets all assume that the array is one of the following:

```
  String[] strings = new String[] { "A", "B", "C" };
  int[] ints = new int[] { 1, 2, 3, 4 };
```

In addition, each one sets `index` or `index2` to either the index of required element, or `-1` if the element is not present.

Using `Arrays.binarySearch` (for sorted arrays only)

```
  int index = Arrays.binarySearch(strings, "A");
  int index2 = Arrays.binarySearch(ints, 1);
```

Using a `Arrays.asList` (for non-primitive arrays only)

```
int index = Arrays.asList(strings).indexOf("A");
int index2 = Arrays.asList(ints).indexOf(1);  // compilation error
```

Using a `Stream`

Version ≥ Java SE 8

```
int index = IntStream.range(0, strings.length)
                     .filter(i -> "A".equals(strings[i]))
                     .findFirst()
                     .orElse(-1); // If not present, gives us -1.
// Similar for an array of primitives
```

Linear search using a loop

```
int index = -1;
for (int i = 0; i < array.length; i++) {
    if ("A".equals(array[i])) {
        index = i;
        break;
    }
}
// Similar for an array of primitives
```

Linear search using 3rd-party libraries such as org.apache.commons

```
int index = org.apache.commons.lang3.ArrayUtils.contains(strings, "A");
int index2 = org.apache.commons.lang3.ArrayUtils.contains(ints, 1);
```

Note: Using a direct linear search is more efficient than wrapping in a list.

Testing if an array contains an element

The examples above can be adapted to test if the array contains an element by simply testing to see if the index computed is greater or equal to zero.

Alternatively, there are also some more concise variations:

```
boolean isPresent = Arrays.asList(strings).contains("A");
```

Version ≥ Java SE 8

```
boolean isPresent = Stream<String>.of(strings).anyMatch(x -> "A".equals(x));
```

```
boolean isPresent = false;
for (String s : strings) {
    if ("A".equals(s)) {
        isPresent = true;
        break;
    }
}

boolean isPresent = org.apache.commons.lang3.ArrayUtils.contains(ints, 4);
```

Section 23.13: How do you change the size of an array?

The simple answer is that you cannot do this. Once an array has been created, its size cannot be changed. Instead, an array can only be "resized" by creating a new array with the appropriate size and copying the elements from the existing array to the new one.

```
String[] listOfCities = new String[3];   // array created with size 3.
listOfCities[0] = "New York";
```

```
listOfCities[1] = "London";
listOfCities[2] = "Berlin";
```

Suppose (for example) that a new element needs to be added to the `listOfCities` array defined as above. To do this, you will need to:

1. create a new array with size 4,
2. copy the existing 3 elements of the old array to the new array at offsets 0, 1 and 2, and
3. add the new element to the new array at offset 3.

There are various ways to do the above. Prior to Java 6, the most concise way was:

```
String[] newArray = new String[listOfCities.length + 1];
System.arraycopy(listOfCities, 0, newArray, 0, listOfCities.length);
newArray[listOfCities.length] = "Sydney";
```

From Java 6 onwards, the `Arrays.copyOf` and `Arrays.copyOfRange` methods can do this more simply:

```
String[] newArray = Arrays.copyOf(listOfCities, listOfCities.length + 1);
newArray[listOfCities.length] = "Sydney";
```

For other ways to copy an array, refer to the following example. Bear in mind that you need an array copy with a different length to the original when resizing.

- Copying arrays

A better alternatives to array resizing

There two major drawbacks with resizing an array as described above:

- It is inefficient. Making an array bigger (or smaller) involves copying many or all of the existing array elements, and allocating a new array object. The larger the array, the more expensive it gets.
- You need to be able to update any "live" variables that contain references to the old array.

One alternative is to create the array with a large enough size to start with. This is only viable if you can determine that size accurately *before allocating the array*. If you cannot do that, then the problem of resizing the array arises again.

The other alternative is to use a data structure class provided by the Java SE class library or a third-party library. For example, the Java SE "collections" framework provides a number of implementations of the `List`, `Set` and `Map` APIs with different runtime properties. The `ArrayList` class is closest to performance characteristics of a plain array (e.g. O(N) lookup, O(1) get and set, O(N) random insertion and deletion) while providing more efficient resizing without the reference update problem.

(The resize efficiency for `ArrayList` comes from its strategy of doubling the size of the backing array on each resize. For a typical use-case, this means that you only resize occasionally. When you amortize over the lifetime of the list, the resize cost per insert is `O(1)`. It may be possible to use the same strategy when resizing a plain array.)

Section 23.14: Converting arrays between primitives and boxed types

Sometimes conversion of primitive types to boxed types is necessary.

To convert the array, it's possible to use streams (in Java 8 and above):

Version ≥ Java SE 8

```
int[] primitiveArray = {1, 2, 3, 4};
Integer[] boxedArray =
    Arrays.stream(primitiveArray).boxed().toArray(Integer[]::new);
```

With lower versions it can be by iterating the primitive array and explicitly copying it to the boxed array:

Version < Java SE 8

```
int[] primitiveArray = {1, 2, 3, 4};
Integer[] boxedArray = new Integer[primitiveArray.length];
for (int i = 0; i < primitiveArray.length; ++i) {
    boxedArray[i] = primitiveArray[i]; // Each element is autoboxed here
}
```

Similarly, a boxed array can be converted to an array of its primitive counterpart:

Version ≥ Java SE 8

```
Integer[] boxedArray = {1, 2, 3, 4};
int[] primitiveArray =
    Arrays.stream(boxedArray).mapToInt(Integer::intValue).toArray();
```

Version < Java SE 8

```
Integer[] boxedArray = {1, 2, 3, 4};
int[] primitiveArray = new int[boxedArray.length];
for (int i = 0; i < boxedArray.length; ++i) {
    primitiveArray[i] = boxedArray[i]; // Each element is outboxed here
}
```

Section 23.15: Remove an element from an array

Java doesn't provide a direct method in `java.util.Arrays` to remove an element from an array. To perform it, you can either copy the original array to a new one without the element to remove or convert your array to another structure allowing the removal.

Using ArrayList

You can convert the array to a `java.util.List`, remove the element and convert the list back to an array as follows:

```
String[] array = new String[]{"foo", "bar", "baz"};

List<String> list = new ArrayList<>(Arrays.asList(array));
list.remove("foo");

// Creates a new array with the same size as the list and copies the list
// elements to it.
array = list.toArray(new String[list.size()]);

System.out.println(Arrays.toString(array)); //[bar, baz]
```

Using System.arraycopy

`System.arraycopy()` can be used to make a copy of the original array and remove the element you want. Below an example:

```
int[] array = new int[] { 1, 2, 3, 4 }; // Original array.
int[] result = new int[array.length - 1]; // Array which will contain the result.
int index = 1; // Remove the value "2".
```

```
// Copy the elements at the left of the index.
System.arraycopy(array, 0, result, 0, index);
// Copy the elements at the right of the index.
System.arraycopy(array, index + 1, result, index, array.length - index - 1);

System.out.println(Arrays.toString(result)); //[1, 3, 4]
```

Using Apache Commons Lang

To easily remove an element, you can use the Apache Commons Lang library and especially the static method `removeElement()` of the class `ArrayUtils`. Below an example:

```
int[] array = new int[]{1,2,3,4};
array = ArrayUtils.removeElement(array, 2); //remove first occurrence of 2
System.out.println(Arrays.toString(array)); //[1, 3, 4]
```

Section 23.16: Comparing arrays for equality

Array types inherit their `equals()` (and `hashCode()`) implementations from java.lang.Object, so `equals()` will only return true when comparing against the exact same array object. To compare arrays for equality based on their values, use `java.util.Arrays.equals`, which is overloaded for all array types.

```
int[] a = new int[]{1, 2, 3};
int[] b = new int[]{1, 2, 3};
System.out.println(a.equals(b)); //prints "false" because a and b refer to different objects
System.out.println(Arrays.equals(a, b)); //prints "true" because the elements of a and b have the
same values
```

When the element type is a reference type, `Arrays.equals()` calls `equals()` on the array elements to determine equality. In particular, if the element type is itself an array type, identity comparison will be used. To compare multidimensional arrays for equality, use `Arrays.deepEquals()` instead as below:

```
int a[] = { 1, 2, 3 };
int b[] = { 1, 2, 3 };

Object[] aObject = { a }; // aObject contains one element
Object[] bObject = { b }; // bObject contains one element

System.out.println(Arrays.equals(aObject, bObject)); // false
System.out.println(Arrays.deepEquals(aObject, bObject));// true
```

Because sets and maps use `equals()` and `hashCode()`, arrays are generally not useful as set elements or map keys. Either wrap them in a helper class that implements `equals()` and `hashCode()` in terms of the array elements, or convert them to `List` instances and store the lists.

Section 23.17: Copying arrays

Java provides several ways to copy an array.

for loop

```
int[] a = { 4, 1, 3, 2 };
int[] b = new int[a.length];
for (int i = 0; i < a.length; i++) {
    b[i] = a[i];
}
```

Note that using this option with an Object array instead of primitive array will fill the copy with reference to the original content instead of copy of it.

Object.clone()

Since arrays are `Object`s in Java, you can use `Object.clone()`.

```
int[] a = { 4, 1, 3, 2 };
int[] b = a.clone(); // [4, 1, 3, 2]
```

Note that the `Object.clone` method for an array performs a **shallow copy**, i.e. it returns a reference to a new array which references the **same** elements as the source array.

Arrays.copyOf()

`java.util.Arrays` provides an easy way to perform the copy of an array to another. Here is the basic usage:

```
int[] a = {4, 1, 3, 2};
int[] b = Arrays.copyOf(a, a.length); // [4, 1, 3, 2]
```

Note that `Arrays.copyOf` also provides an overload which allows you to change the type of the array:

```
Double[] doubles = { 1.0, 2.0, 3.0 };
Number[] numbers = Arrays.copyOf(doubles, doubles.length, Number[].class);
```

System.arraycopy()

> `public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)`
> Copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array.

Below an example of use

```
int[] a = { 4, 1, 3, 2 };
int[] b = new int[a.length];
System.arraycopy(a, 0, b, 0, a.length); // [4, 1, 3, 2]
```

Arrays.copyOfRange()

Mainly used to copy a part of an Array, you can also use it to copy whole array to another as below:

```
int[] a = { 4, 1, 3, 2 };
int[] b = Arrays.copyOfRange(a, 0, a.length); // [4, 1, 3, 2]
```

Section 23.18: Casting Arrays

Arrays are objects, but their type is defined by the type of the contained objects. Therefore, one cannot just cast `A[]` to `T[]`, but each A member of the specific `A[]` must be cast to a `T` object. Generic example:

```
public static <T, A> T[] castArray(T[] target, A[] array) {
    for (int i = 0; i < array.length; i++) {
        target[i] = (T) array[i];
```

```
    }
    return target;
}
```

Thus, given an `A[]` array:

```
T[] target = new T[array.Length];
target = castArray(target, array);
```

Java SE provides the method `Arrays.copyOf(original, newLength, newType)` for this purpose:

```
Double[] doubles = { 1.0, 2.0, 3.0 };
Number[] numbers = Arrays.copyOf(doubles, doubles.length, Number[].class);
```

Chapter 24: Collections

The collections framework in `java.util` provides a number of generic classes for sets of data with functionality that can't be provided by regular arrays.

Collections framework contains interfaces for `Collection<O>`, with main sub-interfaces `List<O>` and `Set<O>`, and mapping collection `Map<K, V>`. Collections are the root interface and are being implemented by many other collection frameworks.

Section 24.1: Removing items from a List within a loop

It is tricky to remove items from a list while within a loop, this is due to the fact that the index and length of the list gets changed.

Given the following list, here are some examples that will give an unexpected result and some that will give the correct result.

```
List<String> fruits = new ArrayList<String>();
fruits.add("Apple");
fruits.add("Banana");
fruits.add("Strawberry");
```

INCORRECT
Removing in iteration of `for` statement *Skips "Banana":*

The code sample will only print `Apple` and `Strawberry`. `Banana` is skipped because it moves to index `0` once `Apple` is deleted, but at the same time `i` gets incremented to `1`.

```
for (int i = 0; i < fruits.size(); i++) {
    System.out.println (fruits.get(i));
    if ("Apple".equals(fruits.get(i))) {
         fruits.remove(i);
    }
}
```

Removing in the enhanced `for` statement *Throws Exception:*

Because of iterating over collection and modifying it at the same time.

> Throws: java.util.ConcurrentModificationException

```
for (String fruit : fruits) {
    System.out.println(fruit);
    if ("Apple".equals(fruit)) {
        fruits.remove(fruit);
    }
}
```

CORRECT
Removing in while loop using an `Iterator`

```
Iterator<String> fruitIterator = fruits.iterator();
while(fruitIterator.hasNext()) {
    String fruit = fruitIterator.next();
    System.out.println(fruit);
    if ("Apple".equals(fruit)) {
```

```
        fruitIterator.remove();
    }
}
```

The `Iterator` interface has a `remove()` method built in just for this case. However, this method is marked as "optional" in the documentation, and it might throw an `UnsupportedOperationException`.

> Throws: UnsupportedOperationException - if the remove operation is not supported by this iterator

Therefore, it is advisable to check the documentation to make sure this operation is supported (in practice, unless the collection is an immutable one obtained through a 3rd party library or the use of one of the `Collections.unmodifiable...()` method, the operation is almost always supported).

While using an `Iterator` a `ConcurrentModificationException` is thrown when the `modCount` of the `List` is changed from when the `Iterator` was created. This could have happened in the same thread or in a multi-threaded application sharing the same list.

A `modCount` is an **`int`** variable which counts the number of times this list has been structurally modified. A structural change essentially means an `add()` or `remove()` operation being invoked on `Collection` object (changes made by `Iterator` are not counted). When the `Iterator` is created, it stores this `modCount` and on every iteration of the `List` checks if the current `modCount` is same as and when the `Iterator` was created. If there is a change in the `modCount` value it throws a `ConcurrentModificationException`.

Hence for the above-declared list, an operation like below will not throw any exception:

```
Iterator<String> fruitIterator = fruits.iterator();
fruits.set(0, "Watermelon");
while(fruitIterator.hasNext()){
    System.out.println(fruitIterator.next());
}
```

But adding a new element to the `List` after initializing an `Iterator` will throw a `ConcurrentModificationException`:

```
Iterator<String> fruitIterator = fruits.iterator();
fruits.add("Watermelon");
while(fruitIterator.hasNext()){
    System.out.println(fruitIterator.next());    //ConcurrentModificationException here
}
```

Iterating backwards

```
for (int i = (fruits.size() - 1); i >=0; i--) {
    System.out.println (fruits.get(i));
    if ("Apple".equals(fruits.get(i))) {
         fruits.remove(i);
    }
}
```

This does not skip anything. The downside of this approach is that the output is reverse. However, in most cases where you remove items that will not matter. You should never do this with `LinkedList`.

Iterating forward, adjusting the loop index

```
for (int i = 0; i < fruits.size(); i++) {
    System.out.println (fruits.get(i));
```

```
    if ("Apple".equals(fruits.get(i))) {
        fruits.remove(i);
        i--;
    }
}
```

This does not skip anything. When the `i`th element is removed from the `List`, the element originally positioned at index `i+1` becomes the new `i`th element. Therefore, the loop can decrement `i` in order for the next iteration to process the next element, without skipping.

Using a "should-be-removed" list

```
ArrayList shouldBeRemoved = new ArrayList();
for (String str : currentArrayList) {
    if (condition) {
        shouldBeRemoved.add(str);
    }
}
currentArrayList.removeAll(shouldBeRemoved);
```

This solution enables the developer to check if the correct elements are removed in a cleaner way.

`Version ≥ Java SE 8`

In Java 8 the following alternatives are possible. These are cleaner and more straight forward if the removing does not have to happen in a loop.

Filtering a Stream

A `List` can be streamed and filtered. A proper filter can be used to remove all undesired elements.

```
List<String> filteredList =
    fruits.stream().filter(p -> !"Apple".equals(p)).collect(Collectors.toList());
```

Note that unlike all the other examples here, this example produces a new `List` instance and keeps the original `List` unchanged.

Using `removeIf`

Saves the overhead of constructing a stream if all that is needed is to remove a set of items.

```
fruits.removeIf(p -> "Apple".equals(p));
```

Section 24.2: Constructing collections from existing data

Standard Collections
Java Collections framework

A simple way to construct a `List` from individual data values is to use `java.utils.Arrays` method `Arrays.asList`:

```
List<String> data = Arrays.asList("ab", "bc", "cd", "ab", "bc", "cd");
```

All standard collection implementations provide constructors that take another collection as an argument adding all elements to the new collection at the time of construction:

```
List<String> list = new ArrayList<>(data); // will add data as is
Set<String> set1 = new HashSet<>(data); // will add data keeping only unique values
```

```
SortedSet<String> set2 = new TreeSet<>(data); // will add data keeping unique values and sorting
Set<String> set3 = new LinkedHashSet<>(data); // will add data keeping only unique values and
preserving the original order
```

Google Guava Collections framework

Another great framework is `Google Guava` that is amazing utility class (providing convenience static methods) for construction of different types of standard collections `Lists` and `Sets`:

```
import com.google.common.collect.Lists;
import com.google.common.collect.Sets;
...
List<String> list1 = Lists.newArrayList("ab", "bc", "cd");
List<String> list2 = Lists.newArrayList(data);
Set<String> set4 = Sets.newHashSet(data);
SortedSet<String> set5 = Sets.newTreeSet("bc", "cd", "ab", "bc", "cd");
```

Mapping Collections

Java Collections framework

Similarly for maps, given a `Map<String, Object> map` a new map can be constructed with all elements as follows:

```
Map<String, Object> map1 = new HashMap<>(map);
SortedMap<String, Object> map2 = new TreeMap<>(map);
```

Apache Commons Collections framework

Using `Apache Commons` you can create Map using array in `ArrayUtils.toMap` as well as `MapUtils.toMap`:

```
import org.apache.commons.lang3.ArrayUtils;
...
// Taken from org.apache.commons.lang.ArrayUtils#toMap JavaDoc

// Create a Map mapping colors.
Map colorMap = MapUtils.toMap(new String[][] {{
    {"RED", "#FF0000"},
    {"GREEN", "#00FF00"},
    {"BLUE", "#0000FF"}});
```

Each element of the array must be either a Map.Entry or an Array, containing at least two elements, where the first element is used as key and the second as value.

Google Guava Collections framework

Utility class from `Google Guava` framework is named `Maps`:

```
import com.google.common.collect.Maps;
...
void howToCreateMapsMethod(Function<? super K,V> valueFunction,
          Iterable<K> keys1,
          Set<K> keys2,
          SortedSet<K> keys3) {
    ImmutableMap<K, V> map1 = toMap(keys1, valueFunction); // Immutable copy
    Map<K, V> map2 = asMap(keys2, valueFunction); // Live Map view
    SortedMap<K, V> map3 = toMap(keys3, valueFunction); // Live Map view
}
```

Version ≥ Java SE 8

Using `Stream`,

```
Stream.of("xyz", "abc").collect(Collectors.toList());
```

or

```
Arrays.stream("xyz", "abc").collect(Collectors.toList());
```

Section 24.3: Declaring an ArrayList and adding objects

We can create an `ArrayList` (following the `List` interface):

```
List aListOfFruits = new ArrayList();
```

Version ≥ Java SE 5

```
List<String> aListOfFruits = new ArrayList<String>();
```

Version ≥ Java SE 7

```
List<String> aListOfFruits = new ArrayList<>();
```

Now, use the method `add` to add a `String`:

```
aListOfFruits.add("Melon");
aListOfFruits.add("Strawberry");
```

In the above example, the `ArrayList` will contain the `String` "Melon" at index 0 and the `String` "Strawberry" at index 1.

Also we can add multiple elements with `addAll(Collection<? extends E> c)` method

```
List<String> aListOfFruitsAndVeggies = new ArrayList<String>();
aListOfFruitsAndVeggies.add("Onion");
aListOfFruitsAndVeggies.addAll(aListOfFruits);
```

Now "Onion" is placed at 0 index in `aListOfFruitsAndVeggies`, "Melon" is at index 1 and "Strawberry" is at index 2.

Section 24.4: Iterating over Collections

Iterating over List

```
List<String> names  = new ArrayList<>(Arrays.asList("Clementine", "Duran", "Mike"));
```

Version ≥ Java SE 8

```
names.forEach(System.out::println);
```

If we need parallelism use

```
names.parallelStream().forEach(System.out::println);
```

Version ≥ Java SE 5

```
for (String name : names) {
    System.out.println(name);
}
```

Version < Java SE 5

```
for (int i = 0; i < names.size(); i++) {
    System.out.println(names.get(i));
}
```

Version ≥ Java SE 1.2

```
//Creates ListIterator which supports both forward as well as backward traversel
ListIterator<String> listIterator = names.listIterator();

//Iterates list in forward direction
while(listIterator.hasNext()){
    System.out.println(listIterator.next());
}

//Iterates list in backward direction once reaches the last element from above iterator in forward
direction
while(listIterator.hasPrevious()){
    System.out.println(listIterator.previous());
}
```

Iterating over Set

```
Set<String> names = new HashSet<>(Arrays.asList("Clementine", "Duran", "Mike"));
```

Version ≥ Java SE 8

```
names.forEach(System.out::println);
```

Version ≥ Java SE 5

```
for (Iterator<String> iterator = names.iterator(); iterator.hasNext(); ) {
    System.out.println(iterator.next());
}

for (String name : names) {
    System.out.println(name);
}
```

Version < Java SE 5

```
Iterator iterator = names.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

Iterating over Map

```
Map<Integer, String> names = new HashMap<>();
names.put(1, "Clementine");
names.put(2, "Duran");
names.put(3, "Mike");
```

Version ≥ Java SE 8

```
names.forEach((key, value) -> System.out.println("Key: " + key + " Value: " + value));
```

Version ≥ Java SE 5

```
for (Map.Entry<Integer, String> entry : names.entrySet()) {
    System.out.println(entry.getKey());
    System.out.println(entry.getValue());
}

// Iterating over only keys
for (Integer key : names.keySet()) {
    System.out.println(key);
}
// Iterating over only values
for (String value : names.values()) {
    System.out.println(value);
}
```

Version < Java SE 5

```
Iterator entries = names.entrySet().iterator();
while (entries.hasNext()) {
    Map.Entry entry = (Map.Entry) entries.next();
    System.out.println(entry.getKey());
    System.out.println(entry.getValue());
```

```
}
```

Section 24.5: Immutable Empty Collections

Sometimes it is appropriate to use an immutable empty collection. The `Collections` class provides methods to get such collections in an efficient way:

```
List<String> anEmptyList = Collections.emptyList();
Map<Integer, Date> anEmptyMap = Collections.emptyMap();
Set<Number> anEmptySet = Collections.emptySet();
```

These methods are generic and will automatically convert the returned collection to the type it is assigned to. That is, an invocation of e.g. `emptyList()` can be assigned to any type of `List` and likewise for `emptySet()` and `emptyMap()`.

The collections returned by these methods are immutable in that they will throw `UnsupportedOperationException` if you attempt to call methods which would change their contents (`add`, `put`, etc.). These collections are primarily useful as substitutes for empty method results or other default values, instead of using **`null`** or creating objects with **`new`**.

Section 24.6: Sub Collections

List subList(int fromIndex, int toIndex)

Here fromIndex is inclusive and toIndex is exclusive.

```
List list = new ArrayList();
List list1 = list.subList(fromIndex,toIndex);
```

1. If the list doesn't exist in the give range, it throws IndexOutofBoundException.
2. What ever changes made on the list1 will impact the same changes in the list.This is called backed collections.
3. If the fromnIndex is greater than the toIndex (fromIndex > toIndex) it throws IllegalArgumentException.

Example:

```
List<String> list = new ArrayList<String>();
List<String> list = new ArrayList<String>();
list.add("Hello1");
list.add("Hello2");
System.out.println("Before Sublist "+list);
List<String> list2 = list.subList(0, 1);
list2.add("Hello3");
System.out.println("After sublist changes "+list);
```

> Output:
> Before Sublist [Hello1, Hello2]
> After sublist changes [Hello1, Hello3, Hello2]

Set subSet(fromIndex,toIndex)

Here fromIndex is inclusive and toIndex is exclusive.

```
Set set = new TreeSet();
```

```
Set set1 = set.subSet(fromIndex,toIndex);
```

The returned set will throw an IllegalArgumentException on an attempt to insert an element outside its range.

Map subMap(fromKey,toKey)

fromKey is inclusive and toKey is exclusive

```
Map map = new TreeMap();
Map map1 = map.get(fromKey,toKey);
```

If fromKey is greater than toKey or if this map itself has a restricted range, and fromKey or toKey lies outside the bounds of the range then it throws IllegalArgumentException.

All the collections support backed collections means changes made on the sub collection will have same change on the main collection.

Section 24.7: Unmodifiable Collection

Sometimes it's not a good practice expose an internal collection since it can lead to a malicious code vulnerability due to it's mutable characteristic. In order to provide "read-only" collections java provides its unmodifiable versions.

An unmodifiable collection is often a copy of a modifiable collection which guarantees that the collection itself cannot be altered. Attempts to modify it will result in an UnsupportedOperationException exception.

It is important to notice that objects which are present inside the collection can still be altered.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class MyPojoClass {
    private List<Integer> intList = new ArrayList<>();

    public void addValueToIntList(Integer value){
        intList.add(value);
    }

    public List<Integer> getIntList() {
        return Collections.unmodifiableList(intList);
    }
}
```

The following attempt to modify an unmodifiable collection will throw an exception:

```
import java.util.List;

public class App {

    public static void main(String[] args) {
        MyPojoClass pojo = new MyPojoClass();
        pojo.addValueToIntList(42);

        List<Integer> list = pojo.getIntList();
        list.add(69);
    }
}
```

output:

```
Exception in thread "main" java.lang.UnsupportedOperationException
at java.util.Collections$UnmodifiableCollection.add(Collections.java:1055)
at App.main(App.java:12)
```

Section 24.8: Pitfall: concurrent modification exceptions

This exception occurs when a collection is modified while iterating over it using methods other than those provided by the iterator object. For example, we have a list of hats and we want to remove all those that have ear flaps:

```
List<IHat> hats = new ArrayList<>();
hats.add(new Ushanka()); // that one has ear flaps
hats.add(new Fedora());
hats.add(new Sombrero());
for (IHat hat : hats) {
    if (hat.hasEarFlaps()) {
        hats.remove(hat);
    }
}
```

If we run this code, ***ConcurrentModificationException*** will be raised since the code modifies the collection while iterating it. The same exception may occur if one of the multiple threads working with the same list is trying to modify the collection while others iterate over it. Concurrent modification of collections in multiple threads is a natural thing, but should be treated with usual tools from the concurrent programming toolbox such as synchronization locks, special collections adopted for concurrent modification, modifying the cloned collection from initial etc.

Section 24.9: Removing matching items from Lists using Iterator

Above I noticed an example to remove items from a List within a Loop and I thought of another example that may come in handy this time using the `Iterator` interface.
This is a demonstration of a trick that might come in handy when dealing with duplicate items in lists that you want to get rid of.

Note: This is only adding on to the **Removing items from a List within a loop** example:

So let's define our lists as usual

```
    String[] names = {"James","Smith","Sonny","Huckle","Berry","Finn","Allan"};
    List<String> nameList = new ArrayList<>();

    //Create a List from an Array
    nameList.addAll(Arrays.asList(names));

    String[] removeNames = {"Sonny","Huckle","Berry"};
    List<String> removeNameList = new ArrayList<>();

    //Create a List from an Array
    removeNameList.addAll(Arrays.asList(removeNames));
```

The following method takes in two Collection objects and performs the magic of removing the elements in our `removeNameList` that match with elements in `nameList`.

```
private static void removeNames(Collection<String> collection1, Collection<String> collection2) {
      //get Iterator.
    Iterator<String> iterator = collection1.iterator();

    //Loop while collection has items
    while(iterator.hasNext()){
        if (collection2.contains(iterator.next()))
            iterator.remove(); //remove the current Name or Item
    }
}
```

Calling the method and passing in the `nameList` and the `removeNameLista`s follows
`removeNames(nameList,removeNameList);`
Will produce the following output:

> Array List before removing names: **James Smith Sonny Huckle Berry Finn Allan**
> Array List after removing names: **James Smith Finn Allan**

A simple neat use for Collections that may come in handy to remove repeating elements within lists.

Section 24.10: Join lists

Following ways can be used for joining lists without modifying source list(s).

First approach. Has more lines but easy to understand

```
List<String> newList = new ArrayList<String>();
newList.addAll(listOne);
newList.addAll(listTwo);
```

Second approach. Has one less line but less readable.

```
List<String> newList = new ArrayList<String>(listOne);
newList.addAll(listTwo);
```

Third approach. Requires third party Apache commons-collections library.

```
ListUtils.union(listOne,listTwo);
```

Version ≥ Java SE 8

Using Streams the same can be achieved by

```
List<String> newList = Stream.concat(listOne.stream(),
listTwo.stream()).collect(Collectors.toList());
```

References. Interface List

Section 24.11: Creating your own Iterable structure for use with Iterator or for-each loop

To ensure that our collection can be iterated using iterator or for-each loop, we have to take care of following steps:

1. The stuff we want to iterate upon has to be `Iterable` and expose `iterator()`.
2. Design a `java.util.Iterator` by overriding `hasNext()`, `next()` and `remove()`.

I have added a simple generic linked list implementation below that uses above entities to make the linked list iterable.

```
package org.algorithms.linkedlist;

import java.util.Iterator;
import java.util.NoSuchElementException;


public class LinkedList<T> implements Iterable<T> {

  Node<T> head, current;

  private static class Node<T> {
    T data;
    Node<T> next;

    Node(T data) {
      this.data = data;
    }
  }

  public LinkedList(T data) {
    head = new Node<>(data);
  }

  public Iterator<T> iterator() {
    return new LinkedListIterator();
  }

  private class LinkedListIterator implements Iterator<T> {

    Node<T> node = head;

    @Override
    public boolean hasNext() {
      return node != null;
    }

    @Override
    public T next() {
      if (!hasNext())
        throw new NoSuchElementException();
      Node<T> prevNode = node;
      node = node.next;
      return prevNode.data;
    }

    @Override
    public void remove() {
      throw new UnsupportedOperationException("Removal logic not implemented.");
    }
  }

  public void add(T data) {
    Node current = head;
```

```
    while (current.next != null)
      current = current.next;
    current.next = new Node<>(data);
  }

}

class App {
  public static void main(String[] args) {

    LinkedList<Integer> list = new LinkedList<>(1);
    list.add(2);
    list.add(4);
    list.add(3);

    //Test #1
    System.out.println("using Iterator:");
    Iterator<Integer> itr = list.iterator();
    while (itr.hasNext()) {
      Integer i = itr.next();
      System.out.print(i + " ");
    }

    //Test #2
    System.out.println("\n\nusing for-each:");
    for (Integer data : list) {
      System.out.print(data + " ");
    }
  }
}
```

Output

```
using Iterator:
1 2 4 3
using for-each:
1 2 4 3
```

This will run in Java 7+. You can make it run on Java 5 and Java 6 also by substituting:

```
LinkedList<Integer> list = new LinkedList<>(1);
```

with

```
LinkedList<Integer> list = new LinkedList<Integer>(1);
```

or just any other version by incorporating the compatible changes.

Section 24.12: Collections and Primitive Values

Collections in Java only work for objects. I.e. there is no `Map<int, int>` in Java. Instead, primitive values need to be *boxed* into objects, as in `Map<Integer, Integer>`. Java auto-boxing will enable transparent use of these collections:

```
Map<Integer, Integer> map = new HashMap<>();
```

```
map.put(1, 17); // Automatic boxing of int to Integer objects
int a = map.get(1); // Automatic unboxing.
```

Unfortunately, the overhead of this is *substantial*. A `HashMap<Integer, Integer>` will require about 72 bytes per entry (e.g. on 64-bit JVM with compressed pointers, and assuming integers larger than 256, and assuming 50% load of the map). Because the actual data is only 8 bytes, this yields a massive overhead. Furthermore, it requires two level of indirection (Map -> Entry -> Value) it is unnecessarily slow.

There exist several libraries with optimized collections for primitive data types (that require only ~16 bytes per entry at 50% load, i.e. 4x less memory, and one level of indirection less), that can yield substantial performance benefits when using large collections of primitive values in Java.

Chapter 25: Lists

A *list* is an *ordered* collection of values. In Java, lists are part of the Java Collections Framework. Lists implement the `java.util.List` interface, which extends `java.util.Collection`.

Section 25.1: Sorting a generic list

The `Collections` class offers two standard static methods to sort a list:

- `sort(List<T> list)` applicable to lists where `T extends Comparable<? super T>`, and
- `sort(List<T> list, Comparator<? super T> c)` applicable to lists of any type.

Applying the former requires amending the class of list elements being sorted, which is not always possible. It might also be undesirable as although it provides the default sorting, other sorting orders may be required in different circumstances, or sorting is just a one off task.

Consider we have a task of sorting objects that are instances of the following class:

```
public class User {
    public final Long id;
    public final String username;

    public User(Long id, String username) {
        this.id = id;
        this.username = username;
    }

    @Override
    public String toString() {
        return String.format("%s:%d", username, id);
    }
}
```

In order to use `Collections.sort(List<User> list)` we need to modify the `User` class to implement the `Comparable` interface. For example

```
public class User implements Comparable<User> {
    public final Long id;
    public final String username;

    public User(Long id, String username) {
        this.id = id;
        this.username = username;
    }

    @Override
    public String toString() {
        return String.format("%s:%d", username, id);
    }

    @Override
    /** The natural ordering for 'User' objects is by the 'id' field. */
    public int compareTo(User o) {
        return id.compareTo(o.id);
    }
}
```

(Aside: many standard Java classes such as `String`, `Long`, `Integer` implement the `Comparable` interface. This makes lists of those elements sortable by default, and simplifies implementation of `compare` or `compareTo` in other classes.)

With the modification above, the we can easily sort a list of `User` objects based on the classes *natural ordering*. (In this case, we have defined that to be ordering based on `id` values). For example:

```
List<User> users = Lists.newArrayList(
    new User(33L, "A"),
    new User(25L, "B"),
    new User(28L, ""));
Collections.sort(users);

System.out.print(users);
// [B:25, C:28, A:33]
```

However, suppose that we wanted to sort `User` objects by `name` rather than by `id`. Alternatively, suppose that we had not been able to change the class to make it implement `Comparable`.

This is where the `sort` method with the `Comparator` argument is useful:

```
Collections.sort(users, new Comparator<User>() {
    @Override
    /* Order two 'User' objects based on their names. */
    public int compare(User left, User right) {
        return left.username.compareTo(right.username);
    }
});
System.out.print(users);
// [A:33, B:25, C:28]
```

Version ≥ Java SE 8

In Java 8 you can use a *lambda* instead of an anonymous class. The latter reduces to a one-liner:

```
Collections.sort(users, (l, r) -> l.username.compareTo(r.username));
```

Further, there Java 8 adds a default `sort` method on the `List` interface, which simplifies sorting even more.

```
users.sort((l, r) -> l.username.compareTo(r.username))
```

Section 25.2: Convert a list of integers to a list of strings

```
List<Integer> nums = Arrays.asList(1, 2, 3);
List<String> strings = nums.stream()
    .map(Object::toString)
    .collect(Collectors.toList());
```

That is:

1. Create a stream from the list
2. Map each element using `Object::toString`
3. Collect the `String` values into a `List` using `Collectors.toList()`

Section 25.3: Classes implementing List - Pros and Cons

The `List` interface is implemented by different classes. Each of them has its own way for implementing it with different strategies and providing different pros and cons.

Classes implementing List

These are all of the **`public`** classes in Java SE 8 that implement the `java.util.List` interface:

1. **Abstract Classes:**
    - AbstractList
    - AbstractSequentialList
2. **Concrete Classes:**
    - ArrayList
    - AttributeList
    - CopyOnWriteArrayList
    - LinkedList
    - RoleList
    - RoleUnresolvedList
    - Stack
    - Vector

Pros and Cons of each implementation in term of time complexity

ArrayList

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

ArrayList is a resizable-array implementation of the List interface. Storing the list into an array, *ArrayList* provides methods (in addition to the methods implementing the *List* interface) for manipulating the size of the array.

Initialize ArrayList of Integer with size 100

```
List<Integer> myList = new ArrayList<Integer>(100); // Constructs an empty list with the specified initial capacity.
```

- PROS:

The size, isEmpty, ***get***, ***set***, iterator, and listIterator operations run in constant time. So getting and setting each element of the List has the same *time cost*:

```
int e1 = myList.get(0);  //   \
int e2 = myList.get(10); //    | => All the same constant cost => O(1)
myList.set(2,10);        //   /
```

- CONS:

Being implemented with an array (static structure) adding elements over the size of the array has a big cost due to the fact that a new allocation need to be done for all the array. However, from documentation:

> The add operation runs in amortized constant time, that is, adding n elements requires O(n) time

Removing an element requires O(n) time.

AttributeList

On coming

CopyOnWriteArrayList

On coming

LinkedList

```
public class LinkedList<E>
extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable
```

LinkedList is implemented by a doubly-linked list a linked data structure that consists of a set of sequentially linked records called nodes.

Iitialize LinkedList of Integer

```
List<Integer> myList = new LinkedList<Integer>(); // Constructs an empty list.
```

- PROS:

Adding or removing an element to the front of the list or to the end has constant time.

```
myList.add(10);  // \
myList.add(0,2); //  | => constant time => O(1)
myList.remove(); // /
```

- CONS: From documentation:

> Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

Operations such as:

```
myList.get(10);    // \
myList.add(11,25); //  | => worst case done in O(n/2)
myList.set(15,35); // /
```

RoleList

On coming

RoleUnresolvedList

On coming

Stack

On coming

Vector

On coming

Section 25.4: Finding common elements between 2 lists

Suppose you have two lists: A and B, and you need to find the elements that exist in both lists.

You can do it by just invoking the method `List.retainAll()`.

Example:

```
public static void main(String[] args) {
    List<Integer> numbersA = new ArrayList<>();
    List<Integer> numbersB = new ArrayList<>();
    numbersA.addAll(Arrays.asList(new Integer[] { 1, 3, 4, 7, 5, 2 }));
    numbersB.addAll(Arrays.asList(new Integer[] { 13, 32, 533, 3, 4, 2 }));

    System.out.println("A: " + numbersA);
    System.out.println("B: " + numbersB);
    List<Integer> numbersC = new ArrayList<>();
    numbersC.addAll(numbersA);
    numbersC.retainAll(numbersB);

    System.out.println("List A : " + numbersA);
    System.out.println("List B : " + numbersB);
    System.out.println("Common elements between A and B: " + numbersC);

}
```

Section 25.5: In-place replacement of a List element

This example is about replacing a `List` element while ensuring that the replacement element is at the same position as the element that is replaced.

This can be done using these methods:

- set(int index, T type)
- int indexOf(T type)

Consider an `ArrayList` containing the elements "Program starting!", "Hello world!" and "Goodbye world!"

```
List<String> strings = new ArrayList<String>();
strings.add("Program starting!");
strings.add("Hello world!");
strings.add("Goodbye world!");
```

If we know the index of the element we want to replace, we can simply use `set` as follows:

```
strings.set(1, "Hi world");
```

If we don't know the index, we can search for it first. For example:

```
int pos = strings.indexOf("Goodbye world!");
if (pos >= 0) {
```

```
    strings.set(pos, "Goodbye cruel world!");
}
```

Notes:

1. The `set` operation will not cause a `ConcurrentModificationException`.
2. The `set` operation is fast (`O(1)`) for `ArrayList` but slow (`O(N)`) for a `LinkedList`.
3. An `indexOf` search on an `ArrayList` or `LinkedList` is slow (`O(N)`).

Section 25.6: Making a list unmodifiable

The Collections class provides a way to make a list unmodifiable:

```
List<String> ls = new ArrayList<String>();
List<String> unmodifiableList = Collections.unmodifiableList(ls);
```

If you want an unmodifiable list with one item you can use:

```
List<String> unmodifiableList = Collections.singletonList("Only string in the list");
```

Section 25.7: Moving objects around in the list

The Collections class allows for you to move objects around in the list using various methods (ls is the List):

Reversing a list:

```
Collections.reverse(ls);
```

Rotating positions of elements in a list

The rotate method requires an integer argument. This is how many spots to move it along the line by. An example of this is below:

```
List<String> ls = new ArrayList<String>();
ls.add(" how");
ls.add(" are");
ls.add(" you?");
ls.add("hello,");
Collections.rotate(ls, 1);

for(String line : ls) System.out.print(line);
System.out.println();
```

This will print "hello, how are you?"

Shuffling elements around in a list

Using the same list above, we can shuffle the elements in a list:

```
Collections.shuffle(ls);
```

We can also give it a java.util.Random object that it uses to randomly place objects in spots:

```
Random random = new Random(12);
Collections.shuffle(ls, random);
```

Section 25.8: Creating, Adding and Removing element from an ArrayList

`ArrayList` is one of the inbuilt data structures in Java. It is a dynamic array (where the size of the data structure not needed to be declared first) for storing elements (Objects).

It extends `AbstractList` class and implements `List` interface. An `ArrayList` can contain duplicate elements where it maintains insertion order. It should be noted that the class `ArrayList` is non-synchronized, so care should be taken when handling concurrency with `ArrayList`. `ArrayList` allows random access because array works at the index basis. Manipulation is slow in `ArrayList` because of shifting that often occurs when an element is removed from the array list.

An `ArrayList` can be created as follows:

```
List<T> myArrayList = new ArrayList<>();
```

Where `T` (Generics) is the type that will be stored inside `ArrayList`.

The type of the `ArrayList` can be any Object. The type can't be a primitive type (use their *wrapper classes* instead).

To add an element to the `ArrayList`, use `add()` method:

```
myArrayList.add(element);
```

Or to add item to a certain index:

```
myArrayList.add(index, element); //index of the element should be an int (starting from 0)
```

To remove an item from the `ArrayList`, use the `remove()` method:

```
myArrayList.remove(element);
```

Or to remove an item from a certain index:

```
myArrayList.remove(index); //index of the element should be an int (starting from 0)
```

Section 25.9: Creating a List

Giving your list a type

To create a list you need a type (any class, e.g. `String`). This is the type of your `List`. The `List` will only store objects of the specified type. For example:

```
List<String> strings;
```

Can store `"string1"`, `"hello world!"`, `"goodbye"`, etc, but it can't store `9.2`, however:

```
List<Double> doubles;
```

Can store `9.2`, but not `"hello world!"`.

Initialising your list

If you try to add something to the lists above you will get a NullPointerException, because `strings` and `doubles`

both equal **null**!

There are two ways to initialise a list:

Option 1: Use a class that implements List

`List` is an interface, which means that does not have a constructor, rather methods that a class must override. `ArrayList` is the most commonly used `List`, though `LinkedList` is also common. So we initialise our list like this:

```
List<String> strings = new ArrayList<String>();
```

or

```
List<String> strings = new LinkedList<String>();
```

`Version ≥ Java SE 7`

Starting from Java SE 7, you can use a *diamond operator*:

```
List<String> strings = new ArrayList<>();
```

or

```
List<String> strings = new LinkedList<>();
```

Option 2: Use the Collections class

The `Collections` class provides two useful methods for creating Lists without a `List` variable:

- `emptyList()`: returns an empty list.
- `singletonList(T)`: creates a list of type T and adds the element specified.

And a method which uses an existing `List` to fill data in:

- `addAll(L, T...)`: adds all the specified elements to the list passed as the first parameter.

Examples:

import java.util.List; import java.util.Collections; List<Integer> l = Collections.emptyList(); List<Integer> l1 = Collections.singletonList(42); Collections.addAll(l1, 1, 2, 3);

Section 25.10: Positional Access Operations

The List API has eight methods for positional access operations:

- `add(T type)`
- `add(int index, T type)`
- `remove(Object o)`
- `remove(int index)`
- `get(int index)`
- `set(int index, E element)`
- `int indexOf(Object o)`
- `int lastIndexOf(Object o)`

So, if we have a List:

```
List<String> strings = new ArrayList<String>();
```

And we wanted to add the strings "Hello world!" and "Goodbye world!" to it, we would do it as such:

```
strings.add("Hello world!");
strings.add("Goodbye world!");
```

And our list would contain the two elements. Now lets say we wanted to add "Program starting!" at the **front** of the list. We would do this like this:

```
strings.add(0, "Program starting!");
```

NOTE: The first element is 0.

Now, if we wanted to remove the "Goodbye world!" line, we could do it like this:

```
strings.remove("Goodbye world!");
```

And if we wanted to remove the first line (which in this case would be "Program starting!", we could do it like this:

```
strings.remove(0);
```

Note:

1. Adding and removing list elements modify the list, and this can lead to a `ConcurrentModificationException` if the list is being iterated concurrently.
2. Adding and removing elements can be `O(1)` or `O(N)` depending on the list class, the method used, and whether you are adding / removing an element at the start, the end, or in the middle of the list.

In order to retrieve an element of the list at a specified position you can use the `E get(int index);` method of the List API. For example:

```
strings.get(0);
```

will return the first element of the list.

You can replace any element at a specified position by using the `set(int index, E element);`. For example:

```
strings.set(0,"This is a replacement");
```

This will set the String "This is a replacement" as the first element of the list.

Note: The set method will overwrite the element at the position 0. It will not add the new String at the position 0 and push the old one to the position 1.

The `int indexOf(Object o);` returns the position of the first occurrence of the object passed as argument. If there are no occurrences of the object in the list then the -1 value is returned. In continuation of the previous example if you call:

```
strings.indexOf("This is a replacement")
```

the 0 is expected to be returned as we set the String "This is a replacement" in the position 0 of our list. In case where there are more than one occurrence in the list when `int indexOf(Object o);` is called then as mentioned

the index of the first occurrence will be returned. By calling the `int lastIndexOf(Object o)` you can retrieve the index of the last occurrence in the list. So if we add another "This is a replacement":

```
strings.add("This is a replacement");
strings.lastIndexOf("This is a replacement");
```

This time the 1 will be returned and not the 0;

Section 25.11: Iterating over elements in a list

For the example, lets say that we have a List of type String that contains four elements: "hello, ", "how ", "are ", "you?"

The best way to iterate over each element is by using a for-each loop:

```
public void printEachElement(List<String> list){
    for(String s : list){
        System.out.println(s);
    }
}
```

Which would print:

```
hello,
how
are
you?
```

To print them all in the same line, you can use a StringBuilder:

```
public void printAsLine(List<String> list){
    StringBuilder builder = new StringBuilder();
    for(String s : list){
        builder.append(s);
    }
    System.out.println(builder.toString());
}
```

Will print:

```
hello, how are you?
```

Alternatively, you can use element indexing (as described in Accessing element at ith Index from ArrayList) to iterate a list. Warning: this approach is inefficient for linked lists.

Section 25.12: Removing elements from list B that are present in the list A

Lets suppose you have 2 Lists A and B, and you want to remove from **B** all the elements that you have in **A** the method in this case is

```
List.removeAll(Collection c);
```

#Example:

```java
public static void main(String[] args) {
    List<Integer> numbersA = new ArrayList<>();
    List<Integer> numbersB = new ArrayList<>();
    numbersA.addAll(Arrays.asList(new Integer[] { 1, 3, 4, 7, 5, 2 }));
    numbersB.addAll(Arrays.asList(new Integer[] { 13, 32, 533, 3, 4, 2 }));
    System.out.println("A: " + numbersA);
    System.out.println("B: " + numbersB);

    numbersB.removeAll(numbersA);
    System.out.println("B cleared: " + numbersB);
    }
```

this will print

> A: [1, 3, 4, 7, 5, 2]
>
> B: [13, 32, 533, 3, 4, 2]
>
> B cleared: [13, 32, 533]

Chapter 26: Sets

Section 26.1: Initialization

A Set is a Collection that cannot contain duplicate elements. It models the mathematical set abstraction.

`Set` have its implementation in various classes like `HashSet`, `TreeSet`, `LinkedHashSet`.

For example:

HashSet:

```
Set<T> set = new HashSet<T>();
```

Here `T` can be `String`, `Integer` or any other **object**. **HashSet** allows for quick lookup of O(1) but does not sort the data added to it and loses the insertion order of items.

TreeSet:

It stores data in a sorted manner sacrificing some speed for basic operations which take O(lg(n)). It does not maintain the insertion order of items.

```
TreeSet<T> sortedSet = new TreeSet<T>();
```

LinkedHashSet:

It is a linked list implementation of `HashSet` Once can iterate over the items in the order they were added. Sorting is not provided for its contents. O(1) basic operations are provided, however there is higher cost than `HashSet` in maintaining the backing linked list.

```
LinkedHashSet<T> linkedhashset = new LinkedHashSet<T>();
```

Section 26.2: Basics of Set

What is a Set?

A set is a data structure which contains a set of elements with an important property that no two elements in the set are equal.

Types of Set:

1. **HashSet:** A set backed by a hash table (actually a HashMap instance)
2. **Linked HashSet:** A Set backed by Hash table and linked list, with predictable iteration order
3. **TreeSet:** A NavigableSet implementation based on a TreeMap.

Creating a set

```
Set<Integer> set = new HashSet<Integer>(); // Creates an empty Set of Integers

Set<Integer> linkedHashSet = new LinkedHashSet<Integer>(); //Creates a empty Set of Integers, with
predictable iteration order
```

Adding elements to a Set

Elements can be added to a set using the `add()` method

```
set.add(12); //  - Adds element 12 to the set
set.add(13); //  - Adds element 13 to the set
```

Our set after executing this method:

```
set = [12,13]
```

Delete all the elements of a Set

```
set.clear();  //Removes all objects from the collection.
```

After this set will be:

```
set = []
```

Check whether an element is part of the Set

Existence of an element in the set can be checked using the `contains()` method

```
set.contains(0);  //Returns true if a specified object is an element within the set.
```

Output: `False`

Check whether a Set is empty

`isEmpty()` method can be used to check whether a Set is empty.

```
set.isEmpty();  //Returns true if the set has no elements
```

Output: True

Remove an element from the Set

```
set.remove(0); // Removes first occurrence of a specified object from the collection
```

Check the Size of the Set

```
set.size(); //Returns the number of elements in the collection
```

Output: 0

Section 26.3: Types and Usage of Sets

Generally, sets are a type of collection which stores unique values. Uniqueness is determined by the `equals()` and `hashCode()` methods.

Sorting is determined by the type of set.

`HashSet` - Random Sorting

`Version ≥ Java SE 7`

```
Set<String> set = new HashSet<> ();
set.add("Banana");
```

```
set.add("Banana");
set.add("Apple");
set.add("Strawberry");

// Set Elements: ["Strawberry", "Banana", "Apple"]
```

`LinkedHashSet` - Insertion Order

Version ≥ Java SE 7

```
Set<String> set = new LinkedHashSet<> ();
set.add("Banana");
set.add("Banana");
set.add("Apple");
set.add("Strawberry");

// Set Elements: ["Banana", "Apple", "Strawberry"]
```

`TreeSet` - By `compareTo()` or `Comparator`

Version ≥ Java SE 7

```
Set<String> set = new TreeSet<> ();
set.add("Banana");
set.add("Banana");
set.add("Apple");
set.add("Strawberry");

// Set Elements: ["Apple", "Banana", "Strawberry"]
```

Version ≥ Java SE 7

```
Set<String> set = new TreeSet<> ((string1, string2) -> string2.compareTo(string1));
set.add("Banana");
set.add("Banana");
set.add("Apple");
set.add("Strawberry");

// Set Elements: ["Strawberry", "Banana", "Apple"]
```

Section 26.4: Create a list from an existing Set

Using a new List

```
List<String> list = new ArrayList<String>(listOfElements);
```

Using List.addAll() method

```
    Set<String> set = new HashSet<String>();
    set.add("foo");
    set.add("boo");

    List<String> list = new ArrayList<String>();
    list.addAll(set);
```

Using Java 8 Steam API

```
List<String> list = set.stream().collect(Collectors.toList());
```

Section 26.5: Eliminating duplicates using Set

Suppose you have a collection `elements`, and you want to create another collection containing the same elements but with all **duplicates eliminated**:

```
Collection<Type> noDuplicates = new HashSet<Type>(elements);
```

Example:

```
List<String> names = new ArrayList<>(
        Arrays.asList("John", "Marco", "Jenny", "Emily", "Jenny", "Emily", "John"));
Set<String> noDuplicates = new HashSet<>(names);
System.out.println("noDuplicates = " + noDuplicates);
```

Output:

```
noDuplicates = [Marco, Emily, John, Jenny]
```

Section 26.6: Declaring a HashSet with values

You can create a new class that inherits from HashSet:

```
Set<String> h = new HashSet<String>() {{
    add("a");
    add("b");
}};
```

One line solution:

```
Set<String> h = new HashSet<String>(Arrays.asList("a", "b"));
```

Using guava:

```
Sets.newHashSet("a", "b", "c")
```

Using Streams:

```
Set<String> set3 = Stream.of("a", "b", "c").collect(toSet());
```

Chapter 27: List vs Set

What are differences between List and Set collection at the top level and How to choose when to use List in java and when to use Set in Java

Section 27.1: List vs Set

```
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
public class SetAndListExample
{
    public static void main( String[] args )
    {
        System.out.println("List example .....");
        List list = new ArrayList();
        list.add("1");
        list.add("2");
        list.add("3");
        list.add("4");
        list.add("1");
        for (String temp : list){
            System.out.println(temp);
        }

        System.out.println("Set example .....");
        Set<String> set = new HashSet<String>();
        set.add("1");
        set.add("2");
        set.add("3");
        set.add("4");
        set.add("1");
        set.add("2");
        set.add("5");

        for (String temp : set){
            System.out.println(temp);
        }
    }

}
```

Output List example ..... 1 2 3 4 1 Set example 3 2 10 5 4

Chapter 28: Maps

The java.util.Map interface represents a mapping between keys and their values. A map cannot contain duplicate keys; and each key can map to at most one value.

Since `Map` is an interface, then you need to instantiate a concrete implementation of that interface in order to use it; there are several `Map` implementations, and mostly used are the `java.util.HashMap` and `java.util.TreeMap`

Section 28.1: Iterating Map Entries Efficiently

This section provides code and benchmarks for ten unique example implementations which iterate over the entries of a `Map<Integer, Integer>` and generate the sum of the `Integer` values. All of the examples have an algorithmic complexity of `Θ(n)`, however, the benchmarks are still useful for providing insight on which implementations are more efficient in a "real world" environment.

1. Implementation using **`Iterator`** with **`Map.Entry`**

```
Iterator<Map.Entry<Integer, Integer>> it = map.entrySet().iterator();
while (it.hasNext()) {
    Map.Entry<Integer, Integer> pair = it.next();
    sum += pair.getKey() + pair.getValue();
}
```

2. Implementation using **`for`** with **`Map.Entry`**

```
for (Map.Entry<Integer, Integer> pair : map.entrySet()) {
    sum += pair.getKey() + pair.getValue();
}
```

3. Implementation using **`Map.forEach`** (Java 8+)

```
map.forEach((k, v) -> sum[0] += k + v);
```

4. Implementation using **`Map.keySet`** with **`for`**

```
for (Integer key : map.keySet()) {
    sum += key + map.get(key);
}
```

5. Implementation using **`Map.keySet`** with **`Iterator`**

```
Iterator<Integer> it = map.keySet().iterator();
while (it.hasNext()) {
    Integer key = it.next();
    sum += key + map.get(key);
}
```

6. Implementation using **`for`** with **`Iterator`** and **`Map.Entry`**

```
for (Iterator<Map.Entry<Integer, Integer>> entries =
         map.entrySet().iterator(); entries.hasNext(); ) {
    Map.Entry<Integer, Integer> entry = entries.next();
    sum += entry.getKey() + entry.getValue();
}
```

7. Implementation using **`Stream.forEach`** (Java 8+)

```
map.entrySet().stream().forEach(e -> sum += e.getKey() + e.getValue());
```

8. Implementation using **Stream.forEach** with **Stream.parallel** (Java 8+)

```
map.entrySet()
   .stream()
   .parallel()
   .forEach(e -> sum += e.getKey() + e.getValue());
```

9. Implementation using **IterableMap** from Apache Collections

```
MapIterator<Integer, Integer> mit = iterableMap.mapIterator();
while (mit.hasNext()) {
    sum += mit.next() + it.getValue();
}
```

10. Implementation using **MutableMap** from Eclipse Collections

```
mutableMap.forEachKeyValue((key, value) -> {
    sum += key + value;
});
```

Performance Tests (*Code available on Github*)
Test Environment: Windows 8.1 64-bit, Intel i7-4790 3.60GHz, 16 GB

1. Average Performance of 10 Trials (100 elements) Best: 308±21 ns/op

```
Benchmark                           Score     Error  Units
test3_UsingForEachAndJava8            308  ±     21  ns/op
test10_UsingEclipseMutableMap         309  ±      9  ns/op
test1_UsingWhileAndMapEntry           380  ±     14  ns/op
test6_UsingForAndIterator             387  ±     16  ns/op
test2_UsingForEachAndMapEntry         391  ±     23  ns/op
test7_UsingJava8StreamAPI             510  ±     14  ns/op
test9_UsingApacheIterableMap          524  ±      8  ns/op
test4_UsingKeySetAndForEach           816  ±     26  ns/op
test5_UsingKeySetAndIterator          863  ±     25  ns/op
test8_UsingJava8StreamAPIParallel    5552  ±    185  ns/op
```

2. Average Performance of 10 Trials (10000 elements) Best: 37.606±0.790 µs/op

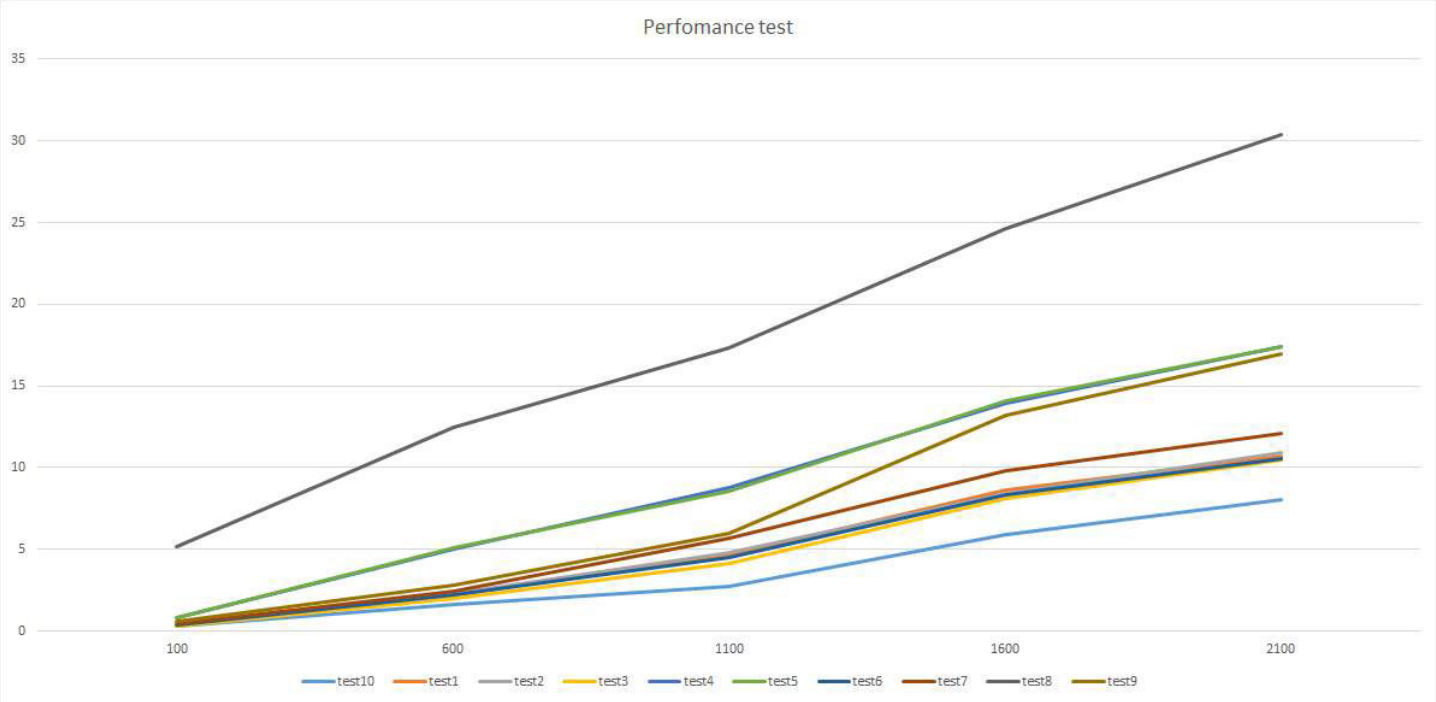
```
Benchmark                           Score     Error  Units
test10_UsingEclipseMutableMap       37606  ±    790  ns/op
test3_UsingForEachAndJava8          50368  ±    887  ns/op
test6_UsingForAndIterator           50332  ±    507  ns/op
test2_UsingForEachAndMapEntry       51406  ±   1032  ns/op
test1_UsingWhileAndMapEntry         52538  ±   2431  ns/op
test7_UsingJava8StreamAPI           54464  ±    712  ns/op
test4_UsingKeySetAndForEach         79016  ±  25345  ns/op
test5_UsingKeySetAndIterator        91105  ±  10220  ns/op
test8_UsingJava8StreamAPIParallel  112511  ±    365  ns/op
test9_UsingApacheIterableMap       125714  ±   1935  ns/op
```

3. Average Performance of 10 Trials (100000 elements) Best: 1184.767±332.968 µs/op

```
Benchmark                           Score       Error  Units
```

```
test1_UsingWhileAndMapEntry       1184.767  ±  332.968  µs/op
test10_UsingEclipseMutableMap     1191.735  ±  304.273  µs/op
test2_UsingForEachAndMapEntry     1205.815  ±  366.043  µs/op
test6_UsingForAndIterator         1206.873  ±  367.272  µs/op
test8_UsingJava8StreamAPIParallel 1485.895  ±  233.143  µs/op
test5_UsingKeySetAndIterator      1540.281  ±  357.497  µs/op
test4_UsingKeySetAndForEach       1593.342  ±  294.417  µs/op
test3_UsingForEachAndJava8        1666.296  ±  126.443  µs/op
test7_UsingJava8StreamAPI         1706.676  ±  436.867  µs/op
test9_UsingApacheIterableMap      3289.866  ± 1445.564  µs/op
```

4. A Comparison of Performance Variations Respective to Map Size




```
    x: Size of Map
 f(x): Benchmark Score (µs/op)
```

Tests f(x)	100	600	1100	1600	2100
10	0.333	1.631	2.752	5.937	8.024
3	0.309	1.971	4.147	8.147	10.473
6	0.372	2.190	4.470	8.322	10.531
1	0.405	2.237	4.616	8.645	10.707
2	0.376	2.267	4.809	8.403	10.910
7	0.473	2.448	5.668	9.790	12.125
9	0.565	2.830	5.952	13.22	16.965
4	0.808	5.012	8.813	13.939	17.407
5	0.81	5.104	8.533	14.064	17.422
8	5.173	12.499	17.351	24.671	30.403

Section 28.2: Usage of HashMap

HashMap is an implementation of the Map interface that provides a Data Structure to store data in Key-Value pairs.

1. Declaring HashMap

```
Map<KeyType, ValueType> myMap = new HashMap<KeyType, ValueType>();
```

KeyType and ValueType must be valid types in Java, such as - String, Integer, Float or any custom class like Employee, Student etc..

For Example : `Map<String,Integer> myMap = new HashMap<String,Integer>();`

2. Putting values in HashMap.

To put a value in the HashMap, we have to call `put` method on the HashMap object by passing the Key and the Value as parameters.

```
myMap.put("key1", 1);
myMap.put("key2", 2);
```

If you call the put method with the Key that already exists in the Map, the method will override its value and return the old value.

3. Getting values from HashMap.

For getting the value from a HashMap you have to call the `get` method, by passing the Key as a parameter.

```
myMap.get("key1");    //return 1 (class Integer)
```

If you pass a key that does not exists in the HashMap, this method will return **`null`**

4. Check whether the Key is in the Map or not.

```
myMap.containsKey(varKey);
```

5. Check whether the Value is in the Map or not.

```
myMap.containsValue(varValue);
```

The above methods will return a **`boolean`** value true or false if key, value exists in the Map or not.

Section 28.3: Using Default Methods of Map from Java 8

Examples of using Default Methods introduced in Java 8 in Map interface

1. Using **getOrDefault**

Returns the value mapped to the key, or if the key is not present, returns the default value

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "First element");
map.get(1);                                 // => First element
map.get(2);                                 // => null
map.getOrDefault(2, "Default element");     // => Default element
```

2. Using **forEach**

Allows to perform the operation specified in the 'action' on each Map Entry

```
Map<Integer, String> map = new HashMap<Integer, String>();
map.put(1, "one");
map.put(2, "two");
map.put(3, "three");
```

```
map.forEach((key, value) -> System.out.println("Key: "+key+ " :: Value: "+value));

 // Key: 1 :: Value: one
 // Key: 2 :: Value: two
 // Key: 3 :: Value: three
```

3. Using **replaceAll**

Will replace with new-value only if key is present

```
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("john", 20);
map.put("paul", 30);
map.put("peter", 40);
map.replaceAll((key,value)->value+10);   //{john=30, paul=40, peter=50}
```

4. Using **putIfAbsent**

Key-Value pair is added to the map, if the key is not present or mapped to null

```
    Map<String, Integer> map = new HashMap<String, Integer>();
    map.put("john", 20);
    map.put("paul", 30);
    map.put("peter", 40);
    map.putIfAbsent("kelly", 50);     //{john=20, paul=30, peter=40, kelly=50}
```

5. Using **remove**

   Removes the key only if its associated with the given value

   ```
   Map<String, Integer> map = new HashMap<String, Integer>();
   map.put("john", 20);
   map.put("paul", 30);
   map.put("peter", 40);
   map.remove("peter",40); //{john=30, paul=40}
   ```

6. Using **replace**

   If the key is present then the value is replaced by new-value. If the key is not present, does nothing.

```
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("john", 20);
map.put("paul", 30);
map.put("peter", 40);
map.replace("peter",50); //{john=20, paul=30, peter=50}
map.replace("jack",60); //{john=20, paul=30, peter=50}
```

7. Using **computeIfAbsent**

This method adds an entry in the Map. the key is specified in the function and the value is the result of the application of the mapping function

```
    Map<String, Integer> map = new HashMap<String, Integer>();
    map.put("john", 20);
    map.put("paul", 30);
    map.put("peter", 40);
```

```
    map.computeIfAbsent("kelly", k->map.get("john")+10); //{john=20, paul=30, peter=40, kelly=30}
    map.computeIfAbsent("peter", k->map.get("john")+10); //{john=20, paul=30, peter=40, kelly=30}
//peter already present
```

8. Using **computeIfPresent**

This method adds an entry or modifies an existing entry in the Map. Does nothing if an entry with that key is not present

```
    Map<String, Integer> map = new HashMap<String, Integer>();
    map.put("john", 20);
    map.put("paul", 30);
    map.put("peter", 40);
    map.computeIfPresent("kelly", (k,v)->v+10); //{john=20, paul=30, peter=40} //kelly not present
    map.computeIfPresent("peter", (k,v)->v+10); //{john=20, paul=30, peter=50} // peter present, so
increase the value
```

9. Using **compute**

This method replaces the value of a key by the newly computed value

```
    Map<String, Integer> map = new HashMap<String, Integer>();
    map.put("john", 20);
    map.put("paul", 30);
    map.put("peter", 40);
    map.compute("peter", (k,v)->v+50); //{john=20, paul=30, peter=90} //Increase the value
```

10. Using **merge**

Adds the key-value pair to the map, if key is not present or value for the key is null Replaces the value with the newly computed value, if the key is present Key is removed from the map , if new value computed is null

```
    Map<String, Integer> map = new HashMap<String, Integer>();
    map.put("john", 20);
    map.put("paul", 30);
    map.put("peter", 40);

    //Adds the key-value pair to the map, if key is not present or value for the key is null
    map.merge("kelly", 50 , (k,v)->map.get("john")+10); // {john=20, paul=30, peter=40, kelly=50}

    //Replaces the value with the newly computed value, if the key is present
    map.merge("peter", 50 , (k,v)->map.get("john")+10); //{john=20, paul=30, peter=30, kelly=50}

    //Key is removed from the map , if new value computed is null
    map.merge("peter", 30 , (k,v)->map.get("nancy")); //{john=20, paul=30, kelly=50}
```

Section 28.4: Iterating through the contents of a Map

Maps provide methods which let you access the keys, values, or key-value pairs of the map as collections. You can iterate through these collections. Given the following map for example:

```
Map<String, Integer> repMap = new HashMap<>();
repMap.put("Jon Skeet", 927_654);
repMap.put("BalusC", 708_826);
repMap.put("Darin Dimitrov", 715_567);
```

Iterating through map keys:

```
for (String key : repMap.keySet()) {
    System.out.println(key);
}
```

Prints:

> Darin Dimitrov
> Jon Skeet
> BalusC

`keySet()` provides the keys of the map as a `Set`. `Set` is used as the keys cannot contain duplicate values. Iterating through the set yields each key in turn. HashMaps are not ordered, so in this example the keys may be returned in any order.

Iterating through map values:

```
for (Integer value : repMap.values()) {
    System.out.println(value);
}
```

Prints:

> 715567
> 927654
> 708826

`values()` returns the values of the map as a `Collection`. Iterating through the collection yields each value in turn. Again, the values may be returned in any order.

Iterating through keys and values together

```
for (Map.Entry<String, Integer> entry : repMap.entrySet()) {
    System.out.printf("%s = %d\n", entry.getKey(), entry.getValue());
}
```

Prints:

> Darin Dimitrov = 715567
> Jon Skeet = 927654
> BalusC = 708826

`entrySet()` returns a collection of `Map.Entry` objects. Map.Entry gives access to the key and value for each entry.

Section 28.5: Merging, combine and composing Maps

Use `putAll` to put every member of one map into another. Keys already present in the map will have their corresponding values overwritten.

```
Map<String, Integer> numbers = new HashMap<>();
numbers.put("One", 1)
```

```
numbers.put("Three", 3)
Map<String, Integer> other_numbers = new HashMap<>();
other_numbers.put("Two", 2)
other_numbers.put("Three", 4)

numbers.putAll(other_numbers)
```

This yields the following mapping in `numbers`:

```
"One" -> 1
"Two" -> 2
"Three" -> 4 //old value 3 was overwritten by new value 4
```

If you want to combine values instead of overwriting them, you can use `Map.merge`, added in Java 8, which uses a user-provided `BiFunction` to merge values for duplicate keys. `merge` operates on individual keys and values, so you'll need to use a loop or `Map.forEach`. Here we concatenate strings for duplicate keys:

```
for (Map.Entry<String, Integer> e : other_numbers.entrySet())
    numbers.merge(e.getKey(), e.getValue(), Integer::sum);
//or instead of the above loop
other_numbers.forEach((k, v) -> numbers.merge(k, v, Integer::sum));
```

If you want to enforce the constraint there are no duplicate keys, you can use a merge function that throws an `AssertionError`:

```
mapA.forEach((k, v) ->
    mapB.merge(k, v, (v1, v2) ->
        {throw new AssertionError("duplicate values for key: "+k);}));
```

Composing Map<X,Y> and Map<Y,Z> to get Map<X,Z>

If you want to compose two mappings, you can do it as follows

```
    Map<String, Integer> map1 = new HashMap<String, Integer>();
    map1.put("key1", 1);
    map1.put("key2", 2);
    map1.put("key3", 3);

    Map<Integer, Double> map2 = new HashMap<Integer, Double>();
    map2.put(1, 1.0);
    map2.put(2, 2.0);
    map2.put(3, 3.0);

    Map<String, Double> map3 = new new HashMap<String, Double>();
    map1.forEach((key,value)->map3.put(key,map2.get(value)));
```

This yields the following mapping

```
    "key1" -> 1.0
    "key2" -> 2.0
    "key3" -> 3.0
```

Section 28.6: Add multiple items

We can use `V put(K key,V value)`:

> Associates the specified value with the specified key in this map (optional operation). If the map previously contained a mapping for the key, the old value is replaced by the specified value.

```
String currentVal;
Map<Integer, String> map = new TreeMap<>();
currentVal = map.put(1, "First element.");
System.out.println(currentVal);// Will print null
currentVal = map.put(2, "Second element.");
System.out.println(currentVal); // Will print null yet again
currentVal = map.put(2, "This will replace 'Second element'");
System.out.println(currentVal); // will print Second element.
System.out.println(map.size()); // Will print 2 as key having
// value 2 was replaced.

Map<Integer, String> map2 = new HashMap<>();
map2.put(2, "Element 2");
map2.put(3, "Element 3");

map.putAll(map2);

System.out.println(map.size());
```

Output:

3

To add many items you can use an inner classes like this:

```
Map<Integer, String> map = new HashMap<>() {{
    // This is now an anonymous inner class with an unnamed instance constructor
    put(5, "high");
    put(4, "low");
    put(1, "too slow");
}};
```

Keep in mind that creating an anonymous inner class is not always efficient and can lead to memory leaks so when possible, use an initializer block instead:

```
static Map<Integer, String> map = new HashMap<>();

static {
    // Now no inner classes are created so we can avoid memory leaks
    put(5, "high");
    put(4, "low");
    put(1, "too slow");
}
```

The example above makes the map static. It can also be used in a non-static context by removing all occurrences of `static`.

In addition to that most implementations support `putAll`, which can add all entries in one map to another like this:

```
another.putAll(one);
```

Section 28.7: Creating and Initializing Maps

Introduction

`Maps` stores key/value pairs, where each key has an associated value. Given a particular key, the map can look up the associated value very quickly.

`Maps`, also known as associate array, is an object that stores the data in form of keys and values. In Java, maps are represented using Map interface which is not an extension of the collection interface.

- Way 1:

```
/*J2SE < 5.0*/
Map map = new HashMap();
map.put("name", "A");
map.put("address", "Malviya-Nagar");
map.put("city", "Jaipur");
System.out.println(map);
```

- Way 2:

```
/*J2SE 5.0+ style (use of generics):*/
Map<String, Object> map = new HashMap<>();
map.put("name", "A");
map.put("address", "Malviya-Nagar");
map.put("city", "Jaipur");
System.out.println(map);
```

- Way 3:

```
Map<String, Object> map = new HashMap<String, Object>(){{
    put("name", "A");
    put("address", "Malviya-Nagar");
    put("city", "Jaipur");
}};
System.out.println(map);
```

- Way 4:

```
Map<String, Object> map = new TreeMap<String, Object>();
    map.put("name", "A");
    map.put("address", "Malviya-Nagar");
    map.put("city", "Jaipur");
System.out.println(map);
```

- Way 5:

```
//Java 8
final Map<String, String> map =
    Arrays.stream(new String[][] {
        { "name", "A" },
        { "address", "Malviya-Nagar" },
        { "city", "jaipur" },
    }).collect(Collectors.toMap(m -> m[0], m -> m[1]));
System.out.println(map);
```

- Way 6:

```
//This way for initial a map in outside the function
final static Map<String, String> map;
static
{
    map = new HashMap<String, String>();
    map.put("a", "b");
    map.put("c", "d");
}
```

- Way 7: Creating an immutable single key-value map.

```
    //Immutable single key-value map
    Map<String, String> singletonMap = Collections.singletonMap("key", "value");
```

Please note, that **it is impossible to modify such map**.

Any attemts to modify the map will result in throwing the UnsupportedOperationException.

```
    //Immutable single key-value pair
    Map<String, String> singletonMap = Collections.singletonMap("key", "value");
    singletonMap.put("newKey", "newValue"); //will throw UnsupportedOperationException
    singletonMap.putAll(new HashMap<>()); //will throw UnsupportedOperationException
    singletonMap.remove("key"); //will throw UnsupportedOperationException
    singletonMap.replace("key", "value", "newValue"); //will throw
UnsupportedOperationException
    //and etc
```

Section 28.8: Check if key exists

```
Map<String, String> num = new HashMap<>();
num.put("one", "first");

if (num.containsKey("one")) {
    System.out.println(num.get("one")); // => first
}
```

Maps can contain null values

For maps, one has to be carrefull not to confuse "containing a key" with "having a value". For example, `HashMap`s can contain null which means the following is perfectly normal behavior :

```
Map<String, String> map = new HashMap<>();
map.put("one", null);
if (map.containsKey("one")) {
    System.out.println("This prints !"); // This line is reached
}
if (map.get("one") != null) {
    System.out.println("This is never reached !"); // This line is never reached
}
```

More formally, there is no guarantee that `map.contains(key) <=> map.get(key)!=null`

Section 28.9: Add an element

1. Addition

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "First element.");
System.out.println(map.get(1));
```

Output: `First element.`

2. Override

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "First element.");
map.put(1, "New element.");
System.out.println(map.get(1));
```

Output: **`New`** `element.`

`HashMap` is used as an example. Other implementations that implement the `Map` interface may be used as well.

Section 28.10: Clear the map

```
Map<Integer, String> map = new HashMap<>();

map.put(1, "First element.");
map.put(2, "Second element.");
map.put(3, "Third element.");

map.clear();

System.out.println(map.size());   // => 0
```

Section 28.11: Use custom object as key

Before using your own object as key you must override hashCode() and equals() method of your object.

In simple case you would have something like:

```
class MyKey {
    private String name;
    MyKey(String name) {
        this.name = name;
    }

    @Override
    public boolean equals(Object obj) {
        if(obj instanceof MyKey) {
            return this.name.equals(((MyKey)obj).name);
        }
        return false;
    }

    @Override
    public int hashCode() {
        return this.name.hashCode();
    }
}
```

`hashCode` will decide which hash bucket the key belongs to and `equals` will decide which object inside that hash bucket.

Without these method, the reference of your object will be used for above comparison which will not work unless you use the same object reference every time.

Chapter 29: LinkedHashMap

LinkedHashMap class is Hash table and Linked list implementation of the Map interface, with predictable iteration order. It inherits HashMap class and implements the Map interface.

The important points about Java LinkedHashMap class are: A LinkedHashMap contains values based on the key. It contains only unique elements. It may have one null key and multiple null values. It is same as HashMap instead maintains insertion order.

Section 29.1: Java LinkedHashMap class

Key Points:

- Is Hash table and Linked list implementation of the Map interface, with predictable iteration order.
- inherits HashMap class and implements the Map interface.
- contains values based on the key.
- only unique elements.
- may have one null key and multiple null values.
- same as HashMap instead maintains insertion order.

Methods:

- void clear().
- boolean containsKey(Object key).
- Object get(Object key).
- protected boolean removeEldestEntry(Map.Entry eldest)

Example:

```
public static void main(String arg[])
    {
        LinkedHashMap<String, String> lhm = new LinkedHashMap<String, String>();
        lhm.put("Ramesh", "Intermediate");
        lhm.put("Shiva", "B-Tech");
        lhm.put("Santosh", "B-Com");
        lhm.put("Asha", "Msc");
        lhm.put("Raghu", "M-Tech");

        Set set = lhm.entrySet();
        Iterator i = set.iterator();
        while (i.hasNext()) {
            Map.Entry me = (Map.Entry) i.next();
            System.out.println(me.getKey() + " : " + me.getValue());
        }

        System.out.println("The Key Contains : " + lhm.containsKey("Shiva"));
        System.out.println("The value to the corresponding to key : " + lhm.get("Asha"));
    }
```

Chapter 30: WeakHashMap

Concepts of weak Hashmap

Section 30.1: Concepts of WeakHashmap

Key Points:

- Implementation of Map.
- stores only weak references to its keys.

Weak References : The objects that are referenced only by weak references are garbage collected eagerly; the GC won't wait until it needs memory in that case.

Difference between Hashmap and WeakHashMap:

If the Java memory manager no longer has a strong reference to the object specified as a key, then the entry in the map will be removed in WeakHashMap.

Example:

```
public class WeakHashMapTest {
    public static void main(String[] args) {
        Map hashMap= new HashMap();

        Map weakHashMap = new WeakHashMap();

        String keyHashMap = new String("keyHashMap");
        String keyWeakHashMap = new String("keyWeakHashMap");

        hashMap.put(keyHashMap, "Ankita");
        weakHashMap.put(keyWeakHashMap, "Atul");
        System.gc();
        System.out.println("Before: hash map value:"+hashMap.get("keyHashMap")+" and weak hash map
value:"+weakHashMap.get("keyWeakHashMap"));

        keyHashMap = null;
        keyWeakHashMap = null;

        System.gc();

        System.out.println("After: hash map value:"+hashMap.get("keyHashMap")+" and weak hash map
value:"+weakHashMap.get("keyWeakHashMap"));
    }
```

Size differences (HashMap vs WeakHashMap):

Calling size() method on HashMap object will return the same number of key-value pairs. size will decrease only if remove() method is called explicitly on the HashMap object.

Because the garbage collector may discard keys at anytime, a WeakHashMap may behave as though an unknown thread is silently removing entries. So it is possible for the size method to return smaller values over time.So, in **WeakHashMap size decrease happens automatically**.

Chapter 31: SortedMap

Introduction to sorted Map.

Section 31.1: Introduction to sorted Map

Keypoint:

- SortedMap interface extends Map.
- entries are maintained in an ascending key order.

Methods of sorted Map :

- Comparator comparator( ).
- Object firstKey( ).
- SortedMap headMap(Object end).
- Object lastKey( ).
- SortedMap subMap(Object start, Object end).
- SortedMap tailMap(Object start).

Example

```
public static void main(String args[]) {
      // Create a hash map
      TreeMap tm = new TreeMap();

      // Put elements to the map
      tm.put("Zara", new Double(3434.34));
      tm.put("Mahnaz", new Double(123.22));
      tm.put("Ayan", new Double(1378.00));
      tm.put("Daisy", new Double(99.22));
      tm.put("Qadir", new Double(-19.08));

      // Get a set of the entries
      Set set = tm.entrySet();

      // Get an iterator
      Iterator i = set.iterator();

      // Display elements
      while(i.hasNext()) {
         Map.Entry me = (Map.Entry)i.next();
         System.out.print(me.getKey() + ": ");
         System.out.println(me.getValue());
      }
      System.out.println();

      // Deposit 1000 into Zara's account
      double balance = ((Double)tm.get("Zara")).doubleValue();
      tm.put("Zara", new Double(balance + 1000));
      System.out.println("Zara's new balance: " + tm.get("Zara"));
   }
```

Chapter 32: TreeMap and TreeSet

`TreeMap` and `TreeSet` are basic Java collections added in Java 1.2. `TreeMap` is a **mutable**, **ordered**, `Map` implementation. Similarly, `TreeSet` is a **mutable**, **ordered** `Set` implementation.

`TreeMap` is implemented as a Red-Black tree, which provides `O(log n)` access times. `TreeSet` is implemented using a `TreeMap` with dummy values.

Both collections are **not** thread-safe.

Section 32.1: TreeMap of a simple Java type

First, we create an empty map, and insert some elements into it:

Version ≥ Java SE 7

```
TreeMap<Integer, String> treeMap = new TreeMap<>();
```

Version < Java SE 7

```
TreeMap<Integer, String> treeMap = new TreeMap<Integer, String>();
```

```
treeMap.put(10, "ten");
treeMap.put(4, "four");
treeMap.put(1, "one");
treeSet.put(12, "twelve");
```

Once we have a few elements in the map, we can perform some operations:

```
System.out.println(treeMap.firstEntry()); // Prints 1=one
System.out.println(treeMap.lastEntry()); // Prints 12=twelve
System.out.println(treeMap.size()); // Prints 4, since there are 4 elemens in the map
System.out.println(treeMap.get(12)); // Prints twelve
System.out.println(treeMap.get(15)); // Prints null, since the key is not found in the map
```

We can also iterate over the map elements using either an Iterator, or a foreach loop. Note that the entries are printed according to their natural ordering, not the insertion order:

Version ≥ Java SE 7

```
for (Entry<Integer, String> entry : treeMap.entrySet()) {
    System.out.print(entry + " "); //prints 1=one 4=four 10=ten 12=twelve
}
```

```
Iterator<Entry<Integer, String>> iter = treeMap.entrySet().iterator();
while (iter.hasNext()) {
    System.out.print(iter.next() + " "); //prints 1=one 4=four 10=ten 12=twelve
}
```

Section 32.2: TreeSet of a simple Java Type

First, we create an empty set, and insert some elements into it:

Version ≥ Java SE 7

```
TreeSet<Integer> treeSet = new TreeSet<>();
```

Version < Java SE 7

```
TreeSet<Integer> treeSet = new TreeSet<Integer>();
```

```
treeSet.add(10);
treeSet.add(4);
treeSet.add(1);
```

```
treeSet.add(12);
```

Once we have a few elements in the set, we can perform some operations:

```
System.out.println(treeSet.first()); // Prints 1
System.out.println(treeSet.last()); // Prints 12
System.out.println(treeSet.size()); // Prints 4, since there are 4 elemens in the set
System.out.println(treeSet.contains(12)); // Prints true
System.out.println(treeSet.contains(15)); // Prints false
```

We can also iterate over the map elements using either an Iterator, or a foreach loop. Note that the entries are printed according to their natural ordering, not the insertion order:

Version ≥ Java SE 7

```
for (Integer i : treeSet) {
    System.out.print(i + " "); //prints 1 4 10 12
}
```

```
Iterator<Integer> iter = treeSet.iterator();
while (iter.hasNext()) {
    System.out.print(iter.next() + " "); //prints 1 4 10 12
}
```

Section 32.3: TreeMap/TreeSet of a custom Java type

Since `TreeMap`s and `TreeSet`s maintain keys/elements according to their natural ordering. Therefor `TreeMap` keys and `TreeSet` elements have to comparable to one another.

Say we have a custom `Person` class:

```
public class Person  {

    private int id;
    private String firstName, lastName;
    private Date birthday;

    //... Constuctors, getters, setters and various methods
}
```

If we store it as-is in a `TreeSet` (or a Key in a TreeMap):

```
TreeSet<Person2> set = ...
set.add(new Person(1,"first","last",Date.from(Instant.now())));
```

Then we'd run into an Exception such as this one:

```
Exception in thread "main" java.lang.ClassCastException: Person cannot be cast to
java.lang.Comparable
    at java.util.TreeMap.compare(TreeMap.java:1294)
    at java.util.TreeMap.put(TreeMap.java:538)
    at java.util.TreeSet.add(TreeSet.java:255)
```

To fix that, let's assume that we want to order `Person` instances based on the order of their ids (`private int id`). We could do it in one of two ways:

1. One solution is to modify `Person` so it would implement the Comparable interface:

```
public class Person implements Comparable<Person> {
    private int id;
    private String firstName, lastName;
    private Date birthday;

    //... Constuctors, getters, setters and various methods

    @Override
    public int compareTo(Person o) {
        return Integer.compare(this.id, o.id); //Compare by id
    }
}
```

2. Another solution is to provide the `TreeSet` with a Comparator:

Version ≥ Java SE 8

```
TreeSet<Person> treeSet = new TreeSet<>((personA, personB) -> Integer.compare(personA.getId(),
personB.getId()));
```

```
TreeSet<Person> treeSet = new TreeSet<>(new Comparator<Person>(){
    @Override
    public int compare(Person personA, Person personB) {
        return Integer.compare(personA.getId(), personB.getId());
    }
});
```

However, there are two caveats to both approaches:

1. It's **very important** not to modify any fields used for ordering once an instance has been inserted into a `TreeSet`/`TreeMap`. In the above example, if we change the `id` of a person that's already inserted into the collection, we might run into unexpected behavior.
2. It's important to implement the comparison properly and consistently. As per the Javadoc:

> The implementor must ensure `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))` for all x and y. (This implies that `x.compareTo(y)` must throw an exception iff `y.compareTo(x)` throws an exception.)
>
> The implementor must also ensure that the relation is transitive: `(x.compareTo(y)>0 && y.compareTo(z)>0)` implies `x.compareTo(z)>0`.
>
> Finally, the implementor must ensure that `x.compareTo(y)==0` implies that `sgn(x.compareTo(z)) == sgn(y.compareTo(z))`, for all z.

Section 32.4: TreeMap and TreeSet Thread Safety

`TreeMap` and `TreeSet` are **not** thread-safe collections, so care must be taken to ensure when used in multi-threaded programs.

Both `TreeMap` and `TreeSet` are safe when read, even concurrently, by multiple threads. So if they have been created and populated by a single thread (say, at the start of the program), and only then read, but not modified by multiple threads, there's no reason for synchronization or locking.

However, if read and modified concurrently, or modified concurrently by more than one thread, the collection might throw a ConcurrentModificationException or behave unexpectedly. In these cases, it's imperative to synchronize/lock access to the collection using one of the following approaches:

1. Using `Collections.synchronizedSorted..`:

```
SortedSet<Integer> set = Collections.synchronizedSortedSet(new TreeSet<Integer>());
SortedMap<Integer,String> map = Collections.synchronizedSortedMap(new
TreeMap<Integer,String>());
```

   This will provide a SortedSet/SortedMap implementation backed by the actual collection, and synchronized on some mutex object. Note that this will synchronize all read and write access to the collection on a single lock, so even concurrent reads would not be possible.

2. By manually synchronizing on some object, like the collection itself:

```
TreeSet<Integer> set = new TreeSet<>();
```

   ...

```
//Thread 1
synchronized (set) {
    set.add(4);
}
```

   ...

```
//Thread 2
synchronized (set) {
    set.remove(5);
}
```

3. By using a lock, such as a ReentrantReadWriteLock:

```
TreeSet<Integer> set = new TreeSet<>();
ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
```

   ...

```
//Thread 1
lock.writeLock().lock();
set.add(4);
lock.writeLock().unlock();
```

   ...

```
//Thread 2
lock.readLock().lock();
set.contains(5);
lock.readLock().unlock();
```

As opposed to the previous synchronization methods, using a ReadWriteLock allows multiple threads to read from the map concurrently.

Chapter 33: Queues and Deques

Section 33.1: The usage of the PriorityQueue

`PriorityQueue` is a data structure. Like `SortedSet`, `PriorityQueue` sorts also its elements based on their priorities. The elements, which have a higher priority, comes first. The type of the `PriorityQueue` should implement `comparable` or `comparator` interface, whose methods decides the priorities of the elements of the data structure.

```
//The type of the PriorityQueue is Integer.
PriorityQueue<Integer> queue = new PriorityQueue<Integer>();

//The elements are added to the PriorityQueue
queue.addAll( Arrays.asList( 9, 2, 3, 1, 3, 8 ) );

//The PriorityQueue sorts the elements by using compareTo method of the Integer Class
//The head of this queue is the least element with respect to the specified ordering
System.out.println( queue );  //The Output: [1, 2, 3, 9, 3, 8]
queue.remove();
System.out.println( queue );  //The Output: [2, 3, 3, 9, 8]
queue.remove();
System.out.println( queue );  //The Output: [3, 8, 3, 9]
queue.remove();
System.out.println( queue );  //The Output: [3, 8, 9]
queue.remove();
System.out.println( queue );  //The Output: [8, 9]
queue.remove();
System.out.println( queue );  //The Output: [9]
queue.remove();
System.out.println( queue );  //The Output: []
```

Section 33.2: Deque

A `Deque` is a "double ended queue" which means that a elements can be added at the front or the tail of the queue. The queue only can add elements to the tail of a queue.

The `Deque` inherits the `Queue` interface which means the regular methods remain, however the Deque interface offers additional methods to be more flexible with a queue. The additional methods really speak for them self if you know how a queue works, since those methods are intended to add more flexibility:

Method	Brief description
`getFirst()`	Gets the first item of the **head** of the queue without removing it.
`getLast()`	Gets the first item of the **tail** of the queue without removing it.
`addFirst(E e)`	Adds an item to the **head** of the queue
`addLast(E e)`	Adds an item to the **tail** of the queue
`removeFirst()`	Removes the first item at the **head** of the queue
`removeLast()`	Removes the first item at the **tail** of the queue

Of course the same options for `offer`, `poll` and `peek` are available, however they do not work with exceptions but rather with special values. There is no point in showing what they do here.

Adding and Accessing Elements

To add elements to the tail of a Deque you call its `add()` method. You can also use the `addFirst()` and `addLast()` methods, which add elements to the head and tail of the deque.

```
Deque<String> dequeA = new LinkedList<>();

dequeA.add("element 1");      //add element at tail
dequeA.addFirst("element 2"); //add element at head
dequeA.addLast("element 3");  //add element at tail
```

You can peek at the element at the head of the queue without taking the element out of the queue. This is done via the `element()` method. You can also use the `getFirst()` and `getLast()` methods, which return the first and last element in the `Deque`. Here is how that looks:

```
String firstElement0 = dequeA.element();
String firstElement1 = dequeA.getFirst();
String lastElement = dequeA.getLast();
```

Removing Elements

To remove elements from a deque, you call the `remove()`, `removeFirst()` and `removeLast()` methods. Here are a few examples:

```
String firstElement = dequeA.remove();
String firstElement = dequeA.removeFirst();
String lastElement  = dequeA.removeLast();
```

Section 33.3: Stacks

What is a Stack?

In Java, Stacks are a LIFO (Last In, First Out) Data structure for objects.

Stack API

Java contains a Stack API with the following methods

```
Stack()            //Creates an empty Stack
isEmpty()          //Is the Stack Empty?            Return Type: Boolean
push(Item item)    //push an item onto the stack
pop()              //removes item from top of stack  Return Type: Item
size()             //returns # of items in stack     Return Type: Int
```

Example

```
import java.util.*;

public class StackExample {

   public static void main(String args[]) {
      Stack st = new Stack();
      System.out.println("stack: " + st);

      st.push(10);
      System.out.println("10 was pushed to the stack");
      System.out.println("stack: " + st);

      st.push(15);
      System.out.println("15 was pushed to the stack");
      System.out.println("stack: " + st);

      st.push(80);
      System.out.println("80 was pushed to the stack");
      System.out.println("stack: " + st);
```

```
      st.pop();
      System.out.println("80 was popped from the stack");
      System.out.println("stack: " + st);

      st.pop();
      System.out.println("15 was popped from the stack");
      System.out.println("stack: " + st);

      st.pop();
      System.out.println("10 was popped from the stack");
      System.out.println("stack: " + st);

     if(st.isEmpty())
      {
         System.out.println("empty stack");
      }
   }
}
```

This returns:

```
stack: []
10 was pushed to the stack
stack: [10]
15 was pushed to the stack
stack: [10, 15]
80 was pushed to the stack
stack: [10, 15, 80]
80 was popped from the stack
stack: [10, 15]
15 was popped from the stack
stack: [10]
10 was popped from the stack
stack: []
empty stack
```

Section 33.4: BlockingQueue

A BlockingQueue is an interface, which is a queue that blocks when you try to dequeue from it and the queue is empty, or if you try to enqueue items to it and the queue is already full. A thread trying to dequeue from an empty queue is blocked until some other thread inserts an item into the queue. A thread trying to enqueue an item in a full queue is blocked until some other thread makes space in the queue, either by dequeuing one or more items or clearing the queue completely.

BlockingQueue methods come in four forms, with different ways of handling operations that cannot be satisfied immediately, but may be satisfied at some point in the future: one throws an exception, the second returns a special value (either null or false, depending on the operation), the third blocks the current thread indefinitely until the operation can succeed, and the fourth blocks for only a given maximum time limit before giving up.

Operation	**Throws Exception**	**Special Value**	**Blocks**	**Times out**
Insert	add()	offer(e)	put(e)	offer(e, time, unit)
Remove	remove()	poll()	take()	poll(time, unit)
Examine	element()	peek()	N/A	N/A

A `BlockingQueue` can be **bounded** or **unbounded**. A bounded BlockingQueue is one which is initialized with initial

capacity.

```
BlockingQueue<String> bQueue = new ArrayBlockingQueue<String>(2);
```

Any calls to a put() method will be blocked if the size of the queue is equal to the initial capacity defined.

An unbounded Queue is one which is initialized without capacity, actually by default it initialized with Integer.MAX_VALUE.

Some common implementations of `BlockingQueue` are:

1. ArrayBlockingQueue
2. LinkedBlockingQueue
3. PriorityBlockingQueue

Now let's look at an example of `ArrayBlockingQueue`:

```
BlockingQueue<String> bQueue = new ArrayBlockingQueue<>(2);
bQueue.put("This is entry 1");
System.out.println("Entry one done");
bQueue.put("This is entry 2");
System.out.println("Entry two done");
bQueue.put("This is entry 3");
System.out.println("Entry three done");
```

This will print:

```
Entry one done
Entry two done
```

And the thread will be blocked after the second output.

Section 33.5: LinkedList as a FIFO Queue

The `java.util.LinkedList` class, while implementing `java.util.List` is a general-purpose implementation of `java.util.Queue` interface too operating on a FIFO (First In, First Out) principle.

In the example below, with `offer()` method, the elements are inserted into the `LinkedList`. This insertion operation is called `enqueue`. In the **`while`** loop below, the elements are removed from the `Queue` based on FIFO. This operation is called `dequeue`.

```
Queue<String> queue = new LinkedList<String>();

queue.offer( "first element" );
queue.offer( "second element" );
queue.offer( "third element" );
queue.offer( "fourth. element" );
queue.offer( "fifth. element" );

while ( !queue.isEmpty() ) {
  System.out.println( queue.poll() );
}
```

The output of this code is

```
first element
```

```
second element
third element
fourth element
fifth element
```

As seen in the output, the first inserted element "first element" is removed firstly, "second element" is removed in the second place etc.

Section 33.6: Queue Interface

Basics

A `Queue` is a collection for holding elements prior to processing. Queues typically, but not necessarily, order elements in a FIFO (first-in-first-out) manner.

Head of the queue is the element that would be removed by a call to remove or poll. In a FIFO queue, all new elements are inserted at the tail of the queue.

The Queue Interface

```
public interface Queue<E> extends Collection<E> {
    boolean add(E e);

    boolean offer(E e);

    E remove();

    E poll();

    E element();

    E peek();
}
```

Each `Queue` method exists in two forms:

- one throws an exception if the operation fails;
- other returns a special value if the operation fails (either `null` or `false` depending on the operation.

Type of operation	Throws exception	Returns special value
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

Chapter 34: Dequeue Interface

A Deque is linear collection that supports element insertion and removal at both ends.

The name deque is short for "double ended queue" and is usually pronounced "deck".

Most Deque implementations place no fixed limits on the number of elements they may contain, but this interface supports capacity-restricted deques as well as those with no fixed size limit.

The Deque interface is a richer abstract data type than both Stack and Queue because it implements both stacks and queues at same time

Section 34.1: Adding Elements to Deque

```
Deque deque = new LinkedList();

//Adding element at tail
deque.add("Item1");

//Adding element at head
deque.addFirst("Item2");

//Adding element at tail
deque.addLast("Item3");
```

Section 34.2: Removing Elements from Deque

```
//Retrieves and removes the head of the queue represented by this deque
Object headItem = deque.remove();

//Retrieves and removes the first element of this deque.
Object firstItem = deque.removeFirst();

//Retrieves and removes the last element of this deque.
Object lastItem = deque.removeLast();
```

Section 34.3: Retrieving Element without Removing

```
//Retrieves, but does not remove, the head of the queue represented by this deque
Object headItem = deque.element();

//Retrieves, but does not remove, the first element of this deque.
Object firstItem = deque.getFirst();

//Retrieves, but does not remove, the last element of this deque.
Object lastItem  = deque.getLast();
```

Section 34.4: Iterating through Deque

```
//Using Iterator
Iterator iterator = deque.iterator();
while(iterator.hasNext(){
  String Item = (String) iterator.next();
}

//Using For Loop
```

```
for(Object object : deque) {
    String Item = (String) object;
}
```

Chapter 35: Enums

Java enums (declared using the `enum` keyword) are shorthand syntax for sizable quantities of constants of a single class.

Section 35.1: Declaring and using a basic enum

Enum can be considered to be syntax sugar for a sealed class that is instantiated only a number of times known at compile-time to define a set of constants.

A simple enum to list the different seasons would be declared as follows:

```
public enum Season {
    WINTER,
    SPRING,
    SUMMER,
    FALL
}
```

While the enum constants don't necessarily need to be in all-caps, it is Java convention that names of constants are entirely uppercase, with words separated by underscores.

You can declare an Enum in its own file:

```
/**
 * This enum is declared in the Season.java file.
*/
public enum Season {
    WINTER,
    SPRING,
    SUMMER,
    FALL
}
```

But you can also declare it inside another class:

```
public class Day {

    private Season season;

    public String getSeason() {
        return season.name();
    }

    public void setSeason(String season) {
        this.season = Season.valueOf(season);
    }

    /**
     * This enum is declared inside the Day.java file and
     * cannot be accessed outside because it's declared as private.
     */
    private enum Season {
        WINTER,
        SPRING,
        SUMMER,
        FALL
```

```
    }

}
```

Finally, you cannot declare an Enum inside a method body or constructor:

```
public class Day {

    /**
     * Constructor
    */
    public Day() {
        // Illegal. Compilation error
        enum Season {
            WINTER,
            SPRING,
            SUMMER,
            FALL
        }
    }

    public void aSimpleMethod() {
        // Legal. You can declare a primitive (or an Object) inside a method. Compile!
        int primitiveInt = 42;

        // Illegal. Compilation error.
        enum Season {
            WINTER,
            SPRING,
            SUMMER,
            FALL
        }

        Season season = Season.SPRING;
    }

}
```

Duplicate enum constants are not allowed:

```
public enum Season {
    WINTER,
    WINTER, //Compile Time Error : Duplicate Constants
    SPRING,
    SUMMER,
    FALL
}
```

Every constant of enum is **`public`, `static`** and **`final`** by default. As every constant is **`static`**, they can be accessed directly using the enum name.

Enum constants can be passed around as method parameters:

```
public static void display(Season s) {
    System.out.println(s.name());  // name() is a built-in method that gets the exact name of the
enum constant
}

display(Season.WINTER);  // Prints out "WINTER"
```

You can get an array of the enum constants using the `values()` method. The values are guaranteed to be in declaration order in the returned array:

```
Season[] seasons = Season.values();
```

Note: this method allocates a new array of values each time it is called.

To iterate over the enum constants:

```
public static void enumIterate() {
    for (Season s : Season.values()) {
        System.out.println(s.name());
    }
}
```

You can use enums in a **`switch`** statement:

```
public static void enumSwitchExample(Season s) {
    switch(s) {
        case WINTER:
            System.out.println("It's pretty cold");
            break;
        case SPRING:
            System.out.println("It's warming up");
            break;
        case SUMMER:
            System.out.println("It's pretty hot");
            break;
        case FALL:
            System.out.println("It's cooling down");
            break;
    }
}
```

You can also compare enum constants using `==`:

```
Season.FALL == Season.WINTER    // false
Season.SPRING == Season.SPRING  // true
```

Another way to compare enum constants is by using `equals()` as below, which is considered bad practice as you can easily fall into pitfalls as follows:

```
Season.FALL.equals(Season.FALL); // true
Season.FALL.equals(Season.WINTER); // false
Season.FALL.equals("FALL"); // false and no compiler error
```

Furthermore, although the set of instances in the **`enum`** cannot be changed at run-time, the instances themselves are not inherently immutable because like any other class, an **`enum`** can contain mutable fields as is demonstrated below.

```
public enum MutableExample {
    A,
    B;

    private int count = 0;

    public void increment() {
```

```
        count++;
    }

    public void print() {
        System.out.println("The count of " + name() + " is " + count);
    }
}

// Usage:
MutableExample.A.print();       // Outputs 0
MutableExample.A.increment();
MutableExample.A.print();       // Outputs 1 -- we've changed a field
MutableExample.B.print();       // Outputs 0 -- another instance remains unchanged
```

However, a good practice is to make **enum** instances immutable, i.e. when they either don't have any additional fields or all such fields are marked as **final** and are immutable themselves. This will ensure that for a lifetime of the application an **enum** won't leak any memory and that it is safe to use its instances across all threads.

Enums implicitly implement Serializable and Comparable because the **Enum** class does:

```
public abstract class Enum<E extends Enum<E>>
extends Object
implements Comparable<E>, Serializable
```

Section 35.2: Enums with constructors

An **enum** cannot have a public constructor; however, private constructors are acceptable (constructors for enums are package-private by default):

```
public enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25); // usual names for US coins
    // note that the above parentheses and the constructor arguments match
    private int value;

    Coin(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}

int p = Coin.NICKEL.getValue(); // the int value will be 5
```

It is recommended that you keep all fields private and provide getter methods, as there are a finite number of instances for an enum.

If you were to implement an **Enum** as a **class** instead, it would look like this:

```
public class Coin<T extends Coin<T>> implements Comparable<T>, Serializable{
    public static final Coin PENNY = new Coin(1);
    public static final Coin NICKEL = new Coin(5);
    public static final Coin DIME = new Coin(10);
    public static final Coin QUARTER = new Coin(25);

    private int value;
```

```
    private Coin(int value){
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}

int p = Coin.NICKEL.getValue(); // the int value will be 5
```

Enum constants are technically mutable, so a setter could be added to change the internal structure of an enum constant. However, this is considered very bad practice and should be avoided.

Best practice is to make Enum fields immutable, with **`final`**:

```
public enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);

    private final int value;

    Coin(int value){
        this.value = value;
    }

    ...

}
```

You may define multiple constructors in the same enum. When you do, the arguments you pass in your enum declaration decide which constructor is called:

```
public enum Coin {
    PENNY(1, true), NICKEL(5, false), DIME(10), QUARTER(25);

    private final int value;
    private final boolean isCopperColored;

    Coin(int value){
        this(value, false);
    }

    Coin(int value, boolean isCopperColored){
        this.value = value;
        this.isCopperColored = isCopperColored;
    }

    ...

}
```

Note: All non-primitive enum fields should implement `Serializable` because the **`Enum`** class does.

Section 35.3: Enums with Abstract Methods

Enums can define abstract methods, which each **`enum`** member is required to implement.

```
enum Action {
    DODGE {
```

```
        public boolean execute(Player player) {
            return player.isAttacking();
        }
    },
    ATTACK {
        public boolean execute(Player player) {
            return player.hasWeapon();
        }
    },
    JUMP {
        public boolean execute(Player player) {
            return player.getCoordinates().equals(new Coordinates(0, 0));
        }
    };

    public abstract boolean execute(Player player);
}
```

This allows for each enum member to define its own behaviour for a given operation, without having to switch on types in a method in the top-level definition.

Note that this pattern is a short form of what is typically achieved using polymorphism and/or implementing interfaces.

Section 35.4: Implements Interface

This is an `enum` that is also a callable function that tests `String` inputs against precompiled regular expression patterns.

```
import java.util.function.Predicate;
import java.util.regex.Pattern;

enum RegEx implements Predicate<String> {
    UPPER("[A-Z]+"), LOWER("[a-z]+"), NUMERIC("[+-]?[0-9]+");

    private final Pattern pattern;

    private RegEx(final String pattern) {
        this.pattern = Pattern.compile(pattern);
    }

    @Override
    public boolean test(final String input) {
        return this.pattern.matcher(input).matches();
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println(RegEx.UPPER.test("ABC"));
        System.out.println(RegEx.LOWER.test("abc"));
        System.out.println(RegEx.NUMERIC.test("+111"));
    }
}
```

Each member of the enum can also implement the method:

```
import java.util.function.Predicate;
```

```
enum Acceptor implements Predicate<String> {
    NULL {
        @Override
        public boolean test(String s) { return s == null; }
    },
    EMPTY {
        @Override
        public boolean test(String s) { return s.equals(""); }
    },
    NULL_OR_EMPTY {
        @Override
        public boolean test(String s) { return NULL.test(s) || EMPTY.test(s); }
    };
}

public class Main {
    public static void main(String[] args) {
        System.out.println(Acceptor.NULL.test(null));  // true
        System.out.println(Acceptor.EMPTY.test(""));   // true
        System.out.println(Acceptor.NULL_OR_EMPTY.test(" ")); // false
    }
}
```

Section 35.5: Implement Singleton pattern with a single-element enum

Enum constants are instantiated when an enum is referenced for the first time. Therefore, that allows to implement Singleton software design pattern with a single-element enum.

```
public enum Attendant {

    INSTANCE;

    private Attendant() {
        // perform some initialization routine
    }

    public void sayHello() {
        System.out.println("Hello!");
    }
}


public class Main {

    public static void main(String... args) {
        Attendant.INSTANCE.sayHello();// instantiated at this point
    }
}
```

According to "Effective Java" book by Joshua Bloch, a single-element enum is the best way to implement a singleton. This approach has following advantages:

- thread safety
- guarantee of single instantiation
- out-of-the-box serialization

And as shown in the section implements interface this singleton might also implement one or more interfaces.

Section 35.6: Using methods and static blocks

An enum can contain a method, just like any class. To see how this works, we'll declare an enum like this:

```
public enum Direction {
    NORTH, SOUTH, EAST, WEST;
}
```

Let's have a method that returns the enum in the opposite direction:

```
public enum Direction {
    NORTH, SOUTH, EAST, WEST;

    public Direction getOpposite(){
        switch (this){
            case NORTH:
                return SOUTH;
            case SOUTH:
                return NORTH;
            case WEST:
                return EAST;
            case EAST:
                return WEST;
            default: //This will never happen
                return null;
        }
    }
}
```

This can be improved further through the use of fields and static initializer blocks:

```
public enum Direction {
    NORTH, SOUTH, EAST, WEST;

    private Direction opposite;

    public Direction getOpposite(){
        return opposite;
    }

    static {
        NORTH.opposite = SOUTH;
        SOUTH.opposite = NORTH;
        WEST.opposite = EAST;
        EAST.opposite = WEST;
    }
}
```

In this example, the opposite direction is stored in a private instance field `opposite`, which is statically initialized the first time a `Direction` is used. In this particular case (because `NORTH` references `SOUTH` and conversely), we cannot use Enums with constructors here (Constructors `NORTH(SOUTH)`, `SOUTH(NORTH)`, `EAST(WEST)`, `WEST(EAST)` would be more elegant and would allow `opposite` to be declared **`final`**, but would be self-referential and therefore are not allowed).

Section 35.7: Zero instance enum

```
enum Util {
    /* No instances */;
```

```
    public static int clamp(int min, int max, int i) {
        return Math.min(Math.max(i, min), max);
    }

    // other utility methods...
}
```

Just as `enum` can be used for singletons (1 instance classes), it can be used for utility classes (0 instance classes). Just make sure to terminate the (empty) list of enum constants with a `;`.

See the question Zero instance enum vs private constructors for preventing instantiation for a discussion on pro's and con's compared to private constructors.

Section 35.8: Enum as a bounded type parameter

When writing a class with generics in java, it is possible to ensure that the type parameter is an enum. Since all enums extend the `Enum` class, the following syntax may be used.

```
public class Holder<T extends Enum<T>> {
    public final T value;

    public Holder(T init) {
        this.value = init;
    }
}
```

In this example, the type `T` *must* be an enum.

Section 35.9: Documenting enums

Not always the `enum` name is clear enough to be understood. To document an `enum`, use standard javadoc:

```
/**
 * United States coins
 */
public enum Coins {

    /**
     * One-cent coin, commonly known as a penny,
     * is a unit of currency equaling one-hundredth
     * of a United States dollar
     */
    PENNY(1),

    /**
     * A nickel is a five-cent coin equaling
     * five-hundredth of a United States dollar
     */
    NICKEL(5),

    /**
     * The dime is a ten-cent coin refers to
     * one tenth of a United States dollar
     */
    DIME(10),

    /**
     * The quarter is a US coin worth 25 cents,
     * one-fourth of a United States dollar
```

```
     */
    QUARTER(25);

    private int value;

    Coins(int value){
        this.value = value;
    }

    public int getValue(){
        return value;
    }
}
```

Section 35.10: Enum constant specific body

In an `enum` it is possible to define a specific behavior for a particular constant of the `enum` which overrides the default behavior of the `enum`, this technique is known as *constant specific body*.

Suppose three piano students - John, Ben and Luke - are defined in an `enum` named `PianoClass`, as follows:

```
    enum PianoClass {
    JOHN, BEN, LUKE;
        public String getSex() {
            return "Male";
        }
        public String getLevel() {
            return "Beginner";
        }
    }
```

And one day two other students arrive - Rita and Tom - with a sex (Female) and level (Intermediate) that do not match the previous ones:

```
    enum PianoClass2 {
    JOHN, BEN, LUKE, RITA, TOM;
        public String getSex() {
            return "Male"; // issue, Rita is a female
        }
        public String getLevel() {
            return "Beginner"; // issue, Tom is an intermediate student
        }
    }
```

so that simply adding the new students to the constant declaration, as follows, is not correct:

```
PianoClass2 tom = PianoClass2.TOM;
PianoClass2 rita = PianoClass2.RITA;
System.out.println(tom.getLevel()); // prints Beginner -> wrong Tom's not a beginner
System.out.println(rita.getSex()); // prints Male -> wrong Rita's not a male
```

It's possible to define a specific behavior for each of the constant, Rita and Tom, which overrides the `PianoClass2` default behavior as follows:

```
enum PianoClass3 {
    JOHN, BEN, LUKE,
    RITA {
        @Override
```

```
        public String getSex() {
            return "Female";
        }
    },
    TOM {
        @Override
        public String getLevel() {
            return "Intermediate";
        }
    };
    public String getSex() {
        return "Male";
    }
    public String getLevel() {
        return "Beginner";
    }
}
```

and now Tom's level and Rita's sex are as they should be:

```
PianoClass3 tom = PianoClass3.TOM;
PianoClass3 rita = PianoClass3.RITA;
System.out.println(tom.getLevel()); // prints Intermediate
System.out.println(rita.getSex()); // prints Female
```

Another way to define content specific body is by using constructor, for instance:

```
enum Friend {
    MAT("Male"),
    JOHN("Male"),
    JANE("Female");

    private String gender;

    Friend(String gender) {
        this.gender = gender;
    }

    public String getGender() {
        return this.gender;
    }
}
```

and usage:

```
Friend mat = Friend.MAT;
Friend john = Friend.JOHN;
Friend jane = Friend.JANE;
System.out.println(mat.getGender());     // Male
System.out.println(john.getGender());    // Male
System.out.println(jane.getGender());    // Female
```

Section 35.11: Getting the values of an enum

Each enum class contains an implicit static method named `values()`. This method returns an array containing all values of that enum. You can use this method to iterate over the values. It is important to note however that this method returns a **new** array every time it is called.

```
public enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;

    /**
    * Print out all the values in this enum.
    */
    public static void printAllDays() {
        for(Day day : Day.values()) {
            System.out.println(day.name());
        }
    }
}
```

If you need a `Set` you can use `EnumSet.allOf(Day.class)` as well.

Section 35.12: Enum Polymorphism Pattern

When a method need to accept an "extensible" set of **`enum`** values, the programmer can apply polymorphism like on a normal **`class`** by creating an interface which will be used anywere where the **`enum`**s shall be used:

```
public interface ExtensibleEnum {
    String name();
}
```

This way, any **`enum`** tagged by (implementing) the interface can be used as a parameter, allowing the programmer to create a variable amount of **`enum`**s that will be accepted by the method. This can be useful, for example, in APIs where there is a default (unmodifiable) **`enum`** and the user of these APIs want to "extend" the **`enum`** with more values.

A set of default enum values can be defined as follows:

```
public enum DefaultValues implements ExtensibleEnum {
    VALUE_ONE, VALUE_TWO;
}
```

Additional values can then be defined like this:

```
public enum ExtendedValues implements ExtensibleEnum {
    VALUE_THREE, VALUE_FOUR;
}
```

Sample which shows how to use the enums - note how `printEnum()` accepts values from both **`enum`** types:

```
private void printEnum(ExtensibleEnum val) {
    System.out.println(val.name());
}

printEnum(DefaultValues.VALUE_ONE);    // VALUE_ONE
printEnum(DefaultValues.VALUE_TWO);    // VALUE_TWO
printEnum(ExtendedValues.VALUE_THREE); // VALUE_THREE
printEnum(ExtendedValues.VALUE_FOUR);  // VALUE_FOUR
```

Note: This pattern does not prevent you from redefining enum values, which are already defined in one enum, in another enum. These enum values would be different instances then. Also, it is not possible to use switch-on-enum since all we have is the interface, not the real **`enum`**.

Section 35.13: Compare and Contains for Enum values

Enums contains only constants and can be compared directly with `==`. So, only reference check is needed, no need to use `.equals` method. Moreover, if `.equals` used incorrectly, may raise the `NullPointerException` while that's not the case with `==` check.

```
enum Day {
    GOOD, AVERAGE, WORST;
}

public class Test {

    public static void main(String[] args) {
        Day day = null;

        if (day.equals(Day.GOOD)) {//NullPointerException!
            System.out.println("Good Day!");
        }

        if (day == Day.GOOD) {//Always use == to compare enum
            System.out.println("Good Day!");
        }

    }
}
```

To group, complement, range the enum values we have `EnumSet` class which contains different methods.

- `EnumSet#range` : To get subset of enum by range defined by two endpoints
- `EnumSet#of` : Set of specific enums without any range. Multiple overloaded `of` methods are there.
- `EnumSet#complementOf` : Set of enum which is complement of enum values provided in method parameter

```
enum Page {
   A1, A2, A3, A4, A5, A6, A7, A8, A9, A10
}

public class Test {

  public static void main(String[] args) {
      EnumSet<Page> range = EnumSet.range(Page.A1, Page.A5);

      if (range.contains(Page.A4)) {
          System.out.println("Range contains A4");
      }

      EnumSet<Page> of = EnumSet.of(Page.A1, Page.A5, Page.A3);

      if (of.contains(Page.A1)) {
          System.out.println("Of contains A1");
      }
  }
}
```

Section 35.14: Get enum constant by name

Say we have an enum `DayOfWeek`:

```
enum DayOfWeek {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;
}
```

An enum is compiled with a built-in static `valueOf()` method which can be used to lookup a constant by its name:

```
String dayName = DayOfWeek.SUNDAY.name();
assert dayName.equals("SUNDAY");

DayOfWeek day = DayOfWeek.valueOf(dayName);
assert day == DayOfWeek.SUNDAY;
```

This is also possible using a dynamic enum type:

```
Class<DayOfWeek> enumType = DayOfWeek.class;
DayOfWeek day = Enum.valueOf(enumType, "SUNDAY");
assert day == DayOfWeek.SUNDAY;
```

Both of these `valueOf()` methods will throw an `IllegalArgumentException` if the specified enum does not have a constant with a matching name.

The Guava library provides a helper method `Enums.getIfPresent()` that returns a Guava `Optional` to eliminate explicit exception handling:

```
DayOfWeek defaultDay = DayOfWeek.SUNDAY;
DayOfWeek day = Enums.valueOf(DayOfWeek.class, "INVALID").or(defaultDay);
assert day == DayOfWeek.SUNDAY;
```

Section 35.15: Enum with properties (fields)

In case we want to use **`enum`** with more information and not just as constant values, and we want to be able to compare two enums.

Consider the following example:

```
public enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);

    private final int value;

    Coin(int value){
        this.value = value;
    }

    public boolean isGreaterThan(Coin other){
        return this.value > other.value;
    }

}
```

Here we defined an **`Enum`** called `Coin` which represent its value. With the method `isGreaterThan` we can compare two **`enum`**s:

```
Coin penny = Coin.PENNY;
Coin dime = Coin.DIME;

System.out.println(penny.isGreaterThan(dime)); // prints: false
```

```
System.out.println(dime.isGreaterThan(penny)); // prints: true
```

Section 35.16: Convert enum to String

Sometimes you want to convert your enum to a String, there are two ways to do that.

Assume we have:

```
public enum Fruit {
    APPLE, ORANGE, STRAWBERRY, BANANA, LEMON, GRAPE_FRUIT;
}
```

So how do we convert something like `Fruit.APPLE` to `"APPLE"`?

Convert using `name()`

`name()` is an internal method in **`enum`** that returns the `String` representation of the enum, the return `String` represents ***exactly*** how the enum value was defined.

For example:

```
System.out.println(Fruit.BANANA.name());       // "BANANA"
System.out.println(Fruit.GRAPE_FRUIT.name()); // "GRAPE_FRUIT"
```

Convert using `toString()`

`toString()` is, *by default*, overridden to have the same behavior as `name()`

However, `toString()` is likely overridden by *developers* to make it print a more user friendly `String`

> Don't use `toString()` if you want to do checking in your code, `name()` is much more stable for that. Only use `toString()` when you are going to output the value to logs or stdout or something

By default:

```
System.out.println(Fruit.BANANA.toString());       // "BANANA"
System.out.println(Fruit.GRAPE_FRUIT.toString()); // "GRAPE_FRUIT"
```

Example of being overridden

```
System.out.println(Fruit.BANANA.toString());       // "Banana"
System.out.println(Fruit.GRAPE_FRUIT.toString()); // "Grape Fruit"
```

Section 35.17: Enums with static fields

If your enum class is required to have static fields, keep in mind they are created **after** the enum values themselves. That means, the following code will result in a `NullPointerException`:

```
enum Example {
    ONE(1), TWO(2);

    static Map<String, Integer> integers = new HashMap<>();

    private Example(int value) {
```

```
        integers.put(this.name(), value);
    }
}
```

A possible way to fix this:

```
enum Example {
    ONE(1), TWO(2);

    static Map<String, Integer> integers;

    private Example(int value) {
        putValue(this.name(), value);
    }

    private static void putValue(String name, int value) {
        if (integers == null)
            integers = new HashMap<>();
        integers.put(name, value);
    }
}
```

Do not initialize the static field:

```
enum Example {
    ONE(1), TWO(2);

    // after initialisisation integers is null!!
    static Map<String, Integer> integers = null;

    private Example(int value) {
        putValue(this.name(), value);
    }

    private static void putValue(String name, int value) {
        if (integers == null)
            integers = new HashMap<>();
        integers.put(name, value);
    }
    // !!this may lead to null poiner exception!!
    public int getValue(){
        return (Example.integers.get(this.name()));
    }
}
```

initialisisation:

- create the enum values
    - as side effect putValue() called that initializes integers
- the static values are set
    - integers = null; // is executed after the enums so the content of integers is lost

Chapter 36: Enum Map

Java EnumMap class is the specialized Map implementation for enum keys. It inherits Enum and AbstractMap classes.

the Parameters for java.util.EnumMap class.

K: It is the type of keys maintained by this map. V: It is the type of mapped values.

Section 36.1: Enum Map Book Example

```
import java.util.*;
class Book {
int id;
String name,author,publisher;
int quantity;
public Book(int id, String name, String author, String publisher, int quantity) {
    this.id = id;
    this.name = name;
    this.author = author;
    this.publisher = publisher;
    this.quantity = quantity;
}
}
public class EnumMapExample {
// Creating enum
    public enum Key{
           One, Two, Three
           };
public static void main(String[] args) {
    EnumMap<Key, Book> map = new EnumMap<Key, Book>(Key.class);
    // Creating Books
    Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
    Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
    Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
    // Adding Books to Map
       map.put(Key.One, b1);
       map.put(Key.Two, b2);
       map.put(Key.Three, b3);
    // Traversing EnumMap
       for(Map.Entry<Key, Book> entry:map.entrySet()){
            Book b=entry.getValue();
            System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
        }
}
}
```

Chapter 37: EnumSet class

Java EnumSet class is the specialized Set implementation for use with enum types. It inherits AbstractSet class and implements the Set interface.

Section 37.1: Enum Set Example

```
import java.util.*;
enum days {
  SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
public class EnumSetExample {
  public static void main(String[] args) {
    Set<days> set = EnumSet.of(days.TUESDAY, days.WEDNESDAY);
    // Traversing elements
    Iterator<days> iter = set.iterator();
    while (iter.hasNext())
      System.out.println(iter.next());
  }
}
```

Chapter 38: Enum starting with number

Java does not allow the name of enum to start with number like 100A, 25K. In that case, we can append the code with _ (underscore) or any allowed pattern and make check of it.

Section 38.1: Enum with name at beginning

```
public enum BookCode {
    _10A("Simon Haykin", "Communication System"),
    _42B("Stefan Hakins", "A Brief History of Time"),
    E1("Sedra Smith", "Electronics Circuits");

    private String author;
    private String title;

    BookCode(String author, String title) {
        this.author = author;
        this.title = title;
    }

    public String getName() {
        String name = name();
        if (name.charAt(0) == '_') {
            name = name.substring(1, name.length());
        }
        return name;
    }

    public static BookCode of(String code) {
        if (Character.isDigit(code.charAt(0))) {
            code = "_" + code;
        }
        return BookCode.valueOf(code);
    }
}
```

Chapter 39: Hashtable

Hashtable is a class in Java collections which implements Map interface and extends the Dictionary Class

Contains only unique elements and its synchronized

Section 39.1: Hashtable

```
import java.util.*;
public class HashtableDemo {
   public static void main(String args[]) {
   // create and populate hash table
   Hashtable<Integer, String> map = new Hashtable<Integer, String>();
   map.put(101,"C Language");
   map.put(102, "Domain");
   map.put(104, "Databases");
   System.out.println("Values before remove: "+ map);
   // Remove value for key 102
   map.remove(102);
   System.out.println("Values after remove: "+ map);
   }
}
```

Chapter 40: Operators

Operators in Java programming language are special symbols that perform specific operations on one, two, or three operands, and then return a result.

Section 40.1: The Increment/Decrement Operators (++/--)

Variables can be incremented or decremented by 1 using the `++` and `--` operators, respectively.

When the `++` and `--` operators follow variables, they are called **post-increment** and **post-decrement** respectively.

```
int a = 10;
a++; // a now equals 11
a--; // a now equals 10 again
```

When the `++` and `--` operators precede the variables the operations are called **pre-increment** and **pre-decrement** respectively.

```
int x = 10;
--x; // x now equals 9
++x; // x now equals 10
```

If the operator precedes the variable, the value of the expression is the value of the variable after being incremented or decremented. If the operator follows the variable, the value of the expression is the value of the variable prior to being incremented or decremented.

```
int x=10;

System.out.println("x=" + x + " x=" + x++ + " x=" + x); // outputs x=10 x=10 x=11
System.out.println("x=" + x + " x=" + ++x + " x=" + x); // outputs x=11 x=12 x=12
System.out.println("x=" + x + " x=" + x-- + " x=" + x); // outputs x=12 x=12 x=11
System.out.println("x=" + x + " x=" + --x + " x=" + x); // outputs x=11 x=10 x=10
```

Be careful not to overwrite post-increments or decrements. This happens if you use a post-in/decrement operator at the end of an expression which is reassigned to the in/decremented variable itself. The in/decrement will not have an effect. Even though the variable on the left hand side is incremented correctly, its value will be immediately overwritten with the previously evaluated result from the right hand side of the expression:

```
int x = 0;
x = x++ + 1 + x++;      // x = 0 + 1 + 1
                        // do not do this - the last increment has no effect (bug!)
System.out.println(x);  // prints 2 (not 3!)
```

Correct:

```
int x = 0;
x = x++ + 1 + x;        // evaluates to x = 0 + 1 + 1
x++;                    // adds 1
System.out.println(x);  // prints 3
```

Section 40.2: The Conditional Operator (? :)

Syntax

{condition-to-evaluate} **?** *{statement-executed-on-true}* **:** *{statement-executed-on-false}*

As shown in the syntax, the Conditional Operator (also known as the Ternary Operator1) uses the ? (question mark) and : (colon) characters to enable a conditional expression of two possible outcomes. It can be used to replace longer `if-else` blocks to return one of two values based on condition.

```
result = testCondition ? value1 : value2
```

Is equivalent to

```
if (testCondition) {
    result = value1;
} else {
    result = value2;
}
```

It can be read as "***If testCondition is true, set result to value1; otherwise, set result to value2***".

For example:

```
// get absolute value using conditional operator
a = -10;
int absValue = a < 0 ? -a : a;
System.out.println("abs = " + absValue); // prints "abs = 10"
```

Is equivalent to

```
// get absolute value using if/else loop
a = -10;
int absValue;
if (a < 0) {
    absValue = -a;
} else {
    absValue = a;
}
System.out.println("abs = " + absValue); // prints "abs = 10"
```

Common Usage

You can use the conditional operator for conditional assignments (like null checking).

```
String x = y != null ? y.toString() : ""; //where y is an object
```

This example is equivalent to:

```
String x = "";

if (y != null) {
    x = y.toString();
}
```

Since the Conditional Operator has the second-lowest precedence, above the Assignment Operators, there is rarely a need for use parenthesis around the *condition*, but parenthesis is required around the entire Conditional Operator construct when combined with other operators:

```
// no parenthesis needed for expressions in the 3 parts
```

```
10 <= a && a < 19 ? b * 5 : b * 7

// parenthesis required
7 * (a > 0 ? 2 : 5)
```

Conditional operators nesting can also be done in the third part, where it works more like chaining or like a switch statement.

```
a ? "a is true" :
b ? "a is false, b is true" :
c ? "a and b are false, c is true" :
    "a, b, and c are false"

//Operator precedence can be illustrated with parenthesis:

a ? x : (b ? y : (c ? z : w))
```

Footnote:

1 - Both the Java Language Specification and the Java Tutorial call the (`? :`) operator the *Conditional Operator*. The Tutorial says that it is "also known as the Ternary Operator" as it is (currently) the only ternary operator defined by Java. The "Conditional Operator" terminology is consistent with C and C++ and other languages with an equivalent operator.

Section 40.3: The Bitwise and Logical Operators (~, &, |, ^)

The Java language provides 4 operators that perform bitwise or logical operations on integer or boolean operands.

- The complement (~) operator is a unary operator that performs a bitwise or logical inversion of the bits of one operand; see JLS 15.15.5..
- The AND (&) operator is a binary operator that performs a bitwise or logical "and" of two operands; see JLS 15.22.2..
- The OR (|) operator is a binary operator that performs a bitwise or logical "inclusive or" of two operands; see JLS 15.22.2..
- The XOR (^) operator is a binary operator that performs a bitwise or logical "exclusive or" of two operands; see JLS 15.22.2..

The logical operations performed by these operators when the operands are booleans can be summarized as follows:

A	B	~A	A & B	A | B	A ^ B
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Note that for integer operands, the above table describes what happens for individual bits. The operators actually operate on all 32 or 64 bits of the operand or operands in parallel.

Operand types and result types.

The usual arithmetic conversions apply when the operands are integers. Common use-cases for the bitwise operators

The `~` operator is used to reverse a boolean value, or change all the bits in an integer operand.

The `&` operator is used for "masking out" some of the bits in an integer operand. For example:

```
int word = 0b00101010;
int mask = 0b00000011;   // Mask for masking out all but the bottom
                         // two bits of a word
int lowBits = word & mask;            // -> 0b00000010
int highBits = word & ~mask;          // -> 0b00101000
```

The `|` operator is used to combine the truth values of two operands. For example:

```
int word2 = 0b01011111;
// Combine the bottom 2 bits of word1 with the top 30 bits of word2
int combined = (word & mask) | (word2 & ~mask);   // -> 0b01011110
```

The `^` operator is used for toggling or "flipping" bits:

```
int word3 = 0b00101010;
int word4 = word3 ^ mask;             // -> 0b00101001
```

For more examples of the use of the bitwise operators, see Bit Manipulation

Section 40.4: The String Concatenation Operator (+)

The `+` symbol can mean three distinct operators in Java:

- If there is no operand before the `+`, then it is the unary Plus operator.
- If there are two operands, and they are both numeric. then it is the binary Addition operator.
- If there are two operands, and at least one of them is a `String`, then it it the binary Concatenation operator.

In the simple case, the Concatenation operator joins two strings to give a third string. For example:

```
String s1 = "a String";
String s2 = "This is " + s1;    // s2 contains "This is a String"
```

When one of the two operands is not a string, it is converted to a `String` as follows:

- An operand whose type is a primitive type is converted *as if* by calling `toString()` on the boxed value.
- An operand whose type is a reference type is converted by calling the operand's `toString()` method. If the operand is `null`, or if the `toString()` method returns `null`, then the string literal `"null"` is used instead.

For example:

```
int one = 1;
String s3 = "One is "  + one;         // s3 contains "One is 1"
String s4 = null + " is null";        // s4 contains "null is null"
String s5 = "{1} is " + new int[]{1}; // s5 contains something like
                                      // "{} is [I@xxxxxxxx"
```

The explanation for the `s5` example is that the `toString()` method on array types is inherited from `java.lang.Object`, and the behavior is to produce a string that consists of the type name, and the object's identity hashcode.

The Concatenation operator is specified to create a new `String` object, except in the case where the expression is a Constant Expression. In the latter case, the expression is evaluated at compile type, and its runtime value is

equivalent to a string literal. This means that there is no runtime overhead in splitting a long string literal like this:

```
String typing = "The quick brown fox " +
                "jumped over the " +
                "lazy dog";           // constant expression
```

Optimization and efficiency

As noted above, with the exception of constant expressions, each string concatenation expression creates a new `String` object. Consider this code:

```
public String stars(int count) {
    String res = "";
    for (int i = 0; i < count; i++) {
        res = res + "*";
    }
    return res;
}
```

In the method above, each iteration of the loop will create a new `String` that is one character longer than the previous iteration. Each concatenation copies all of the characters in the operand strings to form the new `String`. Thus, `stars(N)` will:

- create `N` new `String` objects, and throw away all but the last one,
- copy `N * (N + 1) / 2` characters, and
- generate `O(N^2)` bytes of garbage.

This is very expensive for large `N`. Indeed, any code that concatenates strings in a loop is liable to have this problem. A better way to write this would be as follows:

```
public String stars(int count) {
    // Create a string builder with capacity 'count'
    StringBuilder sb = new StringBuilder(count);
    for (int i = 0; i < count; i++) {
        sb.append("*");
    }
    return sb.toString();
}
```

Ideally, you should set the capacity of the `StringBuilder`, but if this is not practical, the class will automatically *grow* the backing array that the builder uses to hold characters. (Note: the implementation expands the backing array exponentially. This strategy keeps that amount of character copying to a `O(N)` rather than `O(N^2)`.)

Some people apply this pattern to all string concatenations. However, this is unnecessary because the JLS *allows* a Java compiler to optimize string concatenations within a single expression. For example:

```
String s1 = ...;
String s2 = ...;
String test = "Hello " + s1 + ". Welcome to " + s2 + "\n";
```

will *typically* be optimized by the bytecode compiler to something like this;

```
StringBuilder tmp = new StringBuilder();
tmp.append("Hello ")
tmp.append(s1 == null ? "null" + s1);
tmp.append("Welcome to ");
```

```
tmp.append(s2 == null ? "null" + s2);
tmp.append("\n");
String test = tmp.toString();
```

(The JIT compiler may optimize that further if it can deduce that `s1` or `s2` cannot be **`null`**.) But note that this optimization is only permitted within a single expression.

In short, if you are concerned about the efficiency of string concatenations:

- Do hand-optimize if you are doing repeated concatenation in a loop (or similar).
- Don't hand-optimize a single concatenation expression.

Section 40.5: The Arithmetic Operators (+, -, *, /, %)

The Java language provides 7 operators that perform arithmetic on integer and floating point values.

- There are two `+` operators:
  - The binary addition operator adds one number to another one. (There is also a binary `+` operator that performs string concatenation. That is described in a separate example.)
  - The unary plus operator does nothing apart from triggering numeric promotion (see below)
- There are two `-` operators:
  - The binary subtraction operator subtracts one number from another one.
  - The unary minus operator is equivalent to subtracting its operand from zero.
- The binary multiply operator (*) multiplies one number by another.
- The binary divide operator (/) divides one number by another.
- The binary remainder1 operator (%) calculates the remainder when one number is divided by another.

1. This is often incorrectly referred to as the "modulus" operator. "Remainder" is the term that is used by the JLS. "Modulus" and "remainder" are not the same thing.

Operand and result types, and numeric promotion

The operators require numeric operands and produce numeric results. The operand types can be any primitive numeric type (i.e. **`byte`**, **`short`**, **`char`**, **`int`**, **`long`**, **`float`** or **`double`**) or any numeric wrapper type define in `java.lang`; i.e. (`Byte`, `Character`, `Short`, `Integer`, `Long`, `Float` or `Double`.

The result type is determined base on the types of the operand or operands, as follows:

- If either of the operands is a **`double`** or `Double`, then the result type is **`double`**.
- Otherwise, if either of the operands is a **`float`** or `Float`, then the result type is **`float`**.
- Otherwise, if either of the operands is a **`long`** or `Long`, then the result type is **`long`**.
- Otherwise, the result type is **`int`**. This covers **`byte`**, **`short`** and **`char`** operands as well as `int.

The result type of the operation determines how the arithmetic operation is performed, and how the operands are handled

- If the result type is **`double`**, the operands are promoted to **`double`**, and the operation is performed using 64-bit (double precision binary) IEE 754 floating point arithmetic.
- If the result type is **`float`**, the operands are promoted to **`float`**, and the operation is performed using 32-bit (single precision binary) IEE 754 floating point arithmetic.
- If the result type is **`long`**, the operands are promoted to **`long`**, and the operation is performed using 64-bit signed twos-complement binary integer arithmetic.
- If the result type is **`int`**, the operands are promoted to **`int`**, and the operation is performed using 32-bit signed twos-complement binary integer arithmetic.

Promotion is performed in two stages:

- If the operand type is a wrapper type, the operand value is *unboxed* to a value of the corresponding primitive type.
- If necessary, the primitive type is promoted to the required type:
    - Promotion of integers to **`int`** or **`long`** is loss-less.
    - Promotion of **`float`** to **`double`** is loss-less.
    - Promotion of an integer to a floating point value can lead to loss of precision. The conversion is performed using IEE 768 "round-to-nearest" semantics.

The meaning of division

The / operator divides the left-hand operand `n` (the *dividend*) and the right-hand operand `d` (the *divisor*) and produces the result `q` (the *quotient*).

Java integer division rounds towards zero. The JLS Section 15.17.2 specifies the behavior of Java integer division as follows:

> The quotient produced for operands `n` and `d` is an integer value `q` whose magnitude is as large as possible while satisfying `|d · q| ≤ |n|`. Moreover, `q` is positive when `|n| ≥ |d|` and `n` and `d` have the same sign, but `q` is negative when `|n| ≥ |d|` and `n` and `d` have opposite signs.

There are a couple of special cases:

- If the `n` is `MIN_VALUE`, and the divisor is -1, then integer overflow occurs and the result is `MIN_VALUE`. No exception is thrown in this case.
- If `d` is 0, then `ArithmeticException is thrown.

Java floating point division has more edge cases to consider. However the basic idea is that the result `q` is the value that is closest to satisfying `d . q = n`.

Floating point division will never result in an exception. Instead, operations that divide by zero result in an INF and NaN values; see below.

The meaning of remainder

Unlike C and C++, the remainder operator in Java works with both integer and floating point operations.

For integer cases, the result of `a % b` is defined to be the number `r` such that `(a / b) * b + r` is equal to `a`, where `/`, `*` and `+` are the appropriate Java integer operators. This applies in all cases except when `b` is zero. That case, remainder results in an `ArithmeticException`.

It follows from the above definition that `a % b` can be negative only if `a` is negative, and it be positive only if `a` is positive. Moreover, the magnitude of `a % b` is always less than the magnitude of `b`.

Floating point remainder operation is a generalization of the integer case. The result of `a % b` is the remainder `r` is defined by the mathematical relation `r = a - (b · q)` where:

- `q` is an integer,
- it is negative only if `a / b` is negative an positive only if `a / b` is positive, and
- its magnitude is as large as possible without exceeding the magnitude of the true mathematical quotient of `a` and `b`.

Floating point remainder can produce `INF` and `NaN` values in edge-cases such as when `b` is zero; see below. It will not throw an exception.

Important note:

> The result of a floating-point remainder operation as computed by `%` **is not the same** as that produced by the remainder operation defined by IEEE 754. The IEEE 754 remainder may be computed using the `Math.IEEEremainder` library method.

Integer Overflow

Java 32 and 64 bit integer values are signed and use twos-complement binary representation. For example, the range of numbers representable as (32 bit) `int` -231 through +231 - 1.

When you add, subtract or multiple two N bit integers (N == 32 or 64), the result of the operation may be too large to represent as an N bit integer. In this case, the operation leads to *integer overflow*, and the result can be computed as follows:

- The mathematical operation is performed to give a intermediate two's-complement representation of the entire number. This representation will be larger than N bits.
- The bottom 32 or 64 bits of the intermediate representation are used as the result.

It should be noted that integer overflow does not result in exceptions under any circumstances.

Floating point INF and NAN values

Java uses IEE 754 floating point representations for `float` and `double`. These representations have some special values for representing values that fall outside of the domain of Real numbers:

- The "infinite" or INF values denote numbers that are too large. The `+INF` value denote numbers that are too large and positive. The `-INF` value denote numbers that are too large and negative.
- The "indefinite" / "not a number" or NaN denote values resulting from meaningless operations.

The INF values are produced by floating operations that cause overflow, or by division by zero.

The NaN values are produced by dividing zero by zero, or computing zero remainder zero.

Surprisingly, it is possible perform arithmetic using INF and NaN operands without triggering exceptions. For example:

- Adding +INF and a finite value gives +INF.
- Adding +INF and +INF gives +INF.
- Adding +INF and -INF gives NaN.
- Dividing by INF gives either +0.0 or -0.0.
- All operations with one or more NaN operands give NaN.

For full details, please refer to the relevant subsections of JLS 15. Note that this is largely "academic". For typical calculations, an `INF` or `NaN` means that something has gone wrong; e.g. you have incomplete or incorrect input data, or the calculation has been programmed incorrectly.

Section 40.6: The Shift Operators (<<, >> and >>>)

The Java language provides three operator for performing bitwise shifting on 32 and 64 bit integer values. These

are all binary operators with the first operand being the value to be shifted, and the second operand saying how far to shift.

- The `<<` or *left shift* operator shifts the value given by the first operand *leftwards* by the number of bit positions given by the second operand. The empty positions at the right end are filled with zeros.
- The '>>' or *arithmetic shift* operator shifts the value given by the first operand *rightwards* by the number of bit positions given by the second operand. The empty positions at the left end are filled by copying the left-most bit. This process is known as *sign extension*.
- The '>>>' or *logical right shift* operator shifts the value given by the first operand *rightwards* by the number of bit positions given by the second operand. The empty positions at the left end are filled with zeros.

Notes:

1. These operators require an **`int`** or **`long`** value as the first operand, and produce a value with the same type as the first operand. (You will need to use an explicit type cast when assigning the result of a shift to a **`byte`**, **`short`** or **`char`** variable.)
2. If you use a shift operator with a first operand that is a **`byte`**, **`char`** or **`short`**, it is promoted to an **`int`** and the operation produces an **`int`**.)
3. The second operand is reduced *modulo the number of bits of the operation* to give the amount of the shift. For more about the **mod mathematical concept**, see Modulus examples.
4. The bits that are shifted off the left or right end by the operation are discarded. (Java does not provide a primitive "rotate" operator.)
5. The arithmetic shift operator is equivalent dividing a (two's complement) number by a power of 2.
6. The left shift operator is equivalent multiplying a (two's complement) number by a power of 2.

The following table will help you see the effects of the three shift operators. (The numbers have been expressed in binary notation to aid vizualization.)

Operand1	Operand2	`<<`	`>>`	`>>>`
0b0000000000001011	0	0b0000000000001011	0b0000000000001011	0b0000000000001011
0b0000000000001011	1	0b0000000000010110	0b0000000000000101	0b0000000000000101
0b0000000000001011	2	0b0000000000101100	0b0000000000000010	0b0000000000000010
0b0000000000001011	28	0b1011000000000000	0b0000000000000000	0b0000000000000000
0b0000000000001011	31	0b1000000000000000	0b0000000000000000	0b0000000000000000
0b0000000000001011	32	0b0000000000001011	0b0000000000001011	0b0000000000001011
...	...	...	...	...
0b1000000000001011	0	0b1000000000001011	0b1000000000001011	0b1000000000001011
0b1000000000001011	1	0b0000000000010110	0b1100000000000101	0b0100000000000101
0b1000000000001011	2	0b0000000000101100	0b1110000000000010	0b0010000000000100
0b1000000000001011	31	0b1000000000000000	0b1111111111111111	0b0000000000000001

There examples of the user of shift operators in Bit manipulation

Section 40.7: The Instanceof Operator

This operator checks whether the object is of a particular class/interface type. **instanceof** operator is written as:

```
( Object reference variable ) instanceof  (class/interface type)
```

Example:

```
public class Test {

   public static void main(String args[]){
      String name = "Buyya";
      // following will return true since name is type of String
      boolean result = name instanceof String;
      System.out.println( result );
   }
}
```

This would produce the following result:

```
true
```

This operator will still return true if the object being compared is the assignment compatible with the type on the right.

Example:

```
class Vehicle {}

public class Car extends Vehicle {
   public static void main(String args[]){
      Vehicle a = new Car();
      boolean result =  a instanceof Car;
      System.out.println( result );
   }
}
```

This would produce the following result:

```
true
```

Section 40.8: The Assignment Operators (=, +=, -=, *=, /=, %=, <<=, >>= , >>>=, &=, |= and ^=)

The left hand operand for these operators must be a either a non-final variable or an element of an array. The right hand operand must be *assignment compatible* with the left hand operand. This means that either the types must be the same, or the right operand type must be convertible to the left operands type by a combination of boxing, unboxing or widening. (For complete details refer to JLS 5.2.)

The precise meaning of the "operation and assign" operators is specified by JLS 15.26.2 as:

> A compound assignment expression of the form `E1 op= E2` is equivalent to `E1 = (T) ((E1) op (E2))`, where `T` is the type of `E1`, except that `E1` is evaluated only once.

Note that there is an implicit type-cast before the final assignment.

1. =

The simple assignment operator: assigns the value of the right hand operand to the left hand operand.

> Example: `c = a + b` will add the value of `a + b` to the value of `c` and assign it to `c`

2. `+=`

The "add and assign" operator: adds the value of right hand operand to the value of the left hand operand and assigns the result to left hand operand. If the left hand operand has type `String`, then this a "concatenate and assign" operator.

> Example: `c += a` is roughly the same as `c = c + a`

3. `-=`

The "subtract and assign" operator: subtracts the value of the right operand from the value of the left hand operand and assign the result to left hand operand.

> Example: `c -= a` is roughly the same as `c = c - a`

4. `*=`

The "multiply and assign" operator: multiplies the value of the right hand operand by the value of the left hand operand and assign the result to left hand operand. .

> Example: `c *= a` is roughly the same as `c = c * a`

5. `/=`

The "divide and assign" operator: divides the value of the right hand operand by the value of the left hand operand and assign the result to left hand operand.

> Example: `c /*= a` is roughly the same as `c = c / a`

6. `%=`

The "modulus and assign" operator: calculates the modulus of the value of the right hand operand by the value of the left hand operand and assign the result to left hand operand.

> Example: `c %*= a` is roughly the same as `c = c % a`

7. `<<=`

The "left shift and assign" operator.

> Example: `c <<= 2` is roughly the same as `c = c << 2`

8. `>>=`

The "arithmetic right shift and assign" operator.

> Example: `c >>= 2` is roughly the same as `c = c >> 2`

9. `>>>=`

The "logical right shift and assign" operator.

> Example: `c >>>= 2` is roughly the same as `c = c >>> 2`

10. `&=`

The "bitwise and and assign" operator.

> Example: `c &= 2` is roughly the same as `c = c & 2`

11. `|=`

The "bitwise or and assign" operator.

> Example: `c |= 2` is roughly the same as `c = c | 2`

12. `^=`

The "bitwise exclusive or and assign" operator.

> Example: `c ^= 2` is roughly the same as `c = c ^ 2`

Section 40.9: The conditional-and and conditional-or Operators (&& and ||)

Java provides a conditional-and and a conditional-or operator, that both take one or two operands of type **`boolean`** and produce a **`boolean`** result. These are:

- `&&` - the conditional-AND operator,
- `||` - the conditional-OR operators. The evaluation of **`<left-expr>`** `&&` `<right-expr>` is equivalent to the following pseudo-code:

```
{
   boolean L = evaluate(<left-expr>);
   if (L) {
       return evaluate(<right-expr>);
   } else {
       // short-circuit the evaluation of the 2nd operand expression
       return false;
   }
```

```
}
```

The evaluation of **<left-expr> || <right-expr>** is equivalent to the following pseudo-code:

```
{
   boolean L = evaluate(<left-expr>);
   if (!L) {
       return evaluate(<right-expr>);
   } else {
       // short-circuit the evaluation of the 2nd operand expression
       return true;
   }
}
```

As the pseudo-code above illustrates, the behavior of the short-circuit operators are equivalent to using `if` / **`else`** statements.

Example - using && as a guard in an expression

The following example shows the most common usage pattern for the `&&` operator. Compare these two versions of a method to test if a supplied `Integer` is zero.

```
public boolean isZero(Integer value) {
    return value == 0;
}

public boolean isZero(Integer value) {
    return value != null && value == 0;
}
```

The first version works in most cases, but if the `value` argument is **`null`**, then a `NullPointerException` will be thrown.

In the second version we have added a "guard" test. The `value != null && value == 0` expression is evaluated by first performing the `value != null` test. If the **`null`** test succeeds (i.e. it evaluates to **`true`**) then the `value == 0` expression is evaluated. If the **`null`** test fails, then the evaluation of `value == 0` is skipped (short-circuited), and we *don't* get a `NullPointerException`.

Example - using && to avoid a costly calculation

The following example shows how `&&` can be used to avoid a relatively costly calculation:

```
public boolean verify(int value, boolean needPrime) {
    return !needPrime | isPrime(value);
}

public boolean verify(int value, boolean needPrime) {
    return !needPrime || isPrime(value);
}
```

In the first version, both operands of the `|` will always be evaluated, so the (expensive) `isPrime` method will be called unnecessarily. The second version avoids the unnecessary call by using `||` instead of `|`.

Section 40.10: The Relational Operators (<, <=, >, >=)

The operators `<`, `<=`, `>` and `>=` are binary operators for comparing numeric types. The meaning of the operators is as

you would expect. For example, if `a` and `b` are declared as any of **`byte`**, **`short`**, **`char`**, **`int`**, **`long`**, **`float`**, **`double`** or the corresponding boxed types:

```
- `a < b` tests if the value of `a` is less than the value of `b`.
- `a <= b` tests if the value of `a` is less than or equal to the value of `b`.
- `a > b` tests if the value of `a` is greater than the value of `b`.
- `a >= b` tests if the value of `a` is greater than or equal to the value of `b`.
```

The result type for these operators is **`boolean`** in all cases.

Relational operators can be used to compare numbers with different types. For example:

```
int i = 1;
long l = 2;
if (i < l) {
    System.out.println("i is smaller");
}
```

Relational operators can be used when either or both numbers are instances of boxed numeric types. For example:

```
Integer i = 1;   // 1 is autoboxed to an Integer
Integer j = 2;   // 2 is autoboxed to an Integer
if (i < j) {
    System.out.println("i is smaller");
}
```

The precise behavior is summarized as follows:

1. If one of the operands is a boxed type, it is unboxed.
2. If either of the operands now a **`byte`**, **`short`** or **`char`**, it is promoted to an **`int`**.
3. If the types of the operands are not the same, then the operand with the "smaller" type is promoted to the "larger" type.
4. The comparison is performed on the resulting **`int`**, **`long`**, **`float`** or **`double`** values.

You need to be careful with relational comparisons that involve floating point numbers:

- Expressions that compute floating point numbers often incur rounding errors due to the fact that the computer floating-point representations have limited precision.
- When comparing an integer type and a floating point type, the conversion of the integer to floating point can also lead to rounding errors.

Finally, Java does bit support the use of relational operators with any types other than the ones listed above. For example, you *cannot* use these operators to compare strings, arrays of numbers, and so on.

Section 40.11: The Equality Operators (==, !=)

The `==` and `!=` operators are binary operators that evaluate to **`true`** or **`false`** depending on whether the operands are equal. The `==` operator gives **`true`** if the operands are equal and **`false`** otherwise. The `!=` operator gives **`false`** if the operands are equal and **`true`** otherwise.

These operators can be used operands with primitive and reference types, but the behavior is significantly different. According to the JLS, there are actually three distinct sets of these operators:

- The Boolean `==` and `!=` operators.
- The Numeric `==` and `!=` operators.

- The Reference `==` and `!=` operators.

However, in all cases, the result type of the `==` and `!=` operators is **`boolean`**.

The Numeric `==` and `!=` operators

When one (or both) of the operands of an `==` or `!=` operator is a primitive numeric type (**`byte`**, **`short`**, **`char`**, **`int`**, **`long`**, **`float`** or **`double`**), the operator is a numeric comparison. The second operand must be either a primitive numeric type, or a boxed numeric type.

The behavior other numeric operators is as follows:

1. If one of the operands is a boxed type, it is unboxed.
2. If either of the operands now a **`byte`**, **`short`** or **`char`**, it is promoted to an **`int`**.
3. If the types of the operands are not the same, then the operand with the "smaller" type is promoted to the "larger" type.
4. The comparison is then carried out as follows:
    - If the promoted operands are **`int`** or **`long`** then the values are tested to see if they are identical.
    - If the promoted operands are **`float`** or **`double`** then:
        - the two versions of zero (`+0.0` and `-0.0`) are treated as equal
        - a `NaN` value is treated as not equals to anything, and
        - other values are equal if their IEEE 754 representations are identical.

Note: you need to be careful when using `==` and `!=` to compare floating point values.

The Boolean `==` and `!=` operators

If both operands are **`boolean`**, or one is **`boolean`** and the other is `Boolean`, these operators the Boolean `==` and `!=` operators. The behavior is as follows:

1. If one of the operands is a `Boolean`, it is unboxed.
2. The unboxed operands are tested and the boolean result is calculated according to the following truth table

A	B	A == B	A != B
false	false	true	false
false	true	false	true
true	false	false	true
true	true	true	false

There are two "pitfalls" that make it advisable to use `==` and `!=` sparingly with truth values:

- If you use `==` or `!=` to compare two `Boolean` objects, then the Reference operators are used. This may give an unexpected result; see Pitfall: using == to compare primitive wrappers objects such as Integer
- The `==` operator can easily be mistyped as `=`. For most operand types, this mistake leads to a compilation error. However, for **`boolean`** and `Boolean` operands the mistake leads to incorrect runtime behavior; see Pitfall - Using '==' to test a boolean

The Reference `==` and `!=` operators

If both operands are object references, the `==` and `!=` operators test if the two operands **refer to the same object**. This often not what you want. To test if two objects are equal *by value*, the `.equals()` method should be used instead.

```
String s1 = "We are equal";
String s2 = new String("We are equal");

s1.equals(s2); // true

// WARNING - don't use == or != with String values
s1 == s2;      // false
```

Warning: using `==` and `!=` to compare `String` values is **incorrect** in most cases; see http://stackoverflow.com/documentation/java/4388/java-pitfalls/16290/using-to-compare-strings . A similar problem applies to primitive wrapper types; see http://stackoverflow.com/documentation/java/4388/java-pitfalls/8996/using-to-compare-primitive-wrappers-objects-such-as-integer .

About the NaN edge-cases

JLS 15.21.1 states the following:

> If either operand is `NaN`, then the result of `==` is **`false`** but the result of `!=` is **`true`**. Indeed, the test `x != x` is **`true`** if and only if the value of `x` is `NaN`.

This behavior is (to most programmers) unexpected. If you test if a `NaN` value is equal to itself, the answer is "No it isn't!". In other words, `==` is not *reflexive* for `NaN` values.

However, this is not a Java "oddity", this behavior is specified in the IEEE 754 floating-point standards, and you will find that it is implemented by most modern programming languages. (For more information, see http://stackoverflow.com/a/1573715/139985 ... noting that this is written by someone who was "in the room when the decisions were made"!)

Section 40.12: The Lambda operator (->)

From Java 8 onwards, the Lambda operator (`->`) is the operator used to introduce a Lambda Expression. There are two common syntaxes, as illustrated by these examples:

Version ≥ Java SE 8

```
  a -> a + 1              // a lambda that adds one to its argument
  a -> { return a + 1; }  // an equivalent lambda using a block.
```

A lambda expression defines an anonymous function, or more correctly an instance of an anonymous class that implements a *functional interface*.

(This example is included here for completeness. Refer to the Lambda Expressions topic for the full treatment.)

Chapter 41: Constructors

While not required, constructors in Java are methods recognized by the compiler to instantiate specific values for the class which may be essential to the role of the object. This topic demonstrates proper usage of Java class constructors.

Section 41.1: Default Constructor

The "default" for constructors is that they do not have any arguments. In case you do not specify **any** constructor, the compiler will generate a default constructor for you.
This means the following two snippets are semantically equivalent:

```
public class TestClass {
    private String test;
}
```

```
public class TestClass {
    private String test;
    public TestClass() {

    }
}
```

The visibility of the default constructor is the same as the visibility of the class. Thus a class defined package-privately has a package-private default constructor

However, if you have non-default constructor, the compiler will not generate a default constructor for you. So these are not equivalent:

```
public class TestClass {
    private String test;
    public TestClass(String arg) {
    }
}
```

```
public class TestClass {
    private String test;
    public TestClass() {
    }
    public TestClass(String arg) {
    }
}
```

Beware that the generated constructor performs no non-standard initialization. This means all fields of your class will have their default value, unless they have an initializer.

```
public class TestClass {

    private String testData;

    public TestClass() {
        testData = "Test"
    }
}
```

Constructors are called like this:

```
TestClass testClass = new TestClass();
```

Section 41.2: Call parent constructor

Say you have a Parent class and a Child class. To construct a Child instance always requires some Parent constructor to be run at the very gebinning of the Child constructor. We can select the Parent constructor we want by explicitly calling `super(...)` with the appropriate arguments as our first Child constructor statement. Doing this saves us time by reusing the Parent classes' constructor instead of rewriting the same code in the Child classes' constructor.

Without `super(...)` method:

(implicitly, the no-args version `super()` is called invisibly)

```
class Parent {
    private String name;
    private int age;

    public Parent() {} // necessary because we call super() without arguments

    public Parent(String tName, int tAge) {
        name = tName;
        age = tAge;
    }
}

// This does not even compile, because name and age are private,
// making them invisible even to the child class.
class Child extends Parent {
    public Child() {
        // compiler implicitly calls super() here
        name = "John";
        age = 42;
    }
}
```

With `super()` method:

```
class Parent {
    private String name;
    private int age;
    public Parent(String tName, int tAge) {
        name = tName;
        age = tAge;
    }
}

class Child extends Parent {
    public Child() {
        super("John", 42);   // explicit super-call
    }
}
```

Note: Calls to another constructor (chaining) or the super constructor **MUST** be the first statement inside the constructor.

If you call the `super(...)` constructor explicitly, a matching parent constructor must exist (that's straightforward, isn't it?).

If you don't call any `super(...)` constructor explicitly, your parent class must have a no-args constructor - and this can be either written explicitly or created as a default by the compiler if the parent class doesn't provide any constructor.

```
class Parent{
    public Parent(String tName, int tAge) {}
}

class Child extends Parent{
    public Child(){}
}
```

The class Parent has no default constructor, so, the compiler can't add `super` in the Child constructor. This code will not compile. You must change the constructors to fit both sides, or write your own `super` call, like that:

```
class Child extends Parent{
    public Child(){
          super("",0);
    }
}
```

Section 41.3: Constructor with Arguments

Constructors can be created with any kinds of arguments.

```
public class TestClass {

    private String testData;

    public TestClass(String testData) {
        this.testData = testData;
    }
}
```

Called like this:

```
TestClass testClass = new TestClass("Test Data");
```

A class can have multiple constructors with different signatures. To chain constructor calls (call a different constructor of the same class when instantiating) use `this()`.

```
public class TestClass {

    private String testData;

    public TestClass(String testData) {
        this.testData = testData;
    }

    public TestClass() {
        this("Test"); // testData defaults to "Test"
    }
}
```

Called like this:

```
TestClass testClass1 = new TestClass("Test Data");
```

```
TestClass testClass2 = new TestClass();
```

Chapter 42: Object Class Methods and Constructor

This documentation page is for showing details with example about java class constructors and about Object Class Methods which are automatically inherited from the superclass `Object` of any newly created class.

Section 42.1: hashCode() method

When a Java class overrides the `equals` method, it should override the `hashCode` method as well. As defined in the method's contract:

> - Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
> - If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
> - It is not required that if two objects are unequal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

Hash codes are used in hash implementations such as `HashMap`, `HashTable`, and `HashSet`. The result of the `hashCode` function determines the bucket in which an object will be put. These hash implementations are more efficient if the provided `hashCode` implementation is good. An important property of good `hashCode` implementation is that the distribution of the `hashCode` values is uniform. In other words, there is a small probability that numerous instances will be stored in the same bucket.

An algorithm for computing a hash code value may be similar to the following:

```
public class Foo {
    private int field1, field2;
    private String field3;

    public Foo(int field1, int field2, String field3) {
        this.field1 = field1;
        this.field2 = field2;
        this.field3 = field3;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null || getClass() != obj.getClass()) {
            return false;
        }

        Foo f = (Foo) obj;
        return field1 == f.field1 &&
               field2 == f.field2 &&
               (field3 == null ? f.field3 == null : field3.equals(f.field3);
    }
```

```
    @Override
    public int hashCode() {
        int hash = 1;
        hash = 31 * hash + field1;
        hash = 31 * hash + field2;
        hash = 31 * hash + (field3 == null ? 0 : field3.hashCode());
        return hash;
    }
}
```

Using Arrays.hashCode() as a short cut

Version ≥ Java SE 1.2

In Java 1.2 and above, instead of developing an algorithm to compute a hash code, one can be generated using `java.util.Arrays#hashCode` by supplying an Object or primitives array containing the field values:

```
@Override
public int hashCode() {
    return Arrays.hashCode(new Object[] {field1, field2, field3});
}
```

Version ≥ Java SE 7

Java 1.7 introduced the `java.util.Objects` class which provides a convenience method, `hash(Object... objects)`, that computes a hash code based on the values of the objects supplied to it. This method works just like `java.util.Arrays#hashCode`.

```
@Override
public int hashCode() {
    return Objects.hash(field1, field2, field3);
}
```

Note: this approach is inefficient, and produces garbage objects each time your custom `hashCode()` method is called:

- A temporary `Object[]` is created. (In the `Objects.hash()` version, the array is created by the "varargs" mechanism.)
- If any of the fields are primitive types, they must be boxed and that may create more temporary objects.
- The array must be populated.
- The array must iterated by the `Arrays.hashCode` or `Objects.hash` method.
- The calls to `Object.hashCode()` that `Arrays.hashCode` or `Objects.hash` has to make (probably) cannot be inlined.

Internal caching of hash codes

Since the calculation of an object's hash code can be expensive, it can be attractive to cache the hash code value within the object the first time that it is calculated. For example

```
public final class ImmutableArray {
    private int[] array;
    private volatile int hash = 0;

    public ImmutableArray(int[] initial) {
        array = initial.clone();
    }

    // Other methods
```

```
    @Override
    public boolean equals(Object obj) {
         // ...
    }

    @Override
    public int hashCode() {
        int h = hash;
        if (h == 0) {
            h = Arrays.hashCode(array);
            hash = h;
        }
        return h;
    }
}
```

This approach trades off the cost of (repeatedly) calculating the hash code against the overhead of an extra field to cache the hash code. Whether this pays off as a performance optimization will depend on how often a given object is hashed (looked up) and other factors.

You will also notice that if the true hashcode of an `ImmutableArray` happens to be zero (one chance in 232), the cache is ineffective.

Finally, this approach is much harder to implement correctly if the object we are hashing is mutable. However, there are bigger concerns if hash codes change; see the contract above.

Section 42.2: toString() method

The `toString()` method is used to create a `String` representation of an object by using the object´s content. This method should be overridden when writing your class. `toString()` is called implicitly when an object is concatenated to a string as in `"hello " + anObject`.

Consider the following:

```
public class User {
    private String firstName;
    private String lastName;

    public User(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Override
    public String toString() {
        return firstName + " " + lastName;
    }

    public static void main(String[] args) {
        User user = new User("John", "Doe");
        System.out.println(user.toString()); // Prints "John Doe"
    }
}
```

Here `toString()` from `Object` class is overridden in the `User` class to provide meaningful data regarding the object when printing it.

When using `println()`, the object's `toString()` method is implicitly called. Therefore, these statements do the

same thing:

```
System.out.println(user); // toString() is implicitly called on `user`
System.out.println(user.toString());
```

If the `toString()` is not overridden in the above mentioned `User` class, `System.out.println(user)` may return `User@659e0bfd` or a similar String with almost no useful information except the class name. This will be because the call will use the `toString()` implementation of the base Java `Object` class which does not know anything about the `User` class's structure or business rules. If you want to change this functionality in your class, simply override the method.

Section 42.3: equals() method

TL;DR

`==` tests for reference equality (whether they are the ***same object***)

`.equals()` tests for value equality (whether they are ***logically "equal"***)

`equals()` is a method used to compare two objects for equality. The default implementation of the `equals()` method in the `Object` class returns **`true`** if and only if both references are pointing to the same instance. It therefore behaves the same as comparison by `==`.

```
public class Foo {
    int field1, field2;
    String field3;

    public Foo(int i, int j, String k) {
        field1 = i;
        field2 = j;
        field3 = k;
    }

    public static void main(String[] args) {
        Foo foo1 = new Foo(0, 0, "bar");
        Foo foo2 = new Foo(0, 0, "bar");

        System.out.println(foo1.equals(foo2)); // prints false
    }
}
```

Even though `foo1` and `foo2` are created with the same fields, they are pointing to two different objects in memory. The default `equals()` implementation therefore evaluates to **`false`**.

To compare the contents of an object for equality, `equals()` has to be overridden.

```
public class Foo {
    int field1, field2;
    String field3;

    public Foo(int i, int j, String k) {
        field1 = i;
        field2 = j;
        field3 = k;
    }

    @Override
```

```
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null || getClass() != obj.getClass()) {
            return false;
        }

        Foo f = (Foo) obj;
        return field1 == f.field1 &&
               field2 == f.field2 &&
               (field3 == null ? f.field3 == null : field3.equals(f.field3));
    }

    @Override
    public int hashCode() {
        int hash = 1;
        hash = 31 * hash + this.field1;
        hash = 31 * hash + this.field2;
        hash = 31 * hash + (field3 == null ? 0 : field3.hashCode());
        return hash;
    }

    public static void main(String[] args) {
        Foo foo1 = new Foo(0, 0, "bar");
        Foo foo2 = new Foo(0, 0, "bar");

        System.out.println(foo1.equals(foo2)); // prints true
    }
}
```

Here the overridden `equals()` method decides that the objects are equal if their fields are the same.

Notice that the `hashCode()` method was also overwritten. The contract for that method states that when two objects are equal, their hash values must also be the same. That's why one must almost always override `hashCode()` and `equals()` together.

Pay special attention to the argument type of the `equals` method. It is `Object obj`, not `Foo obj`. If you put the latter in your method, that is not an override of the `equals` method.

When writing your own class, you will have to write similar logic when overriding `equals()` and `hashCode()`. Most IDEs can automatically generate this for you.

An example of an `equals()` implementation can be found in the `String` class, which is part of the core Java API. Rather than comparing pointers, the `String` class compares the content of the `String`.

Version ≥ Java SE 7

Java 1.7 introduced the `java.util.Objects` class which provides a convenience method, `equals`, that compares two potentially `null` references, so it can be used to simplify implementations of the `equals` method.

```
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null || getClass() != obj.getClass()) {
        return false;
    }
```

```
    Foo f = (Foo) obj;
    return field1 == f.field1 && field2 == f.field2 && Objects.equals(field3, f.field3);
}
```

Class Comparison

Since the equals method can run against any object, one of the first things the method often does (after checking for `null`) is to check if the class of the object being compared matches the current class.

```
@Override
public boolean equals(Object obj) {
    //...check for null
    if (getClass() != obj.getClass()) {
        return false;
    }
    //...compare fields
}
```

This is typically done as above by comparing the class objects. However, that can fail in a few special cases which may not be obvious. For example, some frameworks generate dynamic proxies of classes and these dynamic proxies are actually a different class. Here is an example using JPA.

```
Foo detachedInstance = ...
Foo mergedInstance = entityManager.merge(detachedInstance);
if (mergedInstance.equals(detachedInstance)) {
    //Can never get here if equality is tested with getClass()
    //as mergedInstance is a proxy (subclass) of Foo
}
```

One mechanism to work around that limitation is to compare classes using `instanceof`

```
@Override
public final boolean equals(Object obj) {
    if (!(obj instanceof Foo)) {
        return false;
    }
    //...compare fields
}
```

However, there are a few pitfalls that must be avoided when using `instanceof`. Since Foo could potentially have other subclasses and those subclasses might override `equals()` you could get into a case where a `Foo` is equal to a `FooSubclass` but the `FooSubclass` is not equal to `Foo`.

```
Foo foo = new Foo(7);
FooSubclass fooSubclass = new FooSubclass(7, false);
foo.equals(fooSubclass) //true
fooSubclass.equals(foo) //false
```

This violates the properties of symmetry and transitivity and thus is an invalid implementation of the `equals()` method. As a result, when using `instanceof`, a good practice is to make the `equals()` method `final` (as in the above example). This will ensure that no subclass overrides `equals()` and violates key assumptions.

Section 42.4: wait() and notify() methods

`wait()` and `notify()` work in tandem – when one thread calls `wait()` on an object, that thread will block until another thread calls `notify()` or `notifyAll()` on that same object.

(See Also: wait()/notify())

```
package com.example.examples.object;

import java.util.concurrent.atomic.AtomicBoolean;

public class WaitAndNotify {

    public static void main(String[] args) throws InterruptedException {
        final Object obj = new Object();
        AtomicBoolean aHasFinishedWaiting = new AtomicBoolean(false);

        Thread threadA = new Thread("Thread A") {
            public void run() {
                System.out.println("A1: Could print before or after B1");
                System.out.println("A2: Thread A is about to start waiting...");
                try {
                    synchronized (obj) { // wait() must be in a synchronized block
                        // execution of thread A stops until obj.notify() is called
                        obj.wait();
                    }
                    System.out.println("A3: Thread A has finished waiting. "
                            + "Guaranteed to happen after B3");
                } catch (InterruptedException e) {
                    System.out.println("Thread A was interrupted while waiting");
                } finally {
                    aHasFinishedWaiting.set(true);
                }
            }
        };

        Thread threadB = new Thread("Thread B") {
            public void run() {
                System.out.println("B1: Could print before or after A1");

                System.out.println("B2: Thread B is about to wait for 10 seconds");
                for (int i = 0; i < 10; i++) {
                    try {
                        Thread.sleep(1000); // sleep for 1 second
                    } catch (InterruptedException e) {
                        System.err.println("Thread B was interrupted from waiting");
                    }
                }

                System.out.println("B3: Will ALWAYS print before A3 since "
                        + "A3 can only happen after obj.notify() is called.");

                while (!aHasFinishedWaiting.get()) {
                    synchronized (obj) {
                        // notify ONE thread which has called obj.wait()
                        obj.notify();
                    }
                }
            }
        };

        threadA.start();
        threadB.start();

        threadA.join();
        threadB.join();
```

```
        System.out.println("Finished!");
    }
}
```

Some example output:

```
A1: Could print before or after B1
B1: Could print before or after A1
A2: Thread A is about to start waiting...
B2: Thread B is about to wait for 10 seconds
B3: Will ALWAYS print before A3 since A3 can only happen after obj.notify() is called.
A3: Thread A has finished waiting. Guaranteed to happen after B3
Finished!

B1: Could print before or after A1
B2: Thread B is about to wait for 10 seconds
A1: Could print before or after B1
A2: Thread A is about to start waiting...
B3: Will ALWAYS print before A3 since A3 can only happen after obj.notify() is called.
A3: Thread A has finished waiting. Guaranteed to happen after B3
Finished!

A1: Could print before or after B1
A2: Thread A is about to start waiting...
B1: Could print before or after A1
B2: Thread B is about to wait for 10 seconds
B3: Will ALWAYS print before A3 since A3 can only happen after obj.notify() is called.
A3: Thread A has finished waiting. Guaranteed to happen after B3
Finished!
```

Section 42.5: getClass() method

The `getClass()` method can be used to find the runtime class type of an object. See the example below:

```
public class User {

    private long userID;
    private String name;

    public User(long userID, String name) {
        this.userID = userID;
        this.name = name;
    }
}

public class SpecificUser extends User {
    private String specificUserID;

    public SpecificUser(String specificUserID, long userID, String name) {
        super(userID, name);
        this.specificUserID = specificUserID;
    }
}

public static void main(String[] args){
    User user = new User(879745, "John");
    SpecificUser specificUser = new SpecificUser("1AAAA", 877777, "Jim");
    User anotherSpecificUser = new SpecificUser("1BBBB", 812345, "Jenny");

    System.out.println(user.getClass()); //Prints "class User"
```

```
    System.out.println(specificUser.getClass()); //Prints "class SpecificUser"
    System.out.println(anotherSpecificUser.getClass()); //Prints "class SpecificUser"
}
```

The `getClass()` method will return the most specific class type, which is why when `getClass()` is called on `anotherSpecificUser`, the return value is **`class`** `SpecificUser` because that is lower down the inheritance tree than `User`.

It is noteworthy that, while the `getClass` method is declared as:

```
public final native Class<?> getClass();
```

The actual static type returned by a call to `getClass` is `Class<? extends T>` where `T` is the static type of the object on which `getClass` is called.

i.e. the following will compile:

```
Class<? extends String> cls = "".getClass();
```

Section 42.6: clone() method

The `clone()` method is used to create and return a copy of an object. This method arguable should be avoided as it is problematic and a copy constructor or some other approach for copying should be used in favour of `clone()`.

For the method to be used all classes calling the method must implement the `Cloneable` interface.

The `Cloneable` interface itself is just a tag interface used to change the behaviour of the **`native`** `clone()` method which checks if the calling objects class implements `Cloneable`. If the caller does not implement this interface a `CloneNotSupportedException` will be thrown.

The `Object` class itself does not implement this interface so a `CloneNotSupportedException` will be thrown if the calling object is of class `Object`.

For a clone to be correct it should be independent of the object it is being cloned from, therefore it may be necessary to modify the object before it gets returned. This means to essentially create a "deep copy" by also copying any of the *mutable* objects that make up the internal structure of the object being cloned. If this is not implemented correctly the cloned object will not be independent and have the same references to the mutable objects as the object that it was cloned from. This would result in inconsistent behaviour as any changes to those in one would affect the other.

```
class Foo implements Cloneable {
    int w;
    String x;
    float[] y;
    Date z;

    public Foo clone() {
        try {
            Foo result = new Foo();
            // copy primitives by value
            result.w = this.w;
            // immutable objects like String can be copied by reference
            result.x = this.x;

            // The fields y and z refer to a mutable objects; clone them recursively.
            if (this.y != null) {
```

```
            result.y = this.y.clone();
        }
        if (this.z != null) {
            result.z = this.z.clone();
        }

        // Done, return the new object
        return result;

    } catch (CloneNotSupportedException e) {
        // in case any of the cloned mutable fields do not implement Cloneable
        throw new AssertionError(e);
    }
  }
}
```

Section 42.7: Object constructor

All constructors in Java must make a call to the `Object` constructor. This is done with the call `super()`. This has to be the first line in a constructor. The reason for this is so that the object can actually be created on the heap before any additional initialization is performed.

If you do not specify the call to `super()` in a constructor the compiler will put it in for you.

So all three of these examples are functionally identical

with explicit call to `super()` constructor

```
public class MyClass {

    public MyClass() {
        super();
    }
}
```

with implicit call to `super()` constructor

```
public class MyClass {

    public MyClass() {
        // empty
    }
}
```

with implicit constructor

```
public class MyClass {

}
```

> What about Constructor-Chaining?

It is possible to call other constructors as the first instruction of a constructor. As both the explicit call to a super constructor and the call to another constructor have to be both first instructions, they are mutually exclusive.

```
public class MyClass {
```

```
    public MyClass(int size) {

        doSomethingWith(size);

    }

    public MyClass(Collection<?> initialValues) {

        this(initialValues.size());
        addInitialValues(initialValues);
    }
}
```

Calling new `MyClass(Arrays.asList("a", "b", "c"))` will call the second constructor with the List-argument, which will in turn delegate to the first constructor (which will delegate implicitly to `super()`) and then call `addInitialValues(int size)` with the second size of the list. This is used to reduce code duplication where multiple constructors need to do the same work.

> How do I call a specific constructor?

Given the example above, one can either call `new MyClass("argument")` or `new MyClass("argument", 0)`. In other words, much like method overloading, you just call the constructor with the parameters that are necessary for your chosen constructor.

> What will happen in the Object class constructor?

Nothing more than would happen in a sub-class that has a default empty constructor (minus the call to `super()`).

The default empty constructor can be explicitly defined but if not the compiler will put it in for you as long as no other constructors are already defined.

> How is an Object then created from the constructor in Object?

The actual creation of objects is down to the JVM. Every constructor in Java appears as a special method named `<init>` which is responsible for instance initializing. This `<init>` method is supplied by the compiler and because `<init>` is not a valid identifier in Java, it cannot be used directly in the language.

> How does the JVM invoke this `<init>` method?

The JVM will invoke the `<init>` method using the `invokespecial` instruction and can only be invoked on uninitialized class instances.

For more information take a look at the JVM specification and the Java Language Specification:

- Special Methods (JVM) - JVMS - 2.9
- Constructors - JLS - 8.8

Section 42.8: finalize() method

This is a *protected* and *non-static* method of the `Object` class. This method is used to perform some final operations

or clean up operations on an object before it gets removed from the memory.

> According to the doc, this method gets called by the garbage collector on an object when garbage collection determines that there are no more references to the object.

But there are no guarantees that `finalize()` method would gets called if the object is still reachable or no Garbage Collectors run when the object become eligible. That's why it's better **not rely** on this method.

In Java core libraries some usage examples could be found, for instance in `FileInputStream.java`:

```
protected void finalize() throws IOException {
    if ((fd != null) &&  (fd != FileDescriptor.in)) {
        /* if fd is shared, the references in FileDescriptor
         * will ensure that finalizer is only called when
         * safe to do so. All references using the fd have
         * become unreachable. We can call close()
         */
        close();
    }
}
```

In this case it's the last chance to close the resource if that resource has not been closed before.

Generally it's considered bad practice to use `finalize()` method in applications of any kind and should be avoided.

Finalizers are *not* meant for freeing resources (e.g., closing files). The garbage collector gets called when (if!) the system runs low on heap space. You can't rely on it to be called when the system is running low on file handles or, for any other reason.

The intended use-case for finalizers is for an object that is about to be reclaimed to notify some other object about its impending doom. A better mechanism now exists for that purpose---the `java.lang.ref.WeakReference<T>` class. If you think you need write a `finalize()` method, then you should look into whether you can solve the same problem using `WeakReference` instead. If that won't solve your problem, then you may need to re-think your design on a deeper level.

For further reading here is an Item about `finalize()` method from "Effective Java" book by Joshua Bloch.

Chapter 43: Annotations

In Java, an annotation is a form of syntactic metadata that can be added to Java source code. It provides data about a program that is not part of the program itself. Annotations have no direct effect on the operation of the code they annotate. Classes, methods, variables, parameters and packages are allowed to be annotated.

Section 43.1: The idea behind Annotations

The Java Language Specification describes Annotations as follows:

> An annotation is a marker which associates information with a program construct, but has no effect at run time.

Annotations may appear before types or declarations. It is possible for them to appear in a place where they could apply to both a type or a declaration.
What exactly an annotation applies to is governed by the "meta-annotation" `@Target`. See "Defining annotation types" for more information.

Annotations are used for a multitude of purposes. Frameworks like Spring and Spring-MVC make use of annotations to define where Dependencies should be injected or where requests should be routed.

Other frameworks use annotations for code-generation. Lombok and JPA are prime examples, that use annotations to generate Java (and SQL) code.

This topic aims to provide a comprehensive overview of:

- How to define your own Annotations?
- What Annotations does the Java Language provide?
- How are Annotations used in practice?

Section 43.2: Defining annotation types

Annotation types are defined with **`@interface`**. Parameters are defined similar to methods of a regular interface.

```
@interface MyAnnotation {
    String param1();
    boolean param2();
    int[] param3();  // array parameter
}
```

Default values

```
@interface MyAnnotation {
    String param1() default "someValue";
    boolean param2() default true;
    int[] param3() default {};
}
```

Meta-Annotations

Meta-annotations are annotations that can be applied to annotation types. Special predefined meta-annotation define how annotation types can be used.

@Target

The `@Target` meta-annotation restricts the types the annotation can be applied to.

```
@Target(ElementType.METHOD)
@interface MyAnnotation {
    // this annotation can only be applied to methods
}
```

Multiple values can be added using array notation, e.g. `@Target({ElementType.FIELD, ElementType.TYPE})`

Available Values

ElementType	target	example usage on target element
ANNOTATION_TYPE	annotation types	`@Retention(RetentionPolicy.RUNTIME)` terface MyAnnotation
CONSTRUCTOR	constructors	`@MyAnnotation`lic MyClass() {}
FIELD	fields, enum constants	`@XmlAttribute`vate int count;

LOCAL_VARIABLE	variable declarations inside methods	`for (@LoopVariable int i = 0; i < 100; i++) {`
		@Unused
		String resultVariable;
		}

PACKAGE	package (in `package-info.java`)	`@Deprecated`kage very.old;
METHOD	methods	`@XmlElement`lic int getCount() {...}

PARAMETER	method/constructor parameters	`public Rectangle(` @NamedArg("width") double width, @NamedArg("height") double height) {
		...
		}

TYPE	classes, interfaces, enums	`@XmlRootElement`lic class Report {}

`Version ≥ Java SE 8`

ElementType	target	example usage on target element
TYPE_PARAMETER	Type parameter declarations	`public <@MyAnnotation T> void f(T t) {}`
TYPE_USE	Use of a type	`Object o = "42";`ing s = (@MyAnnotation String) o;

@Retention

The `@Retention` meta-annotation defines the annotation visibility during the applications compilation process or execution. By default, annotations are included in `.class` files, but are not visible at runtime. To make an annotation accessible at runtime, `RetentionPolicy.RUNTIME` has to be set on that annotation.

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation {
    // this annotation can be accessed with reflections at runtime
}
```

Available values

RetentionPolicy	Effect
CLASS	The annotation is available in the `.class` file, but not at runtime
RUNTIME	The annotation is available at runtime and can be accessed via reflection
SOURCE	The annotation is available at compile time, but not added to the `.class` files. The annotation can be used e.g. by an annotation processor.

@Documented

The `@Documented` meta-annotation is used to mark annotations whose usage should be documented by API documentation generators like javadoc. It has no values. With `@Documented`, all classes that use the annotation will list it on their generated documentation page. Without `@Documented`, it's not possible to see which classes use the

annotation in the documentation.

@Inherited

The `@Inherited` meta-annotation is relevant to annotations that are applied to classes. It has no values. Marking an annotation as `@Inherited` alters the way that annotation querying works.

- For a non-inherited annotation, the query only examines the class being examined.
- For an inherited annotation, the query will also check the super-class chain (recursively) until an instance of the annotation is found.

Note that only the super-classes are queried: any annotations attached to interfaces in the classes hierarchy will be ignored.

@Repeatable

The `@Repeatable` meta-annotation was added in Java 8. It indicates that multiple instances of the annotation can be attached to the annotation's target. This meta-annotation has no values.

Section 43.3: Runtime annotation checks via reflection

Java's Reflection API allows the programmer to perform various checks and operations on class fields, methods and annotations during runtime. However, in order for an annotation to be at all visible at runtime, the `RetentionPolicy` must be changed to `RUNTIME`, as demonstrated in the example below:

```
@interface MyDefaultAnnotation {

}

@Retention(RetentionPolicy.RUNTIME)
@interface MyRuntimeVisibleAnnotation {

}

public class AnnotationAtRuntimeTest {

    @MyDefaultAnnotation
    static class RuntimeCheck1 {
    }

    @MyRuntimeVisibleAnnotation
    static class RuntimeCheck2 {
    }

    public static void main(String[] args) {
        Annotation[] annotationsByType = RuntimeCheck1.class.getAnnotations();
        Annotation[] annotationsByType2 = RuntimeCheck2.class.getAnnotations();

        System.out.println("default retention: " + Arrays.toString(annotationsByType));
        System.out.println("runtime retention: " + Arrays.toString(annotationsByType2));
    }
}
```

Section 43.4: Built-in annotations

The Standard Edition of Java comes with some annotations predefined. You do not need to define them by yourself and you can use them immediately. They allow the compiler to enable some fundamental checking of methods,

classes and code.

@Override

This annotation applies to a method and says that this method must override a superclass' method or implement an abstract superclass' method definition. If this annotation is used with any other kind of method, the compiler will throw an error.

Concrete superclass

```
public class Vehicle {
   public void drive() {
        System.out.println("I am driving");
   }
}

class Car extends Vehicle {
    // Fine
    @Override
    public void drive() {
        System.out.prinln("Brrrm, brrm");
    }
}
```

Abstract class

```
abstract class Animal  {
   public abstract void makeNoise();
}

class Dog extends Animal {
    // Fine
    @Override
    public void makeNoise() {
        System.out.prinln("Woof");
    }
}
```

Does not work

```
class Logger1 {
    public void log(String logString) {
        System.out.prinln(logString);
    }
}

class Logger2 {
    // This will throw compile-time error. Logger2 is not a subclass of Logger1.
    // log method is not overriding anything
    @Override
    public void log(String logString) {
        System.out.println("Log 2" + logString);
    }
}
```

The main purpose is to catch mistyping, where you think you are overriding a method, but are actually defining a new one.

```
class Vehicle {
```

```
    public void drive() {
        System.out.println("I am driving");
    }
}

class Car extends Vehicle {
    // Compiler error. "dirve" is not the correct method name to override.
    @Override
    public void dirve() {
        System.out.prinln("Brrrm, brrm");
    }
}
```

Note that the meaning of `@Override` has changed over time:

- In Java 5, it meant that the annotated method had to override a non-abstract method declared in the superclass chain.
- From Java 6 onward, it is *also* satisfied if the annotated method implements an abstract method declared in the classes superclass / interface hierarchy.

(This can occasionally cause problems when back-porting code to Java 5.)

@Deprecated

This marks the method as deprecated. There can be several reasons for this:

- the API is flawed and is impractical to fix,
- usage of the API is likely to lead to errors,
- the API has been superseded by another API,
- the API is obsolete,
- the API is experimental and is subject to incompatible changes,
- or any combination of the above.

The specific reason for deprecation can usually be found in the documentation of the API.

The annotation will cause the compiler to emit an error if you use it. IDEs may also highlight this method somehow as deprecated

```
class ComplexAlgorithm {
    @Deprecated
    public void oldSlowUnthreadSafeMethod() {
        // stuff here
    }

    public void quickThreadSafeMethod() {
        // client code should use this instead
    }
}
```

@SuppressWarnings

In almost all cases, when the compiler emits a warning, the most appropriate action is to fix the cause. In some instances (Generics code using untype-safe pre-generics code, for example) this may not be possible and it's better to suppress those warnings that you expect and cannot fix, so you can more clearly see unexpected warnings.

This annotation can be applied to a whole class, method or line. It takes the category of warning as a parameter.

```
@SuppressWarnings("deprecation")
public class RiddledWithWarnings {
    // several methods calling deprecated code here
}

@SuppressWarning("finally")
public boolean checkData() {
    // method calling return from within finally block
}
```

It is better to limit the scope of the annotation as much as possible, to prevent unexpected warnings also being suppressed. For example, confining the scope of the annotation to a single-line:

```
ComplexAlgorithm algorithm = new ComplexAlgorithm();
@SuppressWarnings("deprecation") algoritm.slowUnthreadSafeMethod();
// we marked this method deprecated in an example above

@SuppressWarnings("unsafe") List<Integer> list = getUntypeSafeList();
// old library returns, non-generic List containing only integers
```

The warnings supported by this annotation may vary from compiler to compiler. Only the `unchecked` and `deprecation` warnings are specifically mentioned in the JLS. Unrecognized warning types will be ignored.

@SafeVarargs

Because of type erasure, `void method(T... t)` will be converted to `void method(Object[] t)` meaning that the compiler is not always able to verify that the use of varargs is type-safe. For instance:

```
private static <T> void generatesVarargsWarning(T... lists) {
```

There are instances where the use is safe, in which case you can annotate the method with the `SafeVarargs` annotation to suppress the warning. This obviously hides the warning if your use is unsafe too.

@FunctionalInterface

This is an optional annotation used to mark a FunctionalInterface. It will cause the compiler to complain if it does not conform to the FunctionalInterface spec (has a single abstract method)

```
@FunctionalInterface
public interface ITrade {
  public boolean check(Trade t);
}

@FunctionalInterface
public interface Predicate<T> {
  boolean test(T t);
}
```

Section 43.5: Compile time processing using annotation processor

This example demonstrates how to do compile time checking of an annotated element.

The annotation

The `@Setter` annotation is a marker can be applied to methods. The annotation will be discarded during compilation not be available afterwards.

```
package annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.METHOD)
public @interface Setter {
}
```

The annotation processor

The `SetterProcessor` class is used by the compiler to process the annotations. It checks, if the methods annotated with the `@Setter` annotation are **`public`**, non-**`static`** methods with a name starting with `set` and having a uppercase letter as 4th letter. If one of these conditions isn't met, a error is written to the `Messager`. The compiler writes this to stderr, but other tools could use this information differently. E.g. the NetBeans IDE allows the user specify annotation processors that are used to display error messages in the editor.

```
package annotation.processor;

import annotation.Setter;
import java.util.Set;
import javax.annotation.processing.AbstractProcessor;
import javax.annotation.processing.Messager;
import javax.annotation.processing.ProcessingEnvironment;
import javax.annotation.processing.RoundEnvironment;
import javax.annotation.processing.SupportedAnnotationTypes;
import javax.annotation.processing.SupportedSourceVersion;
import javax.lang.model.SourceVersion;
import javax.lang.model.element.Element;
import javax.lang.model.element.ElementKind;
import javax.lang.model.element.ExecutableElement;
import javax.lang.model.element.Modifier;
import javax.lang.model.element.TypeElement;
import javax.tools.Diagnostic;

@SupportedAnnotationTypes({"annotation.Setter"})
@SupportedSourceVersion(SourceVersion.RELEASE_8)
public class SetterProcessor extends AbstractProcessor {

    private Messager messager;

    @Override
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {
        // get elements annotated with the @Setter annotation
        Set<? extends Element> annotatedElements = roundEnv.getElementsAnnotatedWith(Setter.class);

        for (Element element : annotatedElements) {
            if (element.getKind() == ElementKind.METHOD) {
                // only handle methods as targets
                checkMethod((ExecutableElement) element);
            }
        }

        // don't claim annotations to allow other processors to process them
        return false;
```

```
    }

    private void checkMethod(ExecutableElement method) {
        // check for valid name
        String name = method.getSimpleName().toString();
        if (!name.startsWith("set")) {
            printError(method, "setter name must start with \"set\"");
        } else if (name.length() == 3) {
            printError(method, "the method name must contain more than just \"set\"");
        } else if (Character.isLowerCase(name.charAt(3))) {
            if (method.getParameters().size() != 1) {
                printError(method, "character following \"set\" must be upper case");
            }
        }

        // check, if setter is public
        if (!method.getModifiers().contains(Modifier.PUBLIC)) {
            printError(method, "setter must be public");
        }

        // check, if method is static
        if (method.getModifiers().contains(Modifier.STATIC)) {
            printError(method, "setter must not be static");
        }
    }

    private void printError(Element element, String message) {
        messager.printMessage(Diagnostic.Kind.ERROR, message, element);
    }

    @Override
    public void init(ProcessingEnvironment processingEnvironment) {
        super.init(processingEnvironment);

        // get messager for printing errors
        messager = processingEnvironment.getMessager();
    }

}
```

Packaging

To be applied by the compiler, the annotation processor needs to be made available to the SPI (see ServiceLoader).

To do this a text file `META-INF/services/javax.annotation.processing.Processor` needs to be added to the jar file containing the annotation processor and the annotation in addition to the other files. The file needs to include the fully qualified name of the annotation processor, i.e. it should look like this

```
annotation.processor.SetterProcessor
```

We'll assume the jar file is called `AnnotationProcessor.jar` below.

Example annotated class

The following class is example class in the default package with the annotations being applied to the correct elements according to the retention policy. However only the annotation processor only considers the second method a valid annotation target.

```
import annotation.Setter;
```

```
public class AnnotationProcessorTest {

    @Setter
    private void setValue(String value) {}

    @Setter
    public void setString(String value) {}

    @Setter
    public static void main(String[] args) {}

}
```

Using the annotation processor with javac

If the annotation processor is discovered using the SPI, it is automatically used to process annotated elements. E.g. compiling the `AnnotationProcessorTest` class using

```
javac -cp AnnotationProcessor.jar AnnotationProcessorTest.java
```

yields the following output

```
AnnotationProcessorTest.java:6: error: setter must be public
private void setValue(String value) {}
^
AnnotationProcessorTest.java:12: error: setter name must start with "set"
public static void main(String[] args) {}
^
2 errors
```

instead of compiling normally. No **`.class`** file is created.

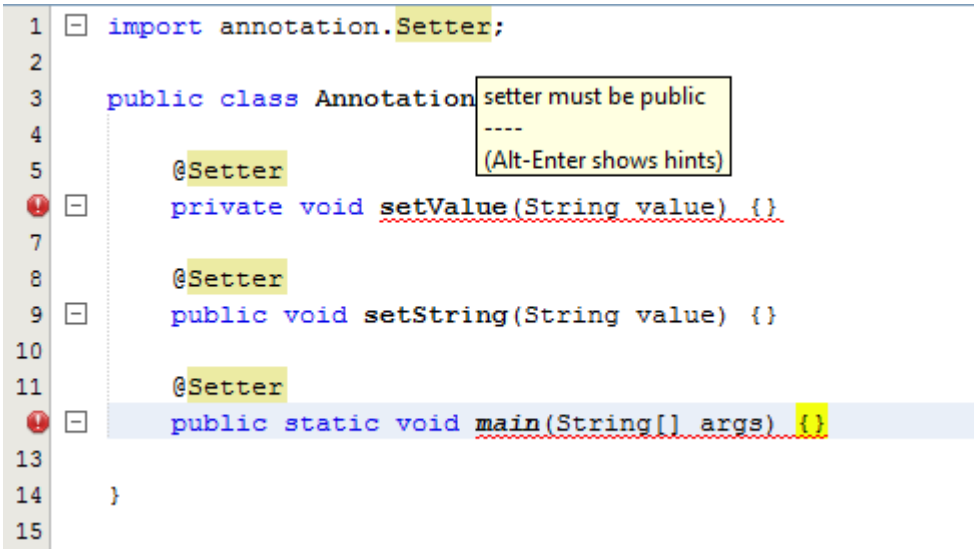
This could be prevented by specifying the `-proc:none` option for `javac`. You could also forgo the usual compilation by specifying `-proc:only` instead.

IDE integration
Netbeans

Annotation processors can be used in the NetBeans editor. To do this the annotation processor needs to be specified in the project settings:

1. go to `Project Properties` > `Build` > `Compiling`
2. add check marks for `Enable Annotation Processing` and `Enable Annotation Processing in Editor`
3. click `Add` next to the annotation processor list
4. in the popup that appears enter the fully qualified class name of the annotation processor and click `Ok`.

Result



Section 43.6: Repeating Annotations

Until Java 8, two instances of the same annotation could not be applied to a single element. The standard workaround was to use a container annotation holding an array of some other annotation:

```
// Author.java
@Retention(RetentionPolicy.RUNTIME)
public @interface Author {
    String value();
}

// Authors.java
@Retention(RetentionPolicy.RUNTIME)
public @interface Authors {
    Author[] value();
}

// Test.java
@Authors({
    @Author("Mary"),
    @Author("Sam")
})
public class Test {
    public static void main(String[] args) {
        Author[] authors = Test.class.getAnnotation(Authors.class).value();
        for (Author author : authors) {
            System.out.println(author.value());
            // Output:
            // Mary
            // Sam
        }
    }
}
```

Version ≥ Java SE 8

Java 8 provides a cleaner, more transparent way of using container annotations, using the `@Repeatable` annotation. First we add this to the `Author` class:

```
@Repeatable(Authors.class)
```

This tells Java to treat multiple `@Author` annotations as though they were surrounded by the `@Authors` container. We can also use `Class.getAnnotationsByType()` to access the `@Author` array by its own class, instead of through its

container:

```
@Author("Mary")
@Author("Sam")
public class Test {
    public static void main(String[] args) {
        Author[] authors = Test.class.getAnnotationsByType(Author.class);
        for (Author author : authors) {
            System.out.println(author.value());
            // Output:
            // Mary
            // Sam
        }
    }
}
```

Section 43.7: Inherited Annotations

By default class annotations do not apply to types extending them. This can be changed by adding the `@Inherited` annotation to the annotation definition

Example

Consider the following 2 Annotations:

```
@Inherited
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface InheritedAnnotationType {
}
```

and

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface UninheritedAnnotationType {
}
```

If three classes are annotated like this:

```
@UninheritedAnnotationType
class A {
}

@InheritedAnnotationType
class B extends A {
}

class C extends B {
}
```

running this code

```
System.out.println(new A().getClass().getAnnotation(InheritedAnnotationType.class));
System.out.println(new B().getClass().getAnnotation(InheritedAnnotationType.class));
System.out.println(new C().getClass().getAnnotation(InheritedAnnotationType.class));
System.out.println("_____________________________________");
System.out.println(new A().getClass().getAnnotation(UninheritedAnnotationType.class));
```

```
System.out.println(new B().getClass().getAnnotation(UninheritedAnnotationType.class));
System.out.println(new C().getClass().getAnnotation(UninheritedAnnotationType.class));
```

will print a result similar to this (depending on the packages of the annotation):

```
null
@InheritedAnnotationType()
@InheritedAnnotationType()
---------------------------------
@UninheritedAnnotationType()
null
null
```

Note that annotations can only be inherited from classes, not interfaces.

Section 43.8: Getting Annotation values at run-time

You can fetch the current properties of the Annotation by using Reflection to fetch the Method or Field or Class which has an Annotation applied to it, and then fetching the desired properties.

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation {
    String key() default "foo";
    String value() default "bar";
}


class AnnotationExample {
    // Put the Annotation on the method, but leave the defaults
    @MyAnnotation
    public void testDefaults() throws Exception {
        // Using reflection, get the public method "testDefaults", which is this method with no args
        Method method = AnnotationExample.class.getMethod("testDefaults", null);

        // Fetch the Annotation that is of type MyAnnotation from the Method
        MyAnnotation annotation = (MyAnnotation)method.getAnnotation(MyAnnotation.class);

        // Print out the settings of the Annotation
        print(annotation);
    }

    //Put the Annotation on the method, but override the settings
    @MyAnnotation(key="baz", value="buzz")
    public void testValues() throws Exception {
        // Using reflection, get the public method "testValues", which is this method with no args
        Method method = AnnotationExample.class.getMethod("testValues", null);

        // Fetch the Annotation that is of type MyAnnotation from the Method
        MyAnnotation annotation = (MyAnnotation)method.getAnnotation(MyAnnotation.class);

        // Print out the settings of the Annotation
        print(annotation);
    }

    public void print(MyAnnotation annotation) {
        // Fetch the MyAnnotation 'key' & 'value' properties, and print them out
        System.out.println(annotation.key() + " = " + annotation.value());
    }

    public static void main(String[] args) {
```

```
        AnnotationExample example = new AnnotationExample();
        try {
            example.testDefaults();
            example.testValues();
        } catch( Exception e ) {
            // Shouldn't throw any Exceptions
            System.err.println("Exception [" + e.getClass().getName() + "] - " + e.getMessage());
            e.printStackTrace(System.err);
        }
    }
}
```

The output will be

```
foo = bar
baz = buzz
```

Section 43.9: Annotations for 'this' and receiver parameters

When Java annotations were first introduced there was no provision for annotating the target of an instance method or the hidden constructor parameter for an inner classes constructor. This was remedied in Java 8 with addition of *receiver parameter* declarations; see JLS 8.4.1.

> The receiver parameter is an optional syntactic device for an instance method or an inner class's constructor. For an instance method, the receiver parameter represents the object for which the method is invoked. For an inner class's constructor, the receiver parameter represents the immediately enclosing instance of the newly constructed object. Either way, the receiver parameter exists solely to allow the type of the represented object to be denoted in source code, so that the type may be annotated. The receiver parameter is not a formal parameter; more precisely, it is not a declaration of any kind of variable (§4.12.3), it is never bound to any value passed as an argument in a method invocation expression or qualified class instance creation expression, and it has no effect whatsoever at run time.

The following example illustrates the syntax for both kinds of receiver parameter:

```
public class Outer {
    public class Inner {
        public Inner (Outer this) {
           // ...
        }
        public void doIt(Inner this) {
           // ...
        }
    }
}
```

The sole purpose of receiver parameters is to allow you to add annotations. For example, you might have a custom annotation `@IsOpen` whose purpose is to assert that a `Closeable` object has not been closed when a method is called. For example:

```
public class MyResource extends Closeable {
    public void update(@IsOpen MyResource this, int value) {
        // ...
    }
```

```
    public void close() {
        // ...
    }
}
```

At one level, the `@IsOpen` annotation on **`this`** could simply serve as documentation. However, we could potentially do more. For example:

- An annotation processor could insert a runtime check that **`this`** is not in closed state when `update` is called.
- A code checker could perform a static code analysis to find cases where **`this`** *could be* closed when `update` is called.

Section 43.10: Add multiple annotation values

An Annotation parameter can accept multiple values if it is defined as an array. For example the standard annotation `@SuppressWarnings` is defined like this:

```
public @interface SuppressWarnings {
    String[] value();
}
```

The `value` parameter is an array of Strings. You can set multiple values by using a notation similar to Array initializers:

```
@SuppressWarnings({"unused"})
@SuppressWarnings({"unused", "javadoc"})
```

If you only need to set a single value, the brackets can be omitted:

```
@SuppressWarnings("unused")
```

Chapter 44: Immutable Class

Immutable objects are instances whose state doesn't change after it has been initialized. For example, String is an immutable class and once instantiated its value never changes.

Section 44.1: Example without mutable refs

```
public final class Color {
    final private int red;
    final private int green;
    final private int blue;

    private void check(int red, int green, int blue) {
        if (red < 0 || red > 255 || green < 0 || green > 255 || blue < 0 || blue > 255) {
            throw new IllegalArgumentException();
        }
    }

    public Color(int red, int green, int blue) {
        check(red, green, blue);
        this.red = red;
        this.green = green;
        this.blue = blue;
    }

    public Color invert() {
        return new Color(255 - red, 255 - green, 255 - blue);
    }
}
```

Section 44.2: What is the advantage of immutability?

The advantage of immutability comes with concurrency. It is difficult to maintain correctness in mutable objects, as multiple threads could be trying to change the state of the same object, leading to some threads seeing a different state of the same object, depending on the timing of the reads and writes to the said object.

By having an immutable object, one can ensure that all threads that are looking at the object will be seeing the same state, as the state of an immutable object will not change.

Section 44.3: Rules to define immutable classes

The following rules define a simple strategy for creating immutable objects.

1. Don't provide "setter" methods - methods that modify fields or objects referred to by fields.
2. Make all fields final and private.
3. Don't allow subclasses to override methods. The simplest way to do this is to declare the class as final. A more sophisticated approach is to make the constructor private and construct instances in factory methods.
4. If the instance fields include references to mutable objects, don't allow those objects to be changed:
5. Don't provide methods that modify the mutable objects.
6. Don't share references to the mutable objects. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.

Section 44.4: Example with mutable refs

In this case class Point is mutable and some user can modify state of object of this class.

```
class Point {
    private int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }

    public int getY() {
        return y;
    }

    public void setY(int y) {
        this.y = y;
    }
}

//...

public final class ImmutableCircle {
    private final Point center;
    private final double radius;

    public ImmutableCircle(Point center, double radius) {
        // we create new object here because it shouldn't be changed
        this.center = new Point(center.getX(), center.getY());
        this.radius = radius;
    }
```

Chapter 45: Immutable Objects

Section 45.1: Creating an immutable version of a type using defensive copying

Some basic types and classes in Java are fundamentally mutable. For example, all array types are mutable, and so are classes like `java.util.Data`. This can be awkward in situations where an immutable type is mandated.

One way to deal with this is to create an immutable wrapper for the mutable type. Here is a simple wrapper for an array of integers

```
public class ImmutableIntArray {
    private final int[] array;

    public ImmutableIntArray(int[] array) {
        this.array = array.clone();
    }

    public int[] getValue() {
        return this.clone();
    }
}
```

This class works by using *defensive copying* to isolate the mutable state (the `int[]`) from any code that might mutate it:

- The constructor uses `clone()` to create a distinct copy of the parameter array. If the caller of the constructor subsequent changed the parameter array, it would not affect the state of the `ImmutableIntArray`.
- The `getValue()` method also uses `clone()` to create the array that is returned. If the caller were to change the result array, it would not affect the state of the `ImmutableIntArray`.

We could also add methods to `ImmutableIntArray` to perform read-only operations on the wrapped array; e.g. get its length, get the value at a particular index, and so on.

Note that an immutable wrapper type implemented this way is not type compatible with the original type. You cannot simply substitute the former for the latter.

Section 45.2: The recipe for an immutable class

An immutable object is an object whose state cannot be changed. An immutable class is a class whose instances are immutable by design, and implementation. The Java class which is most commonly presented as an example of immutability is java.lang.String.

The following is a stereotypical example:

```
public final class Person {
    private final String name;
    private final String ssn;     // (SSN == social security number)

    public Person(String name, String ssn) {
        this.name = name;
        this.ssn = ssn;
    }
```

```
    public String getName() {
        return name;
    }

    public String getSSN() {
        return ssn;
    }
}
```

A variation on this is to declare the constructor as `private` and provide a `public static` factory method instead.

The *standard recipe* for an immutable class is as follows:

- All properties must be set in the constructor(s) or factory method(s).
- There should be no setters.
- If it is necessary to include setters for interface compatibility reasons, they should either do nothing or throw an exception.
- All properties should be declared as `private` and `final`.
- For all properties that are references to mutable types:
    - the property should be initialized with a deep copy of the value passed via the constructor, and
    - the property's getter should return a deep copy of the property value.
- The class should be declared as `final` to prevent someone creating a mutable subclass of an immutable class.

A couple of other things to note:

- Immutability does not prevent object from being nullable; e.g. `null` can be assigned to a `String` variable.
- If an immutable classes properties are declared as `final`, instances are inherently thread-safe. This makes immutable classes a good building block for implementing multi-threaded applications.

Section 45.3: Typical design flaws which prevent a class from being immutable

Using some setters, without setting all needed properties in the constructor(s)

```
public final class Person { // example of a bad immutability
    private final String name;
    private final String surname;
    public Person(String name) {
        this.name = name;
      }
    public String getName() { return name;}
    public String getSurname() { return surname;}
    public void setSurname(String surname) { this.surname = surname); }
}
```

It's easy to show that `Person` class is not immutable:

```
Person person = new Person("Joe");
person.setSurname("Average"); // NOT OK, change surname field after creation
```

To fix it, simply delete `setSurname()` and refactor the constructor as follows:

```
public Person(String name, String surname) {
    this.name = name;
    this.surname = surname;
```

```
  }
```

Not marking instance variables as private and final

Take a look at the following class:

```
public final class Person {
    public String name;
    public Person(String name) {
        this.name = name;
     }
    public String getName() {
        return name;
    }

}
```

The following snippet shows that the above class is not immutable:

```
Person person = new Person("Average Joe");
person.name = "Magic Mike"; // not OK, new name for person after creation
```

To fix it, simply mark name property as **`private`** and **`final`**.

Exposing a mutable object of the class in a getter

Take a look at the following class:

```
import java.util.List;
import java.util.ArrayList;
public final class Names {
    private final List<String> names;
    public Names(List<String> names) {
        this.names = new ArrayList<String>(names);
    }
    public List<String> getNames() {
        return names;
    }
    public int size() {
        return names.size();
    }
}
```

`Names` class seems immutable at the first sight, but it is not as the following code shows:

```
List<String> namesList = new ArrayList<String>();
namesList.add("Average Joe");
Names names = new Names(namesList);
System.out.println(names.size()); // 1, only containing "Average Joe"
namesList = names.getNames();
namesList.add("Magic Mike");
System.out.println(names.size()); // 2, NOT OK, now names also contains "Magic Mike"
```

This happened because a change to the reference List returned by `getNames()` can modify the actual list of `Names`.

To fix this, simply avoid returning references that reference class's mutable objects *either* by making defensive copies, as follows:

```
public List<String> getNames() {
   return new ArrayList<String>(this.names); // copies elements
}
```

or by designing getters in way that only other *immutable objects* and *primitives* are returned, as follows:

```
public String getName(int index) {
    return names.get(index);
}
public int size() {
    return names.size();
}
```

Injecting constructor with object(s) that can be modified outside the immutable class

This is a variation of the previous flaw. Take a look at the following class:

```
import java.util.List;
public final class NewNames {
    private final List<String> names;
    public Names(List<String> names) {
        this.names = names;
    }
    public String getName(int index) {
        return names.get(index);
    }
    public int size() {
        return names.size();
    }
}
```

As `Names` class before, also `NewNames` class seems immutable at the first sight, but it is not, in fact the following snippet proves the contrary:

```
List<String> namesList = new ArrayList<String>();
namesList.add("Average Joe");
NewNames names = new NewNames(namesList);
System.out.println(names.size()); // 1, only containing "Average Joe"
namesList.add("Magic Mike");
System.out.println(names.size()); // 2, NOT OK, now names also contains "Magic Mike"
```

To fix this, as in the previous flaw, simply make defensive copies of the object without assigning it directly to the immutable class, i.e. constructor can be changed as follows:

```
    public Names(List<String> names) {
        this.names = new ArrayList<String>(names);
    }
```

Letting the methods of the class being overridden

Take a look at the following class:

```
public class Person {
    private final String name;
    public Person(String name) {
        this.name = name;
      }
    public String getName() { return name;}
```

```
}
```

`Person` class seems immutable at the first sight, but suppose a new subclass of `Person` is defined:

```
public class MutablePerson extends Person {
    private String newName;
    public MutablePerson(String name) {
        super(name);
    }
    @Override
    public String getName() {
        return newName;
    }
    public void setName(String name) {
        newName = name;
    }
}
```

now `Person` (im)mutability can be exploited through polymorphism by using the new subclass:

```
Person person = new MutablePerson("Average Joe");
System.out.println(person.getName()); prints Average Joe
person.setName("Magic Mike"); // NOT OK, person has now a new name!
System.out.println(person.getName()); // prints Magic Mike
```

To fix this, *either* mark the class as **`final`** so it cannot be extended *or* declare all of its constructor(s) as **`private`**.

Chapter 46: Visibility (controlling access to members of a class)

Section 46.1: Private Visibility

`private` visibility allows a variable to only be accessed by its class. They are often used in **conjunction** with `public` getters and setters.

```
class SomeClass {
    private int variable;

    public int getVariable() {
        return variable;
    }

    public void setVariable(int variable) {
        this.variable = variable;
    }
}

public class SomeOtherClass {
    public static void main(String[] args) {
        SomeClass sc = new SomeClass();

        // These statement won't compile because SomeClass#variable is private:
        sc.variable = 7;
        System.out.println(sc.variable);

        // Instead, you should use the public getter and setter:
        sc.setVariable(7);
        System.out.println(sc.getVariable());
    }
}
```

Section 46.2: Public Visibility

Visible to the class, package, and subclass.

Let's see an example with the class Test.

```
public class Test{
    public int number = 2;

    public Test(){

    }
}
```

Now let's try to create an instance of the class. In this example, **we can** access `number` because it is `public`.

```
public class Other{

    public static void main(String[] args){
        Test t = new Test();
        System.out.println(t.number);
    }
```

```
}
```

Section 46.3: Package Visibility

With **no modifier**, the default is package visibility. *From the Java Documentation,* "[package visibility] indicates whether classes in the same package as the class (regardless of their parentage) have access to the member." In this example from `javax.swing`,

```
package javax.swing;
public abstract class JComponent extends Container ... {
    ...
    static boolean DEBUG_GRAPHICS_LOADED;
    ...
}
```

`DebugGraphics` is in the same package, so `DEBUG_GRAPHICS_LOADED` is accessible.

```
package javax.swing;
public class DebugGraphics extends Graphics {
    ...
    static {
        JComponent.DEBUG_GRAPHICS_LOADED = true;
    }
    ...
}
```

This article gives some background on the topic.

Section 46.4: Protected Visibility

Protected visibility causes means that this member is visible to its package, along with any of its subclasses.

As an example:

```
package com.stackexchange.docs;
public class MyClass{
    protected int variable; //This is the variable that we are trying to access
    public MyClass(){
        variable = 2;
    };
}
```

Now we'll extend this class and try to access one of its **`protected`** members.

```
package some.other.pack;
import com.stackexchange.docs.MyClass;
public class SubClass extends MyClass{
    public SubClass(){
        super();
        System.out.println(super.variable);
    }
}
```

You would be also able to access a **`protected`** member without extending it if you are accessing it from the same package.

Note that this modifier only works on members of a class, not on the class itself.

Section 46.5: Summary of Class Member Access Modifiers

Access Modifier	Visibility	Inheritance
Private	Class only	Can't be inherited
No modifier / Package	In package	Available if subclass in package
Protected	In package	Available in subclass
Public	Everywhere	Available in subclass

There was once a **`private protected`** (both keywords at once) modifier that could be applied to methods or variables to make them accessible from a subclass outside the package, but make them private to the classes in that package. However, this was removed in Java 1.0's release.

Section 46.6: Interface members

```
public interface MyInterface {
    public void foo();
    int bar();

    public String TEXT = "Hello";
    int ANSWER = 42;

    public class X {
    }

    class Y {
    }
}
```

Interface members always have public visibility, even if the **`public`** keyword is omitted. So both `foo()`, `bar()`, `TEXT`, `ANSWER`, `X`, and `Y` have public visibility. However, access may still be limited by the containing interface - since `MyInterface` has public visibility, its members may be accessed from anywhere, but if `MyInterface` had had package visibility, its members would only have been accessible from within the same package.

Chapter 47: Generics

Generics are a facility of generic programming that extend Java's type system to allow a type or method to operate on objects of various types while providing compile-time type safety. In particular, the Java collections framework supports generics to specify the type of objects stored in a collection instance.

Section 47.1: Creating a Generic Class

Generics enable classes, interfaces, and methods to take other classes and interfaces as type parameters.

This example uses generic class `Param` to take a single **type parameter** `T`, delimited by angle brackets (`<>`):

```
public class Param<T> {
    private T value;

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}
```

To instantiate this class, provide a **type argument** in place of `T`. For example, `Integer`:

```
Param<Integer> integerParam = new Param<Integer>();
```

The type argument can be any reference type, including arrays and other generic types:

```
Param<String[]> stringArrayParam;
Param<int[][]> int2dArrayParam;
Param<Param<Object>> objectNestedParam;
```

In Java SE 7 and later, the type argument can be replaced with an empty set of type arguments (`<>`) called the *diamond*:

```
Version ≥ Java SE 7
Param<Integer> integerParam = new Param<>();
```

Unlike other identifiers, type parameters have no naming constraints. However their names are commonly the first letter of their purpose in upper case. (This is true even throughout the official JavaDocs.)
Examples include `T` for "type", `E` for "element" and `K`/`V` for "key"/"value".

Extending a generic class

```
public abstract class AbstractParam<T> {
    private T value;

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
```

```
    }
}
```

`AbstractParam` is an abstract class declared with a type parameter of `T`. When extending this class, that type parameter can be replaced by a type argument written inside `<>`, or the type parameter can remain unchanged. In the first and second examples below, `String` and `Integer` replace the type parameter. In the third example, the type parameter remains unchanged. The fourth example doesn't use generics at all, so it's similar to if the class had an `Object` parameter. The compiler will warn about `AbstractParam` being a raw type, but it will compile the `ObjectParam` class. The fifth example has 2 type parameters (see "multiple type parameters" below), choosing the second parameter as the type parameter passed to the superclass.

```
public class Email extends AbstractParam<String> {
    // ...
}

public class Age extends AbstractParam<Integer> {
    // ...
}

public class Height<T> extends AbstractParam<T> {
    // ...
}

public class ObjectParam extends AbstractParam {
    // ...
}

public class MultiParam<T, E> extends AbstractParam<E> {
    // ...
}
```

The following is the usage:

```
Email email = new Email();
email.setValue("test@example.com");
String retrievedEmail = email.getValue();

Age age = new Age();
age.setValue(25);
Integer retrievedAge = age.getValue();
int autounboxedAge = age.getValue();

Height<Integer> heightInInt = new Height<>();
heightInInt.setValue(125);

Height<Float> heightInFloat = new Height<>();
heightInFloat.setValue(120.3f);

MultiParam<String, Double> multiParam = new MultiParam<>();
multiParam.setValue(3.3);
```

Notice that in the `Email` class, the `T getValue()` method acts as if it had a signature of `String getValue()`, and the `void setValue(T)` method acts as if it was declared `void setValue(String)`.

It is also possible to instantiate with anonymous inner class with an empty curly braces ({}):

```
AbstractParam<Double> height = new AbstractParam<Double>(){};
height.setValue(198.6);
```

Note that using the diamond with anonymous inner classes is not allowed.

Multiple type parameters

Java provides the ability to use more than one type parameter in a generic class or interface. Multiple type parameters can be used in a class or interface by placing a **comma-separated list** of types between the angle brackets. Example:

```
public class MultiGenericParam<T, S> {
    private T firstParam;
    private S secondParam;

    public MultiGenericParam(T firstParam, S secondParam) {
        this.firstParam = firstParam;
        this.secondParam = secondParam;
    }

    public T getFirstParam() {
        return firstParam;
    }

    public void setFirstParam(T firstParam) {
        this.firstParam = firstParam;
    }

    public S getSecondParam() {
        return secondParam;
    }

    public void setSecondParam(S secondParam) {
        this.secondParam = secondParam;
    }
}
```

The usage can be done as below:

```
MultiGenericParam<String, String> aParam = new MultiGenericParam<String, String>("value1",
"value2");
MultiGenericParam<Integer, Double> dayOfWeekDegrees = new MultiGenericParam<Integer, Double>(1,
2.6);
```

Section 47.2: Deciding between `T`, `? super T`, and `? extends T`

The syntax for Java generics bounded wildcards, representing the unknown type by `?` is:

- `? extends T` represents an upper bounded wildcard. The unknown type represents a type that must be a subtype of T, or type T itself.
- `? super T` represents a lower bounded wildcard. The unknown type represents a type that must be a supertype of T, or type T itself.

As a rule of thumb, you should use

- `? extends T` if you only need "read" access ("input")
- `? super T` if you need "write" access ("output")

- `T` if you need both ("modify")

Using **`extends`** or **`super`** is usually *better* because it makes your code more flexible (as in: allowing the use of subtypes and supertypes), as you will see below.

```
class Shoe {}
class IPhone {}
interface Fruit {}
class Apple implements Fruit {}
class Banana implements Fruit {}
class GrannySmith extends Apple {}

   public class FruitHelper {

        public void eatAll(Collection<? extends Fruit> fruits) {}

        public void addApple(Collection<? super Apple> apples) {}
}
```

The compiler will now be able to detect certain bad usage:

```
public class GenericsTest {
     public static void main(String[] args){
  FruitHelper fruitHelper = new FruitHelper() ;
    List<Fruit> fruits = new ArrayList<Fruit>();
    fruits.add(new Apple()); // Allowed, as Apple is a Fruit
    fruits.add(new Banana()); // Allowed, as Banana is a Fruit
    fruitHelper.addApple(fruits); // Allowed, as "Fruit super Apple"
    fruitHelper.eatAll(fruits); // Allowed

    Collection<Banana> bananas = new ArrayList<>();
    bananas.add(new Banana()); // Allowed
    //fruitHelper.addApple(bananas); // Compile error: may only contain Bananas!
    fruitHelper.eatAll(bananas); // Allowed, as all Bananas are Fruits

    Collection<Apple> apples = new ArrayList<>();
    fruitHelper.addApple(apples); // Allowed
    apples.add(new GrannySmith()); // Allowed, as this is an Apple
    fruitHelper.eatAll(apples); // Allowed, as all Apples are Fruits.

    Collection<GrannySmith> grannySmithApples = new ArrayList<>();
    fruitHelper.addApple(grannySmithApples); //Compile error: Not allowed.
                                  // GrannySmith is not a supertype of Apple
    apples.add(new GrannySmith()); //Still allowed, GrannySmith is an Apple
    fruitHelper.eatAll(grannySmithApples);//Still allowed, GrannySmith is a Fruit

    Collection<Object> objects = new ArrayList<>();
    fruitHelper.addApple(objects); // Allowed, as Object super Apple
    objects.add(new Shoe()); // Not a fruit
    objects.add(new IPhone()); // Not a fruit
    //fruitHelper.eatAll(objects); // Compile error: may contain a Shoe, too!
}
```

Choosing the right `T`, `? super T` or `? extends T` is *necessary* to allow the use with subtypes. The compiler can then ensure type safety; you should not need to cast (which is not type safe, and may cause programming errors) if you use them properly.

If it is not easy to understand, please remember **PECS** rule:

> **P**roducer uses "**E**xtends" and **C**onsumer uses "**S**uper".

(Producer has only write access, and Consumer has only read access)

Section 47.3: The Diamond

Version ≥ Java SE 7

Java 7 introduced the *Diamond*1 to remove some boiler-plate around generic class instantiation. With Java 7+ you can write:

```
List<String> list = new LinkedList<>();
```

Where you had to write in previous versions, this:

```
List<String> list = new LinkedList<String>();
```

One limitation is for Anonymous Classes, where you still must provide the type parameter in the instantiation:

```
// This will compile:

Comparator<String> caseInsensitiveComparator = new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        return s1.compareToIgnoreCase(s2);
    }
};

// But this will not:

Comparator<String> caseInsensitiveComparator = new Comparator<>() {
    @Override
    public int compare(String s1, String s2) {
        return s1.compareToIgnoreCase(s2);
    }
};
```

Version > Java SE 8

Although using the diamond with Anonymous Inner Classes is not supported in Java 7 and 8, **it will be included as a new feature in Java 9**.

Footnote:

1 - Some people call the `<>` usage the "diamond *operator*". This is incorrect. The diamond does not behave as an operator, and is not described or listed anywhere in the JLS or the (official) Java Tutorials as an operator. Indeed, `<>` is not even a distinct Java token. Rather it is a `<` token followed by a `>` token, and it is legal (though bad style) to have whitespace or comments between the two. The JLS and the Tutorials consistently refer to `<>` as "the diamond", and that is therefore the correct term for it.

Section 47.4: Declaring a Generic Method

Methods can also have generic type parameters.

```
public class Example {
```

```
    // The type parameter T is scoped to the method
    // and is independent of type parameters of other methods.
    public <T> List<T> makeList(T t1, T t2) {
        List<T> result = new ArrayList<T>();
        result.add(t1);
        result.add(t2);
        return result;
    }

    public void usage() {
        List<String> listString = makeList("Jeff", "Atwood");
        List<Integer> listInteger = makeList(1, 2);
    }
}
```

Notice that we don't have to pass an actual type argument to a generic method. The compiler infers the type argument for us, based on the target type (e.g. the variable we assign the result to), or on the types of the actual arguments. It will generally infer the most specific type argument that will make the call type-correct.

Sometimes, albeit rarely, it can be necessary to override this type inference with explicit type arguments:

```
void usage() {
    consumeObjects(this.<Object>makeList("Jeff", "Atwood").stream());
}

void consumeObjects(Stream<Object> stream) { ... }
```

It's necessary in this example because the compiler can't "look ahead" to see that `Object` is desired for `T` after calling `stream()` and it would otherwise infer `String` based on the `makeList` arguments. Note that the Java language doesn't support omitting the class or object on which the method is called (**`this`** in the above example) when type arguments are explicitly provided.

Section 47.5: Requiring multiple upper bounds (″extends A & B″)

You can require a generic type to extend multiple upper bounds.

Example: we want to sort a list of numbers but `Number` doesn't implement `Comparable`.

```
public <T extends Number & Comparable<T>> void sortNumbers( List<T> n ) {
  Collections.sort( n );
}
```

In this example `T` must extend `Number` *and* implement `Comparable<T>` which should fit all "normal" built-in number implementations like `Integer` or `BigDecimal` but doesn't fit the more exotic ones like `Striped64`.

Since multiple inheritance is not allowed, you can use at most one class as a bound and it must be the first listed. For example, **`<T`** `extends Comparable`**`<T>`** `& Number>` is not allowed because Comparable is an interface, and not a class.

Section 47.6: Obtain class that satisfies generic parameter at runtime

Many unbound generic parameters, like those used in a static method, cannot be recovered at runtime (see *Other Threads* on *Erasure*). However there is a common strategy employed for accessing the type satisfying a generic parameter on a class at runtime. This allows for generic code that depends on access to type *without* having to

thread type information through every call.

Background

Generic parameterization on a class can be inspected by creating an anonymous inner class. This class will capture the type information. In general this mechanism is referred to as **super type tokens**, which are detailed in Neal Gafter's blog post.

Implementations

Three common implementations in Java are:

- Guava's TypeToken
- Spring's ParameterizedTypeReference
- Jackson's TypeReference

Example usage

```
public class DataService<MODEL_TYPE> {
     private final DataDao dataDao = new DataDao();
     private final Class<MODEL_TYPE> type = (Class<MODEL_TYPE>) new TypeToken<MODEL_TYPE>
                                                                (getClass()){}.getRawType();
     public List<MODEL_TYPE> getAll() {
         return dataDao.getAllOfType(type);
    }
}

// the subclass definitively binds the parameterization to User
// for all instances of this class, so that information can be
// recovered at runtime
public class UserService extends DataService<User> {}

public class Main {
    public static void main(String[] args) {
          UserService service = new UserService();
          List<User> users = service.getAll();
    }
}
```

Section 47.7: Benefits of Generic class and interface

Code that uses generics has many benefits over non-generic code. Below are the main benefits

Stronger type checks at compile time

A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.

Elimination of casts

The following code snippet without generics requires casting:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

When re-written to *use generics*, the code does not require casting:

```
List<String> list = new ArrayList<>();
list.add("hello");
String s = list.get(0);   // no cast
```

Enabling programmers to implement generic algorithms

By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

Section 47.8: Instantiating a generic type

Due to type erasure the following will not work:

```
public <T> void genericMethod() {
    T t = new T(); // Can not instantiate the type T.
}
```

The type `T` is erased. Since, at runtime, the JVM does not know what `T` originally was, it does not know which constructor to call.

Workarounds

1. Passing `T`'s class when calling `genericMethod`:

   ```
   public <T> void genericMethod(Class<T> cls) {
       try {
           T t = cls.newInstance();
       } catch (InstantiationException | IllegalAccessException e) {
            System.err.println("Could not instantiate: " + cls.getName());
       }
   }
   ```

   ```
   genericMethod(String.class);
   ```

   Which throws exceptions, since there is no way to know if the passed class has an accessible default constructor.

`Version ≥ Java SE 8`

2. Passing a reference to `T`'s constructor:

   ```
   public <T> void genericMethod(Supplier<T> cons) {
       T t = cons.get();
   }
   ```

   ```
   genericMethod(String::new);
   ```

Section 47.9: Creating a Bounded Generic Class

You can restrict the valid types used in a **generic class** by bounding that type in the class definition. Given the following simple type hierarchy:

```
public abstract class Animal {
    public abstract String getSound();
}

public class Cat extends Animal {
    public String getSound() {
        return "Meow";
    }
}

public class Dog extends Animal {
    public String getSound() {
        return "Woof";
    }
}
```

Without **bounded generics**, we cannot make a container class that is both generic and knows that each element is an animal:

```
public class AnimalContainer<T> {

    private Collection<T> col;

    public AnimalContainer() {
        col = new ArrayList<T>();
    }

    public void add(T t) {
        col.add(t);
    }

    public void printAllSounds() {
        for (T t : col) {
            // Illegal, type T doesn't have makeSound()
            // it is used as an java.lang.Object here
            System.out.println(t.makeSound());
        }
    }
}
```

With generic bound in class definition, this is now possible.

```
public class BoundedAnimalContainer<T extends Animal> { // Note bound here.

    private Collection<T> col;

    public BoundedAnimalContainer() {
        col = new ArrayList<T>();
    }

    public void add(T t) {
        col.add(t);
    }

    public void printAllSounds() {
        for (T t : col) {
            // Now works because T is extending Animal
            System.out.println(t.makeSound());
        }
    }
```

```
}
```

This also restricts the valid instantiations of the generic type:

```
// Legal
AnimalContainer<Cat> a = new AnimalContainer<Cat>();

// Legal
AnimalContainer<String> a = new AnimalContainer<String>();
```

```
// Legal because Cat extends Animal
BoundedAnimalContainer<Cat> b = new BoundedAnimalContainer<Cat>();

// Illegal because String doesn't extends Animal
BoundedAnimalContainer<String> b = new BoundedAnimalContainer<String>();
```

Section 47.10: Referring to the declared generic type within its own declaration

How do you go about using an instance of a (possibly further) inherited generic type within a method declaration in the generic type itself being declared? This is one of the problems you will face when you dig a bit deeper into generics, but still a fairly common one.

Assume we have a `DataSeries<T>` type (interface here), which defines a generic data series containing values of type `T`. It is cumbersome to work with this type directly when we want to perform a lot of operations with e.g. double values, so we define `DoubleSeries extends DataSeries<Double>`. Now assume, the original `DataSeries<T>` type has a method `add(values)` which adds another series of the same length and returns a new one. How do we enforce the type of `values` and the type of the return to be `DoubleSeries` rather than `DataSeries<Double>` in our derived class?

The problem can be solved by adding a generic type parameter referring back to and extending the type being declared (applied to an interface here, but the same stands for classes):

```
public interface DataSeries<T, DS extends DataSeries<T, DS>> {
    DS add(DS values);
    List<T> data();
}
```

Here `T` represents the data type the series holds, e.g. `Double` and `DS` the series itself. An inherited type (or types) can now be easily implemented by substituting the above mentioned parameter by a corresponding derived type, thus, yielding a concrete `Double`-based definition of the form:

```
public interface DoubleSeries extends DataSeries<Double, DoubleSeries> {
    static DoubleSeries instance(Collection<Double> data) {
        return new DoubleSeriesImpl(data);
    }
}
```

At this moment even an IDE will implement the above interface with correct types in place, which, after a bit of content filling may look like this:

```
class DoubleSeriesImpl implements DoubleSeries {
    private final List<Double> data;

    DoubleSeriesImpl(Collection<Double> data) {
        this.data = new ArrayList<>(data);
```

```
    }

    @Override
    public DoubleSeries add(DoubleSeries values) {
        List<Double> incoming = values != null ? values.data() : null;
        if (incoming == null || incoming.size() != data.size()) {
            throw new IllegalArgumentException("bad series");
        }
        List<Double> newdata = new ArrayList<>(data.size());
        for (int i = 0; i < data.size(); i++) {
            newdata.add(this.data.get(i) + incoming.get(i)); // beware autoboxing
        }
        return DoubleSeries.instance(newdata);
    }

    @Override
    public List<Double> data() {
        return Collections.unmodifiableList(data);
    }
}
```

As you can see the `add` method is declared as `DoubleSeries add(DoubleSeries values)` and the compiler is happy.

The pattern can be further nested if required.

Section 47.11: Binding generic parameter to more than 1 type

Generic parameters can also be bound to more than one type using the `T extends Type1 & Type2 & ...` syntax.

Let's say you want to create a class whose Generic type should implement both `Flushable` and `Closeable`, you can write

```
class ExampleClass<T extends Flushable & Closeable> {
}
```

Now, the `ExampleClass` only accepts as generic parameters, types which implement both `Flushable` **and** `Closeable`.

```
ExampleClass<BufferedWriter> arg1; // Works because BufferedWriter implements both Flushable and
Closeable

ExampleClass<Console> arg4; // Does NOT work because Console only implements Flushable
ExampleClass<ZipFile> arg5; // Does NOT work because ZipFile only implements Closeable

ExampleClass<Flushable> arg2; // Does NOT work because Closeable bound is not satisfied.
ExampleClass<Closeable> arg3; // Does NOT work because Flushable bound is not satisfied.
```

The class methods can choose to infer generic type arguments as either `Closeable` or `Flushable`.

```
class ExampleClass<T extends Flushable & Closeable> {
    /* Assign it to a valid type as you want. */
    public void test (T param) {
        Flushable arg1 = param; // Works
        Closeable arg2 = param; // Works too.
    }

    /* You can even invoke the methods of any valid type directly. */
```

```
    public void test2 (T param) {
        param.flush(); // Method of Flushable called on T and works fine.
        param.close(); // Method of Closeable called on T and works fine too.
    }
}
```

Note:

You cannot bind the generic parameter to either of the type using *OR* (`|`) clause. Only the *AND* (`&`) clause is supported. Generic type can extends only one class and many interfaces. Class must be placed at the beginning of the list.

Section 47.12: Using Generics to auto-cast

With generics, it's possible to return whatever the caller expects:

```
private Map<String, Object> data;
public <T> T get(String key) {
    return (T) data.get(key);
}
```

The method will compile with a warning. The code is actually more safe than it looks because the Java runtime will do a cast when you use it:

```
Bar bar = foo.get("bar");
```

It's less safe when you use generic types:

```
List<Bar> bars = foo.get("bars");
```

Here, the cast will work when the returned type is any kind of `List` (i.e. returning `List<String>` would not trigger a `ClassCastException`; you'd eventually get it when taking elements out of the list).

To work around this problem, you can create an API which uses typed keys:

```
public final static Key<List<Bar>> BARS = new Key<>("BARS");
```

along with this `put()` method:

```
public <T> T put(Key<T> key, T value);
```

With this approach, you can't put the wrong type into the map, so the result will always be correct (unless you accidentally create two keys with the same name but different types).

Related:

- Type-safe Map

Section 47.13: Use of instanceof with Generics

Using generics to define the type in instanceof

Consider the following generic class `Example` declared with the formal parameter `<T>`:

```
class Example<T> {
```

```
    public boolean isTypeAString(String s) {
        return s instanceof T; // Compilation error, cannot use T as class type here
    }
}
```

This will always give a Compilation error because as soon as the compiler compiles the *Java source* into *Java bytecode* it applies a process known as *type erasure*, which converts all generic code into non-generic code, making impossible to distinguish among T types at runtime. The type used with **`instanceof`** has to be *reifiable*, which means that all information about the type has to be available at runtime, and this is usually not the case for generic types.

The following class represents what two different classes of `Example`, `Example<String>` and `Example<Number>`, look like after generics has stripped off by *type erasure*:

```
class Example { // formal parameter is gone
    public boolean isTypeAString(String s) {
        return s instanceof Object; // Both <String> and <Number> are now Object
    }
}
```

Since types are gone, it's not possible for the JVM to know which type is `T`.

Exception to the previous rule

You can always use *unbounded wildcard* (?) for specifying a type in the **`instanceof`** as follows:

```
    public boolean isAList(Object obj) {
        return obj instanceof List<?>;
    }
```

This can be useful to evaluate whether an instance `obj` is a `List` or not:

```
System.out.println(isAList("foo")); // prints false
System.out.println(isAList(new ArrayList<String>()); // prints true
System.out.println(isAList(new ArrayList<Float>()); // prints true
```

In fact, unbounded wildcard is considered a reifiable type.

Using a generic instance with instanceof

The other side of the coin is that using an instance `t` of `T` with **`instanceof`** is legal, as shown in the following example:

```
class Example<T> {
    public boolean isTypeAString(T t) {
        return t instanceof String; // No compilation error this time
    }
}
```

because after the type erasure the class will look like the following:

```
class Example { // formal parameter is gone
    public boolean isTypeAString(Object t) {
        return t instanceof String; // No compilation error this time
    }
}
```

Since, even if the type erasure happen anyway, now the JVM can distinguish among different types in memory, even if they use the same reference type (`Object`), as the following snippet shows:

```
Object obj1 = new String("foo"); // reference type Object, object type String
Object obj2 = new Integer(11); // reference type Object, object type Integer
System.out.println(obj1 instanceof String); // true
System.out.println(obj2 instanceof String); // false, it's an Integer, not a String
```

Section 47.14: Different ways for implementing a Generic Interface (or extending a Generic Class)

Suppose the following generic interface has been declared:

```
public interface MyGenericInterface<T> {
    public void foo(T t);
}
```

Below are listed the possible ways to implement it.

Non-generic class implementation with a specific type

Choose a specific type to replace the formal type parameter **<T>** of `MyGenericClass` and implement it, as the following example does:

```
public class NonGenericClass implements MyGenericInterface<String> {
    public void foo(String t) { } // type T has been replaced by String
}
```

This class only deals with `String`, and this means that using `MyGenericInterface` with different parameters (e.g. `Integer`, `Object` etc.) won't compile, as the following snippet shows:

```
NonGenericClass myClass = new NonGenericClass();
myClass.foo("foo_string"); // OK, legal
myClass.foo(11); // NOT OK, does not compile
myClass.foo(new Object()); // NOT OK, does not compile
```

Generic class implementation

Declare another generic interface with the formal type parameter **<T>** which implements `MyGenericInterface`, as follows:

```
public class MyGenericSubclass<T> implements MyGenericInterface<T> {
    public void foo(T t) { } // type T is still the same
    // other methods...
}
```

Note that a different formal type parameter may have been used, as follows:

```
public class MyGenericSubclass<U> implements MyGenericInterface<U> { // equivalent to the previous
declaration
    public void foo(U t) { }
    // other methods...
}
```

Raw type class implementation

Declare a non-generic class which implements `MyGenericInteface` as a *raw type* (not using generic at all), as follows:

```
public class MyGenericSubclass implements MyGenericInterface {
    public void foo(Object t) { } // type T has been replaced by Object
    // other possible methods
}
```

This way is **not** recommended, since it is not 100% safe at runtime because it mixes up *raw type* (of the subclass) with *generics* (of the interface) and it is also confusing. Modern Java compilers will raise a warning with this kind of implementation, nevertheless the code - for compatibility reasons with older JVM (1.4 or earlier) - will compile.

All the ways listed above are also allowed when using a generic class as a supertype instead of a generic interface.

Chapter 48: Classes and Objects

Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating. An object is an instance of a class.

Class − A class can be defined as a template/blueprint that describes the behavior/state that the object of its type support.

Section 48.1: Overloading Methods

Sometimes the same functionality has to be written for different kinds of inputs. At that time, one can use the same method name with a different set of parameters. Each different set of parameters is known as a method signature. As seen per the example, a single method can have multiple signatures.

```
public class Displayer {

    public void displayName(String firstName) {
        System.out.println("Name is: " + firstName);
    }

    public void displayName(String firstName, String lastName) {
        System.out.println("Name is: " + firstName + " " + lastName);
    }

    public static void main(String[] args) {
        Displayer displayer = new Displayer();
        displayer.displayName("Ram");          //prints "Name is: Ram"
        displayer.displayName("Jon", "Skeet"); //prints "Name is: Jon Skeet"
    }
}
```

The advantage is that the same functionality is called with two different numbers of inputs. While invoking the method according to the input we are passing, (In this case either one string value or two string values) the corresponding method is executed.

Methods can be overloaded:

1. Based on the **number of parameters** passed.

   Example: `method(String s)` and `method(String s1, String s2)`.

2. Based on the **order of parameters**.

   Example: `method(int i, float f)` and `method(float f, int i))`.

Note: *Methods cannot be overloaded by changing just the return type (`int method()` is considered the same as `String method()` and will throw a `RuntimeException` if attempted). If you change the return type you must also change the parameters in order to overload.*

Section 48.2: Explaining what is method overloading and overriding

Method Overriding and Overloading are two forms of polymorphism supported by Java.

Method Overloading

Method overloading (also known as static Polymorphism) is a way you can have two (or more) methods (functions) with same name in a single class. Yes its as simple as that.

```
public class Shape{
    //It could be a circle or rectangle or square
    private String type;

    //To calculate area of rectangle
    public Double area(Long length, Long breadth){
        return (Double) length * breadth;
    }

     //To calculate area of a circle
     public Double area(Long radius){
        return (Double) 3.14 * r * r;
    }
}
```

This way user can call the same method for area depending on the type of shape it has.

But the real question now is, how will java compiler will distinguish which method body is to be executed?

Well Java have made it clear that even though the **method names** (`area()` in our case) **can be same but the arguments method is taking should be different.**

> Overloaded methods must have different arguments list (quantity and types).

That being said we cannot add another method to calculate area of a square like this : `public Double area(Long side)` because in this case, it will conflict with area method of circle and will cause **ambiguity** for java compiler.

Thank god, there are some relaxations while writing overloaded methods like

> May have different return types.

> May have different access modifiers.

> May throw different exceptions.

Why is this called static polymorphism?

Well that's because which overloaded methods is to be invoked is decided at compile time, based on the actual number of arguments and the compile-time types of the arguments.

> One of common reasons of using method overloading is the simplicity of code it provides. For example remember `String.valueOf()` which takes almost any type of argument? What is written behind the scene is probably something like this:

```
static String valueOf(boolean b)
static String valueOf(char c)
static String valueOf(char[] data)
static String valueOf(char[] data, int offset, int count)
static String valueOf(double d)
static String valueOf(float f)
static String valueOf(int i)
static String valueOf(long l)
static String valueOf(Object obj)
```

Method Overriding

Well, method overriding (yes you guess it right, it is also known as dynamic polymorphism) is somewhat more interesting and complex topic.

In method overriding we overwrite the method body provided by the parent class. Got it? No? Let's go through an example.

```
public abstract class Shape{

    public abstract Double area(){
        return 0.0;
    }
}
```

So we have a class called Shape and it has method called area which will probably return the area of the shape.

Let's say now we have two classes called Circle and Rectangle.

```
public class Circle extends Shape {
    private Double radius = 5.0;

    // See this annotation @Override, it is telling that this method is from parent
    // class Shape and is overridden here
    @Override
    public Double area(){
        return 3.14 * radius * radius;
    }
}
```

Similarly rectangle class:

```
public class Rectangle extends Shape {
    private Double length = 5.0;
    private Double breadth= 10.0;


    // See this annotation @Override, it is telling that this method is from parent
    // class Shape and is overridden here
    @Override
    public Double area(){
        return length * breadth;
    }
```

```
}
```

So, now both of your children classes have updated method body provided by the parent (`Shape`) class. Now question is how to see the result? Well lets do it the old `psvm` way.

```
public class AreaFinder{

    public static void main(String[] args){

        //This will create an object of circle class
        Shape circle = new Circle();
        //This will create an object of Rectangle class
        Shape rectangle = new Rectangle();

        // Drumbeats ......
        //This should print 78.5
        System.out.println("Shape of circle : "+circle.area());

        //This should print 50.0
        System.out.println("Shape of rectangle: "+rectangle.area());

    }
}
```

Wow! isn't it great? Two objects of same type calling same methods and returning different values. My friend, that's the power of dynamic polymorphism.

Here's a chart to better compare the differences between these two:

Method Overloading	Method Overriding
Method overloading is used to increase the readability of the program.	Method overriding is used to provide the specific implementation of the method that is already provided by its super class.
Method overloading is performed within class.	Method overriding occurs in two classes that have IS-A (inheritance) relationship.
In case of method overloading, parameter must be different.	In case of method overriding, parameter must be same.
Method overloading is the example of compile time polymorphism.	Method overriding is the example of run time polymorphism.
In java, method overloading can't be performed by changing return type of the method only. Return type can be same or different in method overloading. But you must have to change the parameter.	Return type must be same or covariant in method overriding.

Section 48.3: Constructors

Constructors are special methods named after the class and without a return type, and are used to construct objects. Constructors, like methods, can take input parameters. Constructors are used to initialize objects. Abstract classes can have constructors also.

```
public class Hello{
    // constructor
    public Hello(String wordToPrint){
        printHello(wordToPrint);
    }
    public void printHello(String word){
        System.out.println(word);
```

```
    }
}
// instantiates the object during creating and prints out the content
// of wordToPrint
```

It is important to understand that constructors are different from methods in several ways:

1. Constructors can only take the modifiers **`public`**, **`private`**, and **`protected`**, and cannot be declared **`abstract`**, **`final`**, **`static`**, or **`synchronized`**.
2. Constructors do not have a return type.
3. Constructors MUST be named the same as the class name. In the `Hello` example, the `Hello` object's constructor name is the same as the class name.
4. The **`this`** keyword has an additional usage inside constructors. **`this`**`.method(...)` calls a method on the current instance, while **`this`**`(...)` refers to another constructor in the current class with different signatures.

Constructors also can be called through inheritance using the keyword **`super`**.

```
public class SuperManClass{

    public SuperManClass(){
        // some implementation
    }

    // ... methods
}


public class BatmanClass extends SupermanClass{
    public BatmanClass(){
        super();
    }
    //... methods...
}
```

See Java Language Specification #8.8 and #15.9

Section 48.4: Initializing static final fields using a static initializer

To initialize a **`static final`** fields that require using more than a single expression, a **`static`** initializer can be used to assign the value. The following example initializes a unmodifiable set of `String`s:

```
public class MyClass {

    public static final Set<String> WORDS;

    static {
        Set<String> set = new HashSet<>();
        set.add("Hello");
        set.add("World");
        set.add("foo");
        set.add("bar");
        set.add("42");
        WORDS = Collections.unmodifiableSet(set);
    }
```

```
}
```

Section 48.5: Basic Object Construction and Use

Objects come in their own class, so a simple example would be a car (detailed explanations below):

```
public class Car {

    //Variables describing the characteristics of an individual car, varies per  object
   private int milesPerGallon;
   private String name;
   private String color;
   public int numGallonsInTank;

    public Car(){
        milesPerGallon = 0;
        name = "";
        color = "";
        numGallonsInTank = 0;
    }

    //this is where an individual object is created
    public Car(int mpg, int, gallonsInTank, String carName, String carColor){
        milesPerGallon = mpg;
        name = carName;
        color = carColor;
        numGallonsInTank = gallonsInTank;
    }

    //methods to make the object more usable

    //Cars need to drive
    public void drive(int distanceInMiles){
        //get miles left in car
        int miles = numGallonsInTank * milesPerGallon;

        //check that car has enough gas to drive distanceInMiles
        if (miles <= distanceInMiles){
            numGallonsInTank = numGallonsInTank - (distanceInMiles / milesPerGallon)
            System.out.println("Drove " + numGallonsInTank + " miles!");
        } else {
            System.out.println("Could not drive!");
        }
    }

    public void paintCar(String newColor){
        color = newColor;
    }
        //set new Miles Per Gallon
    public void setMPG(int newMPG){
        milesPerGallon = newMPG;
    }

       //set new number of Gallon In Tank
    public void setGallonsInTank(int numGallons){
        numGallonsInTank = numGallons;
    }

    public void nameCar(String newName){
        name = newName;
    }
```

```
    //Get the Car color
    public String getColor(){
        return color;
    }

    //Get the Car name
    public String getName(){
        return name;
    }

    //Get the number of Gallons
    public String getGallons(){
        return numGallonsInTank;
    }

}
```

Objects are **instances of** their class. So, the way you would **create an object** would be by calling the Car class in **one of two ways** in your main class (main method in Java or onCreate in Android).

Option 1

```
`Car newCar = new Car(30, 10, "Ferrari", "Red");
```

Option 1 is where you essentially tell the program everything about the Car upon creation of the object. Changing any property of the car would require calling one of the methods such as the `repaintCar` method. Example:

```
newCar.repaintCar("Blue");
```

Note: Make sure you pass the correct data type to the method. In the example above, you may also pass a variable to the `repaintCar` method **as long as the data type is correct`**.

That was an example of changing properties of an object, receiving properties of an object would require using a method from the Car class that has a return value (meaning a method that is not **`void`**). Example:

```
String myCarName = newCar.getName();  //returns string "Ferrari"
```

Option 1 is the **best** option when you have **all the object's data** at the time of creation.

Option 2

```
`Car newCar = new Car();
```

Option 2 gets the same effect but required more work to create an object correctly. I want to recall this Constructor in the Car class:

```
public void Car(){
        milesPerGallon = 0;
        name = "";
        color = "";
        numGallonsInTank = 0;
    }
```

Notice that you do not have to actually pass any parameters into the object to create it. This is very useful for when you do not have all the aspects of the object but you need to use the parts that you do have. This sets generic data into each of the instance variables of the object so that, if you call for a piece of data that does not exist, no errors

are thrown.

Note: Do not forget that you have to set the parts of the object later that you did not initialize it with. For example,

```
Car myCar = new Car();
String color = Car.getColor(); //returns empty string
```

This is a common mistake amongst objects that are not initialized with all their data. Errors were avoided because there is a Constructor that allows an empty Car object to be created with **stand-in variables** (`public Car(){}`), but no part of the myCar was actually customized. **Correct example of creating Car Object:**

```
Car myCar = new Car();
myCar.nameCar("Ferrari");
myCar.paintCar("Purple");
myCar.setGallonsInTank(10);
myCar.setMPG(30);
```

And, as a reminder, get an object's properties by calling a method in your main class. Example:

```
String myCarName = myCar.getName(); //returns string "Ferrari"
```

Section 48.6: Simplest Possible Class

```
class TrivialClass {}
```

A class consists at a minimum of the `class` keyword, a name, and a body, which might be empty.

You instantiate a class with the `new` operator.

```
TrivialClass tc = new TrivialClass();
```

Section 48.7: Object Member vs Static Member

With this class:

```
class ObjectMemberVsStaticMember {

    static int staticCounter = 0;
    int memberCounter = 0;

    void increment() {
        staticCounter ++;
        memberCounter++;
    }
}
```

the following code snippet:

```
final ObjectMemberVsStaticMember o1 = new ObjectMemberVsStaticMember();
final ObjectMemberVsStaticMember o2 = new ObjectMemberVsStaticMember();

o1.increment();

o2.increment();
o2.increment();
```

```
System.out.println("o1 static counter " + o1.staticCounter);
System.out.println("o1 member counter " + o1.memberCounter);
System.out.println();

System.out.println("o2 static counter " + o2.staticCounter);
System.out.println("o2 member counter " + o2.memberCounter);
System.out.println();

System.out.println("ObjectMemberVsStaticMember.staticCounter = " +
ObjectMemberVsStaticMember.staticCounter);

// the following line does not compile. You need an object
// to access its members
//System.out.println("ObjectMemberVsStaticMember.staticCounter = " +
ObjectMemberVsStaticMember.memberCounter);
```

produces this output:

```
o1 static counter 3
o1 member counter 1

o2 static counter 3
o2 member counter 2

ObjectMemberVsStaticMember.staticCounter = 3
```

Note: You should not call `static` members on objects, but on classes. While it does not make a difference for the JVM, human readers will appreciate it.

`static` members are part of the class and exists only once per class. Non-`static` members exist on instances, there is an independent copy for each instance. This also means that you need access to an object of that class to access its members.

Chapter 49: Local Inner Class

A class i.e. created inside a method is called local inner class in java. If you want to invoke the methods of local inner class, you must instantiate this class inside the method.

Section 49.1: Local Inner Class

```
public class localInner1{
 private int data=30;//instance variable
 void display(){
  class Local{
   void msg(){System.out.println(data);}
  }
  Local l=new Local();
  l.msg();
 }
 public static void main(String args[]){
  localInner1 obj=new localInner1();
  obj.display();
 }
}
```

Chapter 50: Nested and Inner Classes

Using Java, developers have the ability to define a class within another class. Such a class is called a Nested Class. Nested Classes are called Inner Classes if they were declared as non-static, if not, they are simply called Static Nested Classes. This page is to document and provide details with examples on how to use Java Nested and Inner Classes.

Section 50.1: A Simple Stack Using a Nested Class

```
public class IntStack {

    private IntStackNode head;

    // IntStackNode is the inner class of the class IntStack
    // Each instance of this inner class functions as one link in the
    // Overall stack that it helps to represent
    private static class IntStackNode {

        private int val;
        private IntStackNode next;

        private IntStackNode(int v, IntStackNode n) {
            val = v;
            next = n;
        }
    }

    public IntStack push(int v) {
        head = new IntStackNode(v, head);
        return this;
    }

    public int pop() {
        int x = head.val;
        head = head.next;
        return x;
    }
}
```

And the use thereof, which (notably) does not at all acknowledge the existence of the nested class.

```
public class Main {
    public static void main(String[] args) {

        IntStack s = new IntStack();
        s.push(4).push(3).push(2).push(1).push(0);

        //prints: 0, 1, 2, 3, 4,
        for(int i = 0; i < 5; i++) {
            System.out.print(s.pop() + ", ");
        }
    }
}
```

Section 50.2: Static vs Non Static Nested Classes

When creating a nested class, you face a choice of having that nested class static:

```
public class OuterClass1 {

    private static class StaticNestedClass {

    }

}
```

Or non-static:

```
public class OuterClass2 {

    private class NestedClass {

    }

}
```

At its core, static nested classes *do not have a surrounding **instance*** of the outer class, whereas non-static nested classes do. This affects both where/when one is allowed to instantiate a nested class, and what instances of those nested classes are allowed to access. Adding to the above example:

```
public class OuterClass1 {

    private int aField;
    public void aMethod(){}

    private static class StaticNestedClass {
        private int innerField;

        private StaticNestedClass() {
             innerField = aField; //Illegal, can't access aField from static context
             aMethod();           //Illegal, can't call aMethod from static context
        }

        private StaticNestedClass(OuterClass1 instance) {
             innerField = instance.aField; //Legal
        }

    }

    public static void aStaticMethod() {
        StaticNestedClass s = new StaticNestedClass(); //Legal, able to construct in static context
        //Do stuff involving s...
    }

}

public class OuterClass2 {

    private int aField;

    public void aMethod() {}

    private class NestedClass {
        private int innerField;

        private NestedClass() {
             innerField = aField; //Legal
             aMethod(); //Legal
```

```
        }
    }

    public void aNonStaticMethod() {
        NestedClass s = new NestedClass(); //Legal
    }

    public static void aStaticMethod() {
        NestedClass s = new NestedClass(); //Illegal. Can't construct without surrounding
OuterClass2 instance.
                                           //As this is a static context, there is no surrounding
OuterClass2 instance
    }
}
```

Thus, your decision of static vs non-static mainly depends on whether or not you need to be able to directly access fields and methods of the outer class, though it also has consequences for when and where you can construct the nested class.

As a rule of thumb, make your nested classes static unless you need to access fields and methods of the outer class. Similar to making your fields private unless you need them public, this decreases the visibility available to the nested class (by not allowing access to an outer instance), reducing the likelihood of error.

Section 50.3: Access Modifiers for Inner Classes

A full explanation of Access Modifiers in Java can be found here. But how do they interact with Inner classes?

`public`, as usual, gives unrestricted access to any scope able to access the type.

```
public class OuterClass {

    public class InnerClass {

        public int x = 5;

    }

    public InnerClass createInner() {
        return new InnerClass();
    }
}

public class SomeOtherClass {

    public static void main(String[] args) {
        int x = new OuterClass().createInner().x; //Direct field access is legal
    }
}
```

both **`protected`** and the default modifier (of nothing) behave as expected as well, the same as they do for non-nested classes.

`private`, interestingly enough, does not restrict to the class it belongs to. Rather, it restricts to the compilation unit - the .java file. This means that Outer classes have full access to Inner class fields and methods, even if they are marked **`private`**.

```
public class OuterClass {
```

```
    public class InnerClass {

        private int x;
        private void anInnerMethod() {}
    }

    public InnerClass aMethod() {
        InnerClass a = new InnerClass();
        a.x = 5; //Legal
        a.anInnerMethod(); //Legal
        return a;
    }
}
```

The Inner Class itself can have a visibility other than `public`. By marking it `private` or another restricted access modifier, other (external) classes will not be allowed to import and assign the type. They can still get references to objects of that type, however.

```
public class OuterClass {

    private class InnerClass{}

    public InnerClass makeInnerClass() {
        return new InnerClass();
    }
}

public class AnotherClass {

    public static void main(String[] args) {
        OuterClass o = new OuterClass();

        InnerClass x = o.makeInnerClass(); //Illegal, can't find type
        OuterClass.InnerClass x = o.makeInnerClass(); //Illegal, InnerClass has visibility private
        Object x = o.makeInnerClass(); //Legal
    }
}
```

Section 50.4: Anonymous Inner Classes

An anonymous inner class is a form of inner class that is declared and instantiated with a single statement. As a consequence, there is no name for the class that can be used elsewhere in the program; i.e. it is anonymous.

Anonymous classes are typically used in situations where you need to be able to create a light-weight class to be passed as a parameter. This is typically done with an interface. For example:

```
public static Comparator<String> CASE_INSENSITIVE =
        new Comparator<String>() {
            @Override
            public int compare(String string1, String string2) {
                return string1.toUpperCase().compareTo(string2.toUpperCase());
            }
        };
```

This anonymous class defines a `Comparator<String>` object (`CASE_INSENSITIVE`) that compares two strings ignoring differences in case.

Other interfaces that are frequently implemented and instantiated using anonymous classes are `Runnable` and

`Callable`. For example:

```
// An anonymous Runnable class is used to provide an instance that the Thread
// will run when started.
Thread t = new Thread(new Runnable() {
        @Override
        public void run() {
              System.out.println("Hello world");
        }
    });
t.start();  // Prints "Hello world"
```

Anonymous inner classes can also be based on classes. In this case, the anonymous class implicitly **`extends`** the existing class. If the class being extended is abstract, then the anonymous class must implement all abstract methods. It may also override non-abstract methods.

Constructors

An anonymous class cannot have an explicit constructor. Instead, an implicit constructor is defined that uses **`super`**`(...)` to pass any parameters to a constructor in the class that is being extended. For example:

```
SomeClass anon = new SomeClass(1, "happiness") {
            @Override
            public int someMethod(int arg) {
                // do something
            }
        };
```

The implicit constructor for our anonymous subclass of `SomeClass` will call a constructor of `SomeClass` that matches the call signature `SomeClass(int, String)`. If no constructor is available, you will get a compilation error. Any exceptions that are thrown by the matched constructor are also thrown by the implicit constructor.

Naturally, this does not work when extending an interface. When you create an anonymous class from an interface, the classes superclass is `java.lang.Object` which only has a no-args constructor.

Section 50.5: Create instance of non-static inner class from outside

An inner class which is visible to any outside class can be created from this class as well.

The inner class depends on the outside class and requires a reference to an instance of it. To create an instance of the inner class, the **`new`** operator only needs to be called on an instance of the outer class.

```
class OuterClass {

    class InnerClass {
    }
}

class OutsideClass {

    OuterClass outer = new OuterClass();

    OuterClass.InnerClass createInner() {
        return outer.new InnerClass();
    }
}
```

Note the usage as `outer.new`.

Section 50.6: Method Local Inner Classes

A class written within a method called **method local inner class**. In that case the scope of the inner class is restricted within the method.

A method-local inner class can be instantiated only within the method where the inner class is defined.

The example of using method local inner class:

```
public class OuterClass {
    private void outerMethod() {
       final int outerInt = 1;
        // Method Local Inner Class
        class MethodLocalInnerClass {
            private void print() {
                System.out.println("Method local inner class " + outerInt);
            }
        }
        // Accessing the inner class
        MethodLocalInnerClass inner = new MethodLocalInnerClass();
        inner.print();
    }

    public static void main(String args[]) {
        OuterClass outer = new OuterClass();
        outer.outerMethod();
    }
}
```

Executing will give an output:

```
Method local inner class 1
```

.

Section 50.7: Accessing the outer class from a non-static inner class

The reference to the outer class uses the class name and `this`

```
public class OuterClass {
    public class InnerClass {
        public void method() {
            System.out.println("I can access my enclosing class: " + OuterClass.this);
        }
    }
}
```

You can access fields and methods of the outer class directly.

```
public class OuterClass {
    private int counter;

    public class InnerClass {
        public void method() {
            System.out.println("I can access " + counter);
```

```
        }
    }
}
```

But in case of name collision you can use the outer class reference.

```
public class OuterClass {
    private int counter;

    public class InnerClass {
        private int counter;

        public void method() {
            System.out.println("My counter: " + counter);
            System.out.println("Outer counter: " + OuterClass.this.counter);

            // updating my counter
            counter = OuterClass.this.counter;
        }
    }
}
```

Chapter 51: The java.util.Objects Class

Section 51.1: Basic use for object null check

For null check in method

```
Object nullableObject = methodReturnObject();
if (Objects.isNull(nullableObject)) {
    return;
}
```

For not null check in method

```
Object nullableObject = methodReturnObject();
if (Objects.nonNull(nullableObject)) {
    return;
}
```

Section 51.2: Objects.nonNull() method reference use in stream api

In the old fashion way for collection null check

```
List<Object> someObjects = methodGetList();
for (Object obj : someObjects) {
    if (obj == null) {
        continue;
    }
    doSomething(obj);
}
```

With the `Objects.nonNull` method and Java8 Stream API, we can do the above in this way:

```
List<Object> someObjects = methodGetList();
someObjects.stream()
           .filter(Objects::nonNull)
           .forEach(this::doSomething);
```

Chapter 52: Default Methods

Default Method introduced in Java 8, allows developers to add new methods to an interface without breaking the existing implementations of this interface. It provides flexibility to allow the interface to define an implementation which will be used as default when a class which implements that interface fails to provide an implementation of that method.

Section 52.1: Basic usage of default methods

```
/**
 * Interface with default method
 */
public interface Printable {
    default void printString() {
        System.out.println( "default implementation" );
    }
}

/**
 * Class which falls back to default implementation of {@link #printString()}
 */
public class WithDefault
    implements Printable
{
}

/**
 * Custom implementation of {@link #printString()}
 */
public class OverrideDefault
    implements Printable {
    @Override
    public void printString() {
        System.out.println( "overridden implementation" );
    }
}
```

The following statements

```
    new WithDefault().printString();
    new OverrideDefault().printString();
```

Will produce this output:

```
default implementation
```

```
overridden implementation
```

Section 52.2: Accessing overridden default methods from implementing class

In classes, `super.foo()` will look in superclasses only. If you want to call a default implementation from a superinterface, you need to qualify `super` with the interface name: `Fooable.super.foo()`.

```
public interface Fooable {
```

```
    default int foo() {return 3;}
}

public class A extends Object implements Fooable {
    @Override
    public int foo() {
        //return super.foo() + 1; //error: no method foo() in java.lang.Object
        return Fooable.super.foo() + 1; //okay, returns 4
    }
}
```

Section 52.3: Why use Default Methods?

The simple answer is that it allows you to evolve an existing interface without breaking existing implementations.

For example, you have `Swim` interface that you published 20 years ago.

```
public interface Swim {
    void backStroke();
}
```

We did a great job, our interface is very popular, there are many implementation on that around the world and you don't have control over their source code.

```
public class FooSwimmer implements Swim {
    public void backStroke() {
         System.out.println("Do backstroke");
    }
}
```

After 20 years, you've decided to add new functionality to the interface, but it looks like our interface is frozen because it will break existing implementations.

Luckily Java 8 introduces brand new feature called Default method.

We can now add new method to the `Swim` interface.

```
public interface Swim {
    void backStroke();
    default void sideStroke() {
        System.out.println("Default sidestroke implementation. Can be overridden");
    }
}
```

Now all existing implementations of our interface can still work. But most importantly they can implement the newly added method in their own time.

One of the biggest reasons for this change, and one of its biggest uses, is in the Java Collections framework. Oracle could not add a **`foreach`** method to the existing Iterable interface without breaking all existing code which implemented Iterable. By adding default methods, existing Iterable implementation will inherit the default implementation.

Section 52.4: Accessing other interface methods within default method

You can as well access other interface methods from within your default method.

```
public interface Summable {
    int getA();

    int getB();

    default int calculateSum() {
        return getA() + getB();
    }
}

public class Sum implements Summable {
    @Override
    public int getA() {
        return 1;
    }

    @Override
    public int getB() {
        return 2;
    }
}
```

The following statement will print *3*:

```
System.out.println(new Sum().calculateSum());
```

Default methods could be used along with interface static methods as well:

```
public interface Summable {
    static int getA() {
        return 1;
    }

    static int getB() {
        return 2;
    }

    default int calculateSum() {
        return getA() + getB();
    }
}

public class Sum implements Summable {}
```

The following statement will also print 3:

```
System.out.println(new Sum().calculateSum());
```

Section 52.5: Default method multiple inheritance collision

Consider next example:

```
public interface A {
    default void foo() { System.out.println("A.foo"); }
}

public interface B {
    default void foo() { System.out.println("B.foo"); }
```

```
}
```

Here are two interfaces declaring **`default`** method `foo` with the same signature.

If you will try to `extend` these both interfaces in the new interface you have to make choice of two, because Java forces you to resolve this collision explicitly.

First, you can declare method `foo` with the same signature as **`abstract`**, which will override `A` and `B` behaviour.

```
public interface ABExtendsAbstract extends A, B {
    @Override
    void foo();
}
```

And when you will `implement` `ABExtendsAbstract` in the **`class`** you will have to provide `foo` implementation:

```
public class ABExtendsAbstractImpl implements ABExtendsAbstract {
    @Override
    public void foo() { System.out.println("ABImpl.foo"); }
}
```

Or **second**, you can provide a completely new **`default`** implementation. You also may reuse code of `A` and `B` `foo` methods by Accessing overridden default methods from implementing class.

```
public interface ABExtends extends A, B {
    @Override
    default void foo() { System.out.println("ABExtends.foo"); }
}
```

And when you will `implement` `ABExtends` in the **`class`** you will `not` have to provide `foo` implementation:

```
public class ABExtendsImpl implements ABExtends {}
```

Section 52.6: Class, Abstract class and Interface method precedence

Implementations in classes, including abstract declarations, take precedence over all interface defaults.

- Abstract class method takes precedence over Interface Default Method.

```
public interface Swim {
    default void backStroke() {
        System.out.println("Swim.backStroke");
    }
}

public abstract class AbstractSwimmer implements Swim {
    public void backStroke() {
        System.out.println("AbstractSwimmer.backStroke");
    }
}

public class FooSwimmer extends AbstractSwimmer {
}
```

The following statement

```
new FooSwimmer().backStroke();
```

Will produce

```
AbstractSwimmer.backStroke
```

- Class method takes precedence over Interface Default Method

```
public interface Swim {
    default void backStroke() {
        System.out.println("Swim.backStroke");
    }
}

public abstract class AbstractSwimmer implements Swim {
}

public class FooSwimmer extends AbstractSwimmer {
    public void backStroke() {
        System.out.println("FooSwimmer.backStroke");
    }
}
```

The following statement

```
new FooSwimmer().backStroke();
```

Will produce

```
FooSwimmer.backStroke
```

Chapter 53: Packages

package in java is used to group class and interfaces. This helps developer to avoid conflict when there are huge numbers of classes. If we use this package the classes we can create a class/interface with same name in different packages. By using packages we can import the piece of again in another class. There many *built in packages* in java like > 1.java.util > 2.java.lang > 3.java.io We can define our own *user defined packages*.

Section 53.1: Using Packages to create classes with the same name

First Test.class:

```
package foo.bar

public class Test {

}
```

Also Test.class in another package

```
package foo.bar.baz

public class Test {

}
```

The above is fine because the two classes exist in different packages.

Section 53.2: Using Package Protected Scope

In Java if you don't provide an access modifier the default scope for variables is package-protected level. This means that classes can access the variables of other classes within the same package as if those variables were publicly available.

```
package foo.bar

public class ExampleClass {
    double exampleNumber;
    String exampleString;

    public ExampleClass() {
        exampleNumber = 3;
        exampleString = "Test String";
    }
    //No getters or setters
}

package foo.bar

public class AnotherClass {
    ExampleClass clazz = new ExampleClass();

    System.out.println("Example Number: " + clazz.exampleNumber);
    //Prints Example Number: 3
    System.out.println("Example String: " + clazz.exampleString);
    //Prints Example String: Test String
```

```
}
```

This method will not work for a class in another package:

```
package baz.foo

public class ThisShouldNotWork {
    ExampleClass clazz = new ExampleClass();

    System.out.println("Example Number: " + clazz.exampleNumber);
    //Throws an exception
    System.out.println("Example String: " + clazz.exampleString);
    //Throws an exception
}
```

Chapter 54: Inheritance

Inheritance is a basic object oriented feature in which one class acquires and extends upon the properties of another class, using the keyword **`extends`**. For Interfaces and the keyword **`implements`**, see interfaces.

Section 54.1: Inheritance

With the use of the **`extends`** keyword among classes, all the properties of the superclass (also known as the *Parent Class* or *Base Class*) are present in the subclass (also known as the *Child Class* or *Derived Class*)

```
public class BaseClass {

    public void baseMethod(){
        System.out.println("Doing base class stuff");
    }
}

public class SubClass extends BaseClass {

}
```

Instances of `SubClass` have *inherited* the method `baseMethod()`:

```
SubClass s = new SubClass();
s.baseMethod();  //Valid, prints "Doing base class stuff"
```

Additional content can be added to a subclass. Doing so allows for additional functionality in the subclass without any change to the base class or any other subclasses from that same base class:

```
public class Subclass2 extends BaseClass {

    public void anotherMethod() {
        System.out.println("Doing subclass2 stuff");
    }
}

Subclass2 s2 = new Subclass2();
s2.baseMethod(); //Still valid , prints "Doing base class stuff"
s2.anotherMethod(); //Also valid, prints "Doing subclass2 stuff"
```

Fields are also inherited:

```
public class BaseClassWithField {

    public int x;

}

public class SubClassWithField extends BaseClassWithField {

    public SubClassWithField(int x) {
        this.x = x; //Can access fields
    }
}
```

`private` fields and methods still exist within the subclass, but are not accessible:

```
public class BaseClassWithPrivateField {

    private int x = 5;

    public int getX() {
        return x;
    }
}

public class SubClassInheritsPrivateField extends BaseClassWithPrivateField {

    public void printX() {
        System.out.println(x); //Illegal, can't access private field x
        System.out.println(getX()); //Legal, prints 5
    }
}

SubClassInheritsPrivateField s = new SubClassInheritsPrivateField();
int x = s.getX(); //x will have a value of 5.
```

In Java, each class may extend at most one other class.

```
public class A{}
public class B{}
public class ExtendsTwoClasses extends A, B {} //Illegal
```

This is known as multiple inheritance, and while it is legal in some languages, Java does not permit it with classes.

As a result of this, every class has an unbranching ancestral chain of classes leading to `Object`, from which all classes descend.

Section 54.2: Abstract Classes

An abstract class is a class marked with the **`abstract`** keyword. It, contrary to non-abstract class, may contain abstract - implementation-less - methods. It is, however, valid to create an abstract class without abstract methods.

An abstract class cannot be instantiated. It can be sub-classed (extended) as long as the sub-class is either also abstract, or implements all methods marked as abstract by super classes.

An example of an abstract class:

```
public abstract class Component {
    private int x, y;

    public setPosition(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public abstract void render();
}
```

The class must be marked abstract, when it has at least one abstract method. An abstract method is a method that has no implementation. Other methods can be declared within an abstract class that have implementation in order to provide common code for any sub-classes.

Attempting to instantiate this class will provide a compile error:

```
//error: Component is abstract; cannot be instantiated
Component myComponent = new Component();
```

However a class that extends `Component`, and provides an implementation for all of its abstract methods and can be instantiated.

```
public class Button extends Component {

    @Override
    public void render() {
        //render a button
    }
}

public class TextBox extends Component {

    @Override
    public void render() {
        //render a textbox
    }
}
```

Instances of inheriting classes also can be cast as the parent class (normal inheritance) and they provide a polymorphic effect when the abstract method is called.

```
Component myButton = new Button();
Component myTextBox = new TextBox();

myButton.render(); //renders a button
myTextBox.render(); //renders a text box
```

Abstract classes vs Interfaces

Abstract classes and interfaces both provide a way to define method signatures while requiring the extending/implementing class to provide the implementation.

There are two key differences between abstract classes and interfaces:

- A class may only extend a single class, but may implement many interfaces.
- An abstract class can contain instance (non-`static`) fields, but interfaces may only contain `static` fields.

Version < Java SE 8

Methods declared in interfaces could not contain implementations, so abstract classes were used when it was useful to provide additional methods which implementations called the abstract methods.

Version ≥ Java SE 8

Java 8 allows interfaces to contain default methods, usually implemented using the other methods of the interface, making interfaces and abstract classes equally powerful in this regard.

Anonymous subclasses of Abstract Classes

As a convenience java allows for instantiation of anonymous instances of subclasses of abstract classes, which provide implementations for the abstract methods upon creating the new object. Using the above example this could look like this:

```
Component myAnonymousComponent = new Component() {
    @Override
    public void render() {
        // render a quick 1-time use component
    }
}
```

Section 54.3: Using 'final' to restrict inheritance and overriding

Final classes

When used in a **`class`** declaration, the **`final`** modifier prevents other classes from being declared that `extend` the class. A **`final`** class is a "leaf" class in the inheritance class hierarchy.

```
// This declares a final class
final class MyFinalClass {
    /* some code */
}

// Compilation error: cannot inherit from final MyFinalClass
class MySubClass extends MyFinalClass {
    /* more code */
}
```

Use-cases for final classes

Final classes can be combined with a **`private`** constructor to control or prevent the instantiation of a class. This can be used to create a so-called "utility class" that only defines static members; i.e. constants and static methods.

```
public final class UtilityClass {

    // Private constructor to replace the default visible constructor
    private UtilityClass() {}

    // Static members can still be used as usual
    public static int doSomethingCool() {
        return 123;
    }

}
```

Immutable classes should also be declared as **`final`**. (An immutable class is one whose instances cannot be changed after they have been created; see the Immutable Objects topic.) By doing this, you make it impossible to create a mutable subclass of an immutable class. That would violate the Liskov Substitution Principle which requires that a subtype should obey the "behavioral contract" of its supertypes.

From a practical perspective, declaring an immutable class to be **`final`** makes it easier to reason about program behavior. It also addresses security concerns in the scenario where untrusted code is executed in a security sandbox. (For instance, since `String` is declared as **`final`**, a trusted class does not need to worry that it might be tricked into accepting mutable subclass, which the untrusted caller could then surreptitiously change.)

One disadvantage of **`final`** classes is that they do not work with some mocking frameworks such as Mockito. Update: Mockito version 2 now support mocking of final classes.

Final methods

The `final` modifier can also be applied to methods to prevent them being overridden in sub-classes:

```
public class MyClassWithFinalMethod {

    public final void someMethod() {
    }
}

public class MySubClass extends MyClassWithFinalMethod {

    @Override
    public void someMethod() { // Compiler error (overridden method is final)
    }
}
```

Final methods are typically used when you want to restrict what a subclass can change in a class without forbidding subclasses entirely.

The `final` modifier can also be applied to variables, but the meaning of `final` for variables is unrelated to inheritance.

Section 54.4: The Liskov Substitution Principle

Substitutability is a principle in object-oriented programming introduced by Barbara Liskov in a 1987 conference keynote stating that, if class `B` is a subclass of class `A`, then wherever `A` is expected, `B` can be used instead:

```
class A {...}
class B extends A {...}

public void method(A obj) {...}

A a = new B(); // Assignment OK
method(new B()); // Passing as parameter OK
```

This also applies when the type is an interface, where there doesn't need to any hierarchical relationship between the objects:

```
interface Foo {
    void bar();
}

class A implements Foo {
    void bar() {...}
}

class B implements Foo {
    void bar() {...}
}

List<Foo> foos = new ArrayList<>();
foos.add(new A()); // OK
foos.add(new B()); // OK
```

Now the list contains objects that are not from the same class hierarchy.

Section 54.5: Abstract class and Interface usage: "Is-a" relation vs "Has-a" capability

When to use abstract classes: To implement the same or different behaviour among multiple related objects

When to use interfaces: to implement a contract by multiple unrelated objects

Abstract classes create "is a" relations while interfaces provide "has a" capability.

This can be seen in the code below:

```
public class InterfaceAndAbstractClassDemo{
    public static void main(String args[]){

        Dog dog = new Dog("Jack",16);
        Cat cat = new Cat("Joe",20);

        System.out.println("Dog:"+dog);
        System.out.println("Cat:"+cat);

        dog.remember();
        dog.protectOwner();
        Learn dl = dog;
        dl.learn();

        cat.remember();
        cat.protectOwner();

        Climb c = cat;
        c.climb();

        Man man = new Man("Ravindra",40);
        System.out.println(man);

        Climb cm = man;
        cm.climb();
        Think t = man;
        t.think();
        Learn l = man;
        l.learn();
        Apply a = man;
        a.apply();
    }
}

abstract class Animal{
    String name;
    int lifeExpentency;
    public Animal(String name,int lifeExpentency ){
        this.name = name;
        this.lifeExpentency=lifeExpentency;
    }
    public abstract void remember();
    public abstract void protectOwner();

    public String toString(){
        return this.getClass().getSimpleName()+":"+name+":"+lifeExpentency;
    }
}
class Dog extends Animal implements Learn{
```

```
    public Dog(String name,int age){
        super(name,age);
    }
    public void remember(){
        System.out.println(this.getClass().getSimpleName()+" can remember for 5 minutes");
    }
    public void protectOwner(){
        System.out.println(this.getClass().getSimpleName()+ " will protect owner");
    }
    public void learn(){
        System.out.println(this.getClass().getSimpleName()+ " can learn:");
    }
}
class Cat extends Animal implements Climb {
    public Cat(String name,int age){
        super(name,age);
    }
    public void remember(){
        System.out.println(this.getClass().getSimpleName() + " can remember for 16 hours");
    }
    public void protectOwner(){
        System.out.println(this.getClass().getSimpleName()+ " won't protect owner");
    }
    public void climb(){
        System.out.println(this.getClass().getSimpleName()+ " can climb");
    }
}
interface Climb{
    void climb();
}
interface Think {
    void think();
}

interface Learn {
    void learn();
}
interface Apply{
    void apply();
}

class Man implements Think,Learn,Apply,Climb{
    String name;
    int age;

    public Man(String name,int age){
        this.name = name;
        this.age = age;
    }
    public void think(){
        System.out.println("I can think:"+this.getClass().getSimpleName());
    }
    public void learn(){
        System.out.println("I can learn:"+this.getClass().getSimpleName());
    }
    public void apply(){
        System.out.println("I can apply:"+this.getClass().getSimpleName());
    }
    public void climb(){
        System.out.println("I can climb:"+this.getClass().getSimpleName());
    }
    public String toString(){
```

```
        return "Man :"+name+":Age:"+age;
    }
}
```

output:

```
Dog:Dog:Jack:16
Cat:Cat:Joe:20
Dog can remember for 5 minutes
Dog will protect owner
Dog can learn:
Cat can remember for 16 hours
Cat won't protect owner
Cat can climb
Man :Ravindra:Age:40
I can climb:Man
I can think:Man
I can learn:Man
I can apply:Man
```

Key notes:

1. `Animal` is an abstract class with shared attributes: `name` and `lifeExpectancy` and abstract methods: `remember()` and `protectOwner()`. `Dog` and `Cat` are `Animals` that have implemented the `remember()` and `protectOwner()` methods.
2. `Cat` can `climb()` but `Dog` cannot. `Dog` can `think()` but `Cat` cannot. These specific capabilities are added to `Cat` and `Dog` by implementation.
3. `Man` is not an `Animal` but he can `Think` , `Learn`, `Apply`, and `Climb`.
4. `Cat` is not a `Man` but it can `Climb`.
5. `Dog` is not a `Man` but it can `Learn`
6. `Man` is neither a `Cat` nor a `Dog` but can have some of the capabilities of the latter two without extending `Animal`, `Cat`, or `Dog`. This is done with Interfaces.
7. Even though `Animal` is an abstract class, it has a constructor, unlike an interface.

TL;DR:

Unrelated classes can have capabilities through interfaces, but related classes change the behaviour through extension of base classes.

Refer to the Java documentation page to understand which one to use in a specific use case.

Consider using abstract classes if...

1. You want to share code among several closely related classes.
2. You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
3. You want to declare non-static or non-final fields.

Consider using interfaces if...

1. You expect that unrelated classes would implement your interface. For example, many unrelated objects can implement the `Serializable` interface.

2. You want to specify the behaviour of a particular data type but are not concerned about who implements its behaviour.
3. You want to take advantage of multiple inheritance of type.

Section 54.6: Static Inheritance

Static method can be inherited similar to normal methods, however unlike normal methods it is impossible to create "abstract" methods in order to force static method overriding. Writing a method with the same signature as a static method in a super class appears to be a form of overriding, but really this simply creates a new function hides the other.

```
public class BaseClass {

    public static int num = 5;

    public static void sayHello() {
        System.out.println("Hello");
    }

    public static void main(String[] args) {
        BaseClass.sayHello();
        System.out.println("BaseClass's num: " + BaseClass.num);

        SubClass.sayHello();
        //This will be different than the above statement's output, since it runs
        //A different method
        SubClass.sayHello(true);

        StaticOverride.sayHello();
        System.out.println("StaticOverride's num: " + StaticOverride.num);
    }
}

public  class SubClass extends BaseClass {

    //Inherits the sayHello function, but does not override it
    public static void sayHello(boolean test) {
        System.out.println("Hey");
    }
}

public static class StaticOverride extends BaseClass {

    //Hides the num field from BaseClass
    //You can even change the type, since this doesn't affect the signature
    public static String num = "test";

    //Cannot use @Override annotation, since this is static
    //This overrides the sayHello method from BaseClass
    public static void sayHello() {
        System.out.println("Static says Hi");
    }

}
```

Running any of these classes produces the output:

```
Hello
BaseClass's num: 5
```

```
Hello
Hey
Static says Hi
StaticOverride's num: test
```

Note that unlike normal inheritance, in static inheritance methods are not hidden. You can always call the base `sayHello` method by using `BaseClass.sayHello()`. But classes do inherit static methods *if* no methods with the same signature are found in the subclass. If two method's signatures vary, both methods can be run from the subclass, even if the name is the same.

Static fields hide each other in a similar way.

Section 54.7: Programming to an interface

The idea behind programming to an interface is to base the code primarily on interfaces and only use concrete classes at the time of instantiation. In this context, good code dealing with e.g. Java collections will look something like this (not that the method itself is of any use at all, just illustration):

```
public <T> Set<T> toSet(Collection<T> collection) {
  return Sets.newHashSet(collection);
}
```

while bad code might look like this:

```
public <T> HashSet<T> toSet(ArrayList<T> collection) {
  return Sets.newHashSet(collection);
}
```

Not only the former can be applied to a wider choice of arguments, its results will be more compatible with code provided by other developers that generally adhere to the concept of programming to an interface. However, the most important reasons to use the former are:

- most of the time the context, in which the result is used, does not and should not need that many details as the concrete implementation provides;
- adhering to an interface forces cleaner code and less hacks such as yet another public method gets added to a class serving some specific scenario;
- the code is more testable as interfaces are easily mockable;
- finally, the concept helps even if only one implementation is expected (at least for testability).

So how can one easily apply the concept of programming to an interface when writing new code having in mind one particular implementation? One option that we commonly use is a combination of the following patterns:

- programming to an interface
- factory
- builder

The following example based on these principles is a simplified and truncated version of an RPC implementation written for a number of different protocols:

```
public interface RemoteInvoker {
  <RQ, RS> CompletableFuture<RS> invoke(RQ request, Class<RS> responseClass);
}
```

The above interface is not supposed to be instantiated directly via a factory, instead we derive further more

concrete interfaces, one for HTTP invocation and one for AMQP, each then having a factory and a builder to construct instances, which in turn are also instances of the above interface:

```
public interface AmqpInvoker extends RemoteInvoker {
  static AmqpInvokerBuilder with(String instanceId, ConnectionFactory factory) {
    return new AmqpInvokerBuilder(instanceId, factory);
  }
}
```

Instances of `RemoteInvoker` for the use with AMQP can now be constructed as easy as (or more involved depending on the builder):

```
RemoteInvoker invoker = AmqpInvoker.with(instanceId, factory)
  .requestRouter(router)
  .build();
```

And an invocation of a request is as easy as:

```
Response res = invoker.invoke(new Request(data), Response.class).get();
```

Due to Java 8 permitting placing of static methods directly into interfaces, the intermediate factory has become implicit in the above code replaced with `AmqpInvoker.with()`. In Java prior to version 8, the same effect can be achieved with an inner `Factory` class:

```
public interface AmqpInvoker extends RemoteInvoker {
  class Factory {
    public static AmqpInvokerBuilder with(String instanceId, ConnectionFactory factory) {
      return new AmqpInvokerBuilder(instanceId, factory);
    }
  }
}
```

The corresponding instantiation would then turn into:

```
RemoteInvoker invoker = AmqpInvoker.Factory.with(instanceId, factory)
  .requestRouter(router)
  .build();
```

The builder used above could look like this (although this is a simplification as the actual one permits defining of up to 15 parameters deviating from defaults). Note that the construct is not public, so it is specifically usable only from the above `AmqpInvoker` interface:

```
public class AmqpInvokerBuilder {
  ...
  AmqpInvokerBuilder(String instanceId, ConnectionFactory factory) {
    this.instanceId = instanceId;
    this.factory = factory;
  }

  public AmqpInvokerBuilder requestRouter(RequestRouter requestRouter) {
    this.requestRouter = requestRouter;
    return this;
  }

  public AmqpInvoker build() throws TimeoutException, IOException {
    return new AmqpInvokerImpl(instanceId, factory, requestRouter);
  }
```

```
}
```

Generally, a builder can also be generated using a tool like FreeBuilder.

Finally, the standard (and the only expected) implementation of this interface is defined as a package-local class to enforce the use of the interface, the factory and the builder:

```
class AmqpInvokerImpl implements AmqpInvoker {
  AmqpInvokerImpl(String instanceId, ConnectionFactory factory, RequestRouter requestRouter) {
    ...
  }

  @Override
  public <RQ, RS> CompletableFuture<RS> invoke(final RQ request, final Class<RS> respClass) {
    ...
  }
}
```

Meanwhile, this pattern proved to be very efficient in developing all our new code not matter how simple or complex the functionality is.

Section 54.8: Overriding in Inheritance

Overriding in Inheritance is used when you use a already defined method from a super class in a sub class, but in a different way than how the method was originally designed in the super class. Overriding allows the user to reuse code by using existing material and modifying it to suit the user's needs better.

The following example demonstrates how `ClassB` overrides the functionality of `ClassA` by changing what gets sent out through the printing method:

Example:

```
public static void main(String[] args) {
    ClassA a = new ClassA();
    ClassA b = new ClassB();
    a.printing();
    b.printing();
}

class ClassA {
    public void printing() {
        System.out.println("A");
    }
}

class ClassB extends ClassA {
    public void printing() {
         System.out.println("B");
    }
}
```

Output:

```
A
```

> B

Section 54.9: Variable shadowing

Variables are SHADOWED and methods are OVERRIDDEN. Which variable will be used depends on the class that the variable is declared of. Which method will be used depends on the actual class of the object that is referenced by the variable.

```
class Car {
    public int gearRatio = 8;

    public String accelerate() {
        return "Accelerate : Car";
    }
}

class SportsCar extends Car {
    public int gearRatio = 9;

    public String accelerate() {
        return "Accelerate : SportsCar";
    }

    public void test() {

    }


    public static void main(String[] args) {

        Car car = new SportsCar();
        System.out.println(car.gearRatio + "   " + car.accelerate());
        // will print out 8  Accelerate : SportsCar
    }
}
```

Section 54.10: Narrowing and Widening of object references

Casting an instance of a base class to a subclass as in : `b = (B) a;` is called *narrowing* (as you are trying to narrow the base class object to a more specific class object) and needs an explicit type-cast.

Casting an instance of a subclass to a base class as in: `A a = b;` is called *widening* and does not need a type-cast.

To illustrate, consider the following class declarations, and test code:

```
class Vehicle {
}

class Car extends Vehicle {
}

class Truck extends Vehicle {
}

class MotorCycle extends Vehicle {
}

class Test {
```

```
    public static void main(String[] args) {

        Vehicle vehicle = new Car();
        Car car = new Car();

        vehicle = car; // is valid, no cast needed

        Car c = vehicle // not valid
        Car c = (Car) vehicle; //valid
    }
}
```

The statement `Vehicle vehicle = new Car();` is a valid Java statement. Every instance of `Car` is also a `Vehicle`. Therefore, the assignment is legal without the need for an explicit type-cast.

On the other hand, `Car c = vehicle;` is not valid. The static type of the `vehicle` variable is `Vehicle` which means that it could refer to an instance of `Car`, Truck, MotorCycle, `or any other current or future subclass` ofVehicle. `(Or indeed, an instance ofVehicleitself, since we did not declare it as anabstract`**`class`**`.) The assignment cannot be allowed, since that might lead tocarreferring to aTruck` instance.

To prevent this situation, we need to add an explicit type-cast:

```
Car c = (Car) vehicle;
```

The type-cast tells the compiler that we *expect* the value of `vehicle` to be a `Car` or a subclass of `Car`. If necessary, compiler will insert code to perform a run-time type check. If the check fails, then a `ClassCastException` will be thrown when the code is executed.

Note that not all type-casts are valid. For example:

```
String s = (String) vehicle;  // not valid
```

The Java compiler knows that an instance that is type compatible with `Vehicle` *cannot ever be* type compatible with `String`. The type-cast could never succeed, and the JLS mandates that this gives in a compilation error.

Section 54.11: Inheritance and Static Methods

In Java, parent and child class both can have static methods with the same name. But in such cases implementation of static method in child is hiding parent class' implementation, it's not method overriding. For example:

```
class StaticMethodTest {

  // static method and inheritance
  public static void main(String[] args) {
    Parent p = new Child();
    p.staticMethod(); // prints Inside Parent
    ((Child) p).staticMethod(); // prints Inside Child
  }

  static class Parent {
    public static void staticMethod() {
      System.out.println("Inside Parent");
    }
  }

  static class Child extends Parent {
    public static void staticMethod() {
```

```
        System.out.println("Inside Child");
    }
  }
}
```

Static methods are bind to a class not to an instance and this method binding happens at compile time. Since in the first call to `staticMethod()`, parent class reference `p` was used, `Parent`'s version of `staticMethod()` is invoked. In second case, we did cast `p` into `Child` class, `Child`'s `staticMethod()` executed.

Chapter 55: Reference Types

Section 55.1: Different Reference Types

`java.lang.ref` package provides reference-object classes, which support a limited degree of interaction with the garbage collector.

Java has four main different reference types. They are:

- Strong Reference
- Weak Reference
- Soft Reference
- Phantom Reference

1. Strong Reference

This is the usual form of creating objects.

```
MyObject myObject = new MyObject();
```

The variable holder is holding a strong reference to the object created. As long as this variable is live and holds this value, the `MyObject` instance will not be collected by the garbage collector.

2. Weak Reference

When you do not want to keep an object longer, and you need to clear/free the memory allocated for an object as soon as possible, this is the way to do so.

```
WeakReference myObjectRef = new WeakReference(MyObject);
```

Simply, a weak reference is a reference that isn't strong enough to force an object to remain in memory. Weak references allow you to leverage the garbage collector's ability to determine reachability for you, so you don't have to do it yourself.

When you need the object you created, just use `.get()` method:

```
myObjectRef.get();
```

Following code will exemplify this:

```
WeakReference myObjectRef = new WeakReference(MyObject);
System.out.println(myObjectRef.get()); // This will print the object reference address
System.gc();
System.out.println(myObjectRef.get()); // This will print 'null' if the GC cleaned up the object
```

3. Soft Reference

Soft references are slightly stronger than weak references. You can create a soft referenced object as following:

```
SoftReference myObjectRef = new SoftReference(MyObject);
```

They can hold onto the memory more strongly than the weak reference. If you have enough memory supply/resources, garbage collector will not clean the soft references as enthusiastically as weak references.

Soft references are handy to use in caching. You can create soft referenced objects as a cache, where they kept until your memory runs out. When your memory can't supply enough resources, garbage collector will remove soft references.

```
SoftReference myObjectRef = new SoftReference(MyObject);
System.out.println(myObjectRef.get()); // This will print the reference address of the Object
System.gc();
System.out.println(myObjectRef.get()); // This may or may not print the reference address of the
Object
```

4. Phantom Reference

This is the weakest referencing type. If you created an object reference using Phantom Reference, the `get()` method will always return null!

The use of this referencing is that "Phantom reference objects, which are enqueued after the collector determines that their referents may otherwise be reclaimed. Phantom references are most often used for scheduling pre-mortem cleanup actions in a more flexible way than is possible with the Java finalization mechanism." - From Phantom Reference Javadoc from Oracle.

You can create an object of Phantom Reference as following:

```
PhantomReference myObjectRef = new PhantomReference(MyObject);
```

Chapter 56: Console I/O

Section 56.1: Reading user input from the console

Using `BufferedReader`:

```
System.out.println("Please type your name and press Enter.");

BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
try {
    String name = reader.readLine();
    System.out.println("Hello, " + name + "!");
} catch(IOException e) {
    System.out.println("An error occurred: " + e.getMessage());
}
```

The following imports are needed for this code:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
```

Using `Scanner`:

`Version ≥ Java SE 5`

```
System.out.println("Please type your name and press Enter");

Scanner scanner = new Scanner(System.in);
String name = scanner.nextLine();

System.out.println("Hello, " + name + "!");
```

The following import is needed for this example:

```
import java.util.Scanner;
```

To read more than one line, invoke `scanner.nextLine()` repeatedly:

```
System.out.println("Please enter your first and your last name, on separate lines.");

Scanner scanner = new Scanner(System.in);
String firstName = scanner.nextLine();
String lastName = scanner.nextLine();

System.out.println("Hello, " + firstName + " " + lastName + "!");
```

There are two methods for obtaining `Strings`, `next()` and `nextLine()`. `next()` returns text up until the first space (also known as a "token"), and `nextLine()` returns all text that the user inputted until pressing enter.

`Scanner` also provides utility methods for reading data types other than `String`. These include:

```
scanner.nextByte();
scanner.nextShort();
scanner.nextInt();
scanner.nextLong();
scanner.nextFloat();
scanner.nextDouble();
scanner.nextBigInteger();
```

```
scanner.nextBigDecimal();
```

Prefixing any of these methods with `has` (as in `hasNextLine()`, `hasNextInt()`) returns **`true`** if the stream has any more of the request type. Note: These methods will crash the program if the input is not of the requested type (for example, typing "a" for `nextInt()`). You can use a **`try`** `{}` **`catch`**`()` `{}` to prevent this (see: Exceptions)

```
Scanner scanner = new Scanner(System.in); //Create the scanner
scanner.useLocale(Locale.US); //Set number format excepted
System.out.println("Please input a float, decimal separator is .");
if (scanner.hasNextFloat()){ //Check if it is a float
    float fValue = scanner.nextFloat(); //retrive the value directly as float
    System.out.println(fValue + " is a float");
}else{
    String sValue = scanner.next(); //We can not retrive as float
    System.out.println(sValue + " is not a float");
}
```

Using `System.console`:

`Version ≥ Java SE 6`

```
String name = System.console().readLine("Please type your name and press Enter%n");

System.out.printf("Hello, %s!", name);

//To read passwords (without echoing as in unix terminal)
char[] password = System.console().readPassword();
```

Advantages:

- Reading methods are synchronized
- Format string syntax can be used

Note: This will only work if the program is run from a real command line without redirecting the standard input and output streams. It does not work when the program is run from within certain IDEs, such as Eclipse. For code that works within IDEs and with stream redirection, see the other examples.

Section 56.2: Aligning strings in console

The method `PrintWriter.format` (called through `System.out.format`) can be used to print aligned strings in console. The method receives a `String` with the format information and a series of objects to format:

```
String rowsStrings[] = new String[] {"1",
                                     "1234",
                                     "1234567",
                                     "123456789"};

String column1Format = "%-3s";    // min 3 characters, left aligned
String column2Format = "%-5.8s";  // min 5 and max 8 characters, left aligned
String column3Format = "%6.6s";   // fixed size 6 characters, right aligned
String formatInfo = column1Format + " " + column2Format + " " + column3Format;

for(int i = 0; i < rowsStrings.length; i++) {
    System.out.format(formatInfo, rowsStrings[i], rowsStrings[i], rowsStrings[i]);
    System.out.println();
}
```

Output:

```
1   1          1
1234 1234    1234
1234567 1234567 123456
123456789 12345678 123456
```

Using format strings with fixed size permits to print the strings in a table-like appearance with fixed size columns:

```
String rowsStrings[] = new String[] {"1",
                                     "1234",
                                     "1234567",
                                     "123456789"};

String column1Format = "%-3.3s";  // fixed size 3 characters, left aligned
String column2Format = "%-8.8s";  // fixed size 8 characters, left aligned
String column3Format = "%6.6s";   // fixed size 6 characters, right aligned
String formatInfo = column1Format + " " + column2Format + " " + column3Format;

for(int i = 0; i < rowsStrings.length; i++) {
    System.out.format(formatInfo, rowsStrings[i], rowsStrings[i], rowsStrings[i]);
    System.out.println();
}
```

Output:

```
1   1             1
123 1234       1234
123 1234567  123456
123 12345678 123456
```

Format strings examples

- `%s`: just a string with no formatting
- `%5s`: format the string with a **minimum** of 5 characters; if the string is shorter it will be **padded** to 5 characters and **right** aligned
- `%-5s`: format the string with a **minimum** of 5 characters; if the string is shorter it will be **padded** to 5 characters and **left** aligned
- `%5.10s`: format the string with a **minimum** of 5 characters and a **maximum** of 10 characters; if the string is shorter than 5 it will be **padded** to 5 characters and **right** aligned; if the string is longer than 10 it will be **truncated** to 10 characters and **right** aligned
- `%-5.5s`: format the string with a **fixed** size of 5 characters (minimum and maximum are equals); if the string is shorter than 5 it will be **padded** to 5 characters and **left** aligned; if the string is longer than 5 it will be **truncated** to 5 characters and **left** aligned

Section 56.3: Implementing Basic Command-Line Behavior

For basic prototypes or basic command-line behavior, the following loop comes in handy.

```
public class ExampleCli {

    private static final String CLI_LINE   = "example-cli>"; //console like string

    private static final String CMD_QUIT   = "quit";    //string for exiting the program
    private static final String CMD_HELLO  = "hello";    //string for printing "Hello World!" on
the screen
    private static final String CMD_ANSWER = "answer";    //string for printing 42 on the screen
```

```
    public static void main(String[] args) {
        ExampleCli claimCli = new ExampleCli();    // creates an object of this class

        try {
            claimCli.start();    //calls the start function to do the work like console
        }
        catch (IOException e) {
            e.printStackTrace();    //prints the exception log if it is failed to do get the user
input or something like that
        }
    }

    private void start() throws IOException {
        String cmd = "";

        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        while (!cmd.equals(CMD_QUIT)) {    // terminates console if user input is "quit"
            System.out.print(CLI_LINE);    //prints the console-like string

            cmd = reader.readLine();    //takes input from user. user input should be started with
"hello",  "answer" or "quit"
            String[] cmdArr = cmd.split(" ");

            if (cmdArr[0].equals(CMD_HELLO)) {    //executes when user input starts with "hello"
                hello(cmdArr);
            }
            else if (cmdArr[0].equals(CMD_ANSWER)) {    //executes when user input starts with
"answer"
                answer(cmdArr);
            }
        }
    }

    // prints "Hello World!" on the screen if user input starts with "hello"
    private void hello(String[] cmdArr) {
        System.out.println("Hello World!");
    }

    // prints "42" on the screen if user input starts with "answer"
    private void answer(String[] cmdArr) {
        System.out.println("42");
    }
}
```

Chapter 57: Streams

A `Stream` represents a sequence of elements and supports different kind of operations to perform computations upon those elements. With Java 8, `Collection` interface has two methods to generate a `Stream`: `stream()` and `parallelStream()`. `Stream` operations are either intermediate or terminal. Intermediate operations return a `Stream` so multiple intermediate operations can be chained before the `Stream` is closed. Terminal operations are either void or return a non-stream result.

Section 57.1: Using Streams

A `Stream` is a sequence of elements upon which sequential and parallel aggregate operations can be performed. Any given `Stream` can potentially have an unlimited amount of data flowing through it. As a result, data received from a `Stream` is processed individually as it arrives, as opposed to performing batch processing on the data altogether. When combined with lambda expressions they provide a concise way to perform operations on sequences of data using a functional approach.

Example: (see it work on Ideone)

```
Stream<String> fruitStream = Stream.of("apple", "banana", "pear", "kiwi", "orange");

fruitStream.filter(s -> s.contains("a"))
           .map(String::toUpperCase)
           .sorted()
           .forEach(System.out::println);
```

Output:

> APPLE
> BANANA
> ORANGE
> PEAR

The operations performed by the above code can be summarized as follows:

1. Create a `Stream<String>` containing a sequenced ordered `Stream` of fruit `String` elements using the static factory method `Stream.of(values)`.
2. The `filter()` operation retains only elements that match a given predicate (the elements that when tested by the predicate return true). In this case, it retains the elements containing an `"a"`. The predicate is given as a lambda expression.
3. The `map()` operation transforms each element using a given function, called a mapper. In this case, each fruit `String` is mapped to its uppercase `String` version using the method-reference `String::toUppercase`.

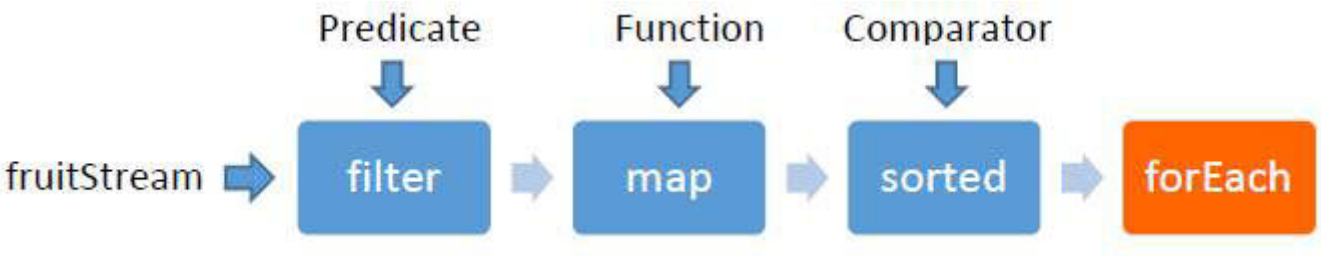
   > **Note** that the `map()` operation will return a stream with a different generic type if the mapping function returns a type different to its input parameter. For example on a `Stream<String>` calling `.map(String::isEmpty)` returns a `Stream<Boolean>`

4. The `sorted()` operation sorts the elements of the `Stream` according to their natural ordering

(lexicographically, in the case of `String`).

5. Finally, the **`forEach`**`(action)` operation performs an action which acts on each element of the `Stream`, passing it to a Consumer. In the example, each element is simply being printed to the console. This operation is a terminal operation, thus making it impossible to operate on it again.

> **Note** that operations defined on the `Stream` are performed *because* of the terminal operation. Without a terminal operation, the stream is not processed. Streams can not be reused. Once a terminal operation is called, the `Stream` object becomes unusable.



Operations (as seen above) are chained together to form what can be seen as a query on the data.

Closing Streams

> **Note that a `Stream` generally does not have to be closed.** It is only required to close streams that operate on IO channels. Most `Stream` types don't operate on resources and therefore don't require closing.

The `Stream` interface extends `AutoCloseable`. Streams can be closed by calling the `close` method or by using try-with-resource statements.

An example use case where a `Stream` should be closed is when you create a `Stream` of lines from a file:

```
try (Stream<String> lines = Files.lines(Paths.get("somePath"))) {
    lines.forEach(System.out::println);
}
```

The `Stream` interface also declares the `Stream.onClose()` method which allows you to register `Runnable` handlers which will be called when the stream is closed. An example use case is where code which produces a stream needs to know when it is consumed to perform some cleanup.

```
public Stream<String>streamAndDelete(Path path) throws IOException {
    return Files.lines(path).onClose(() -> someClass.deletePath(path));
}
```

The run handler will only execute if the `close()` method gets called, either explicitly or implicitly by a try-with-resources statement.

Processing Order

A `Stream` object's processing can be sequential or parallel.

In a **sequential** mode, the elements are processed in the order of the source of the `Stream`. If the `Stream` is ordered (such as a `SortedMap` implementation or a `List`) the processing is guaranteed to match the ordering of the source. In other cases, however, care should be taken not to depend on the ordering (see: is the Java `HashMap` `keySet()` iteration order consistent?).

Example:

```
List<Integer> integerList = Arrays.asList(0, 1, 2, 3, 42);

// sequential
long howManyOddNumbers = integerList.stream()
                                    .filter(e -> (e % 2) == 1)
                                    .count();

System.out.println(howManyOddNumbers); // Output: 2
```

Live on Ideone

Parallel mode allows the use of multiple threads on multiple cores but there is no guarantee of the order in which elements are processed.

If multiple methods are called on a sequential `Stream`, not every method has to be invoked. For example, if a `Stream` is filtered and the number of elements is reduced to one, a subsequent call to a method such as `sort` will not occur. This can increase the performance of a sequential `Stream` — an optimization that is not possible with a parallel `Stream`.

Example:

```
// parallel
long howManyOddNumbersParallel = integerList.parallelStream()
                                            .filter(e -> (e % 2) == 1)
                                            .count();

System.out.println(howManyOddNumbersParallel); // Output: 2
```

Live on Ideone

Differences from Containers (or Collections)

While some actions can be performed on both Containers and Streams, they ultimately serve different purposes and support different operations. Containers are more focused on how the elements are stored and how those elements can be accessed efficiently. A `Stream`, on the other hand, doesn't provide direct access and manipulation to its elements; it is more dedicated to the group of objects as a collective entity and performing operations on that entity as a whole. `Stream` and `Collection` are separate high-level abstractions for these differing purposes.

Section 57.2: Consuming Streams

A `Stream` will only be traversed when there is a *terminal operation*, like `count()`, `collect()` or **`forEach`**`()`. Otherwise, no operation on the `Stream` will be performed.

In the following example, no terminal operation is added to the `Stream`, so the `filter()` operation will not be invoked and no output will be produced because `peek()` is NOT a *terminal operation*.

```
IntStream.range(1, 10).filter(a -> a % 2 == 0).peek(System.out::println);
```

Live on Ideone

This is a `Stream` sequence with a valid *terminal operation*, thus an output is produced.

You could also use **`forEach`** instead of `peek`:

```
IntStream.range(1, 10).filter(a -> a % 2 == 0).forEach(System.out::println);
```

Live on Ideone

Output:

> 2
> 4
> 6
> 8

After the terminal operation is performed, the `Stream` is consumed and cannot be reused.

Although a given stream object cannot be reused, it's easy to create a reusable `Iterable` that delegates to a stream pipeline. This can be useful for returning a modified view of a live data set without having to collect results into a temporary structure.

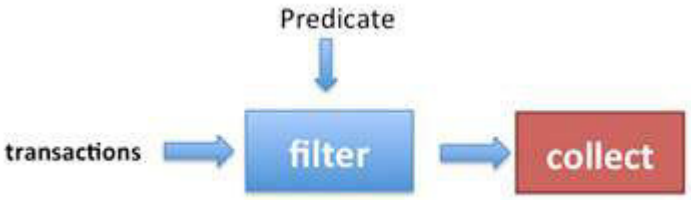
```
List<String> list = Arrays.asList("FOO", "BAR");
Iterable<String> iterable = () -> list.stream().map(String::toLowerCase).iterator();

for (String str : iterable) {
    System.out.println(str);
}
for (String str : iterable) {
    System.out.println(str);
}
```

Output:

> foo
> bar
> foo
> bar

This works because `Iterable` declares a single abstract method `Iterator<T> iterator()`. That makes it effectively a functional interface, implemented by a lambda that creates a new stream on each call.

In general, a `Stream` operates as shown in the following image:



NOTE: Argument checks are always performed, even without a *terminal operation*:

```
try {
    IntStream.range(1, 10).filter(null);
} catch (NullPointerException e) {
    System.out.println("We got a NullPointerException as null was passed as an argument to
filter()");
}
```

Live on Ideone

Output:

> We got a NullPointerException as null was passed as an argument to filter()

Section 57.3: Creating a Frequency Map

The `groupingBy(classifier, downstream)` collector allows the collection of `Stream` elements into a `Map` by classifying each element in a group and performing a downstream operation on the elements classified in the same group.

A classic example of this principle is to use a `Map` to count the occurrences of elements in a `Stream`. In this example, the classifier is simply the identity function, which returns the element as-is. The downstream operation counts the number of equal elements, using `counting()`.

```
Stream.of("apple", "orange", "banana", "apple")
      .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()))
      .entrySet()
      .forEach(System.out::println);
```

The downstream operation is itself a collector (`Collectors.counting()`) that operates on elements of type String and produces a result of type `Long`. The result of the `collect` method call is a `Map<String, Long>`.

This would produce the following output:

> banana=1
> orange=1
> apple=2

Section 57.4: Infinite Streams

It is possible to generate a `Stream` that does not end. Calling a terminal method on an infinite `Stream` causes the `Stream` to enter an infinite loop. The `limit` method of a `Stream` can be used to limit the number of terms of the `Stream` that Java processes.

This example generates a `Stream` of all natural numbers, starting with the number 1. Each successive term of the `Stream` is one higher than the previous. By calling the limit method of this `Stream`, only the first five terms of the `Stream` are considered and printed.

```
// Generate infinite stream - 1, 2, 3, 4, 5, 6, 7, ...
IntStream naturalNumbers = IntStream.iterate(1, x -> x + 1);

// Print out only the first 5 terms
naturalNumbers.limit(5).forEach(System.out::println);
```

Output:

```
1
2
3
4
5
```

Another way of generating an infinite stream is using the Stream.generate method. This method takes a lambda of type Supplier.

```
// Generate an infinite stream of random numbers
Stream<Double> infiniteRandomNumbers = Stream.generate(Math::random);

// Print out only the first 10 random numbers
infiniteRandomNumbers.limit(10).forEach(System.out::println);
```

Section 57.5: Collect Elements of a Stream into a Collection

Collect with `toList()` and `toSet()`

Elements from a `Stream` can be easily collected into a container by using the `Stream.collect` operation:

```
System.out.println(Arrays
    .asList("apple", "banana", "pear", "kiwi", "orange")
    .stream()
    .filter(s -> s.contains("a"))
    .collect(Collectors.toList())
);
// prints: [apple, banana, pear, orange]
```

Other collection instances, such as a `Set`, can be made by using other `Collectors` built-in methods. For example, `Collectors.toSet()` collects the elements of a `Stream` into a `Set`.

Explicit control over the implementation of `List` or `Set`

According to documentation of `Collectors#toList()` and `Collectors#toSet()`, there are no guarantees on the type, mutability, serializability, or thread-safety of the `List` or `Set` returned.

For explicit control over the implementation to be returned, `Collectors#toCollection(Supplier)` can be used instead, where the given supplier returns a new and empty collection.

```
// syntax with method reference
```

```
System.out.println(strings
        .stream()
        .filter(s -> s != null && s.length() <= 3)
        .collect(Collectors.toCollection(ArrayList::new))
);

// syntax with lambda
System.out.println(strings
        .stream()
        .filter(s -> s != null && s.length() <= 3)
        .collect(Collectors.toCollection(() -> new LinkedHashSet<>()))
);
```

Collecting Elements using toMap

Collector accumulates elements into a Map, Where key is the Student Id and Value is Student Value.

```
  List<Student> students = new ArrayList<Student>();
    students.add(new Student(1,"test1"));
    students.add(new Student(2,"test2"));
    students.add(new Student(3,"test3"));

    Map<Integer, String> IdToName = students.stream()
        .collect(Collectors.toMap(Student::getId, Student::getName));
    System.out.println(IdToName);
```

Output :

```
{1=test1, 2=test2, 3=test3}
```

The Collectors.toMap has another implementation `Collector<T, ?, Map<K,U>> toMap(Function<? super T, ? extends K> keyMapper, Function<? super T, ? extends U> valueMapper, BinaryOperator<U> mergeFunction)`.The mergeFunction is mostly used to select either new value or retain old value if the key is repeated when adding a new member in the Map from a list.

The mergeFunction often looks like: `(s1, s2) -> s1` to retain value corresponding to the repeated key, or `(s1, s2) -> s2` to put new value for the repeated key.

Collecting Elements to Map of Collections

Example: from ArrayList to Map<String, List<>>

Often it requires to make a map of list out of a primary list. Example: From a student of list, we need to make a map of list of subjects for each student.

```
    List<Student> list = new ArrayList<>();
    list.add(new Student("Davis", SUBJECT.MATH, 35.0));
    list.add(new Student("Davis", SUBJECT.SCIENCE, 12.9));
    list.add(new Student("Davis", SUBJECT.GEOGRAPHY, 37.0));

    list.add(new Student("Sascha", SUBJECT.ENGLISH, 85.0));
    list.add(new Student("Sascha", SUBJECT.MATH, 80.0));
    list.add(new Student("Sascha", SUBJECT.SCIENCE, 12.0));
    list.add(new Student("Sascha", SUBJECT.LITERATURE, 50.0));

    list.add(new Student("Robert", SUBJECT.LITERATURE, 12.0));
```

```
    Map<String, List<SUBJECT>> map = new HashMap<>();
    list.stream().forEach(s -> {
                map.computeIfAbsent(s.getName(), x -> new ArrayList<>()).add(s.getSubject());
            });
    System.out.println(map);
```

Output:

```
{ Robert=[LITERATURE],
Sascha=[ENGLISH, MATH, SCIENCE, LITERATURE],
Davis=[MATH, SCIENCE, GEOGRAPHY] }
```

Example: from ArrayList to Map<String, Map<>>

```
    List<Student> list = new ArrayList<>();
    list.add(new Student("Davis", SUBJECT.MATH, 1, 35.0));
    list.add(new Student("Davis", SUBJECT.SCIENCE, 2, 12.9));
    list.add(new Student("Davis", SUBJECT.MATH, 3, 37.0));
    list.add(new Student("Davis", SUBJECT.SCIENCE, 4, 37.0));

    list.add(new Student("Sascha", SUBJECT.ENGLISH, 5, 85.0));
    list.add(new Student("Sascha", SUBJECT.MATH, 1, 80.0));
    list.add(new Student("Sascha", SUBJECT.ENGLISH, 6, 12.0));
    list.add(new Student("Sascha", SUBJECT.MATH, 3, 50.0));

    list.add(new Student("Robert", SUBJECT.ENGLISH, 5, 12.0));

    Map<String, Map<SUBJECT, List<Double>>> map = new HashMap<>();

    list.stream().forEach(student -> {
        map.computeIfAbsent(student.getName(), s -> new HashMap<>())
                .computeIfAbsent(student.getSubject(), s -> new ArrayList<>())
                .add(student.getMarks());
    });

    System.out.println(map);
```

Output:

```
{ Robert={ENGLISH=[12.0]},
Sascha={MATH=[80.0, 50.0], ENGLISH=[85.0, 12.0]},
Davis={MATH=[35.0, 37.0], SCIENCE=[12.9, 37.0]} }
```

Cheat-Sheet

Goal	Code
Collect to a `List`	`Collectors.toList()`
Collect to an `ArrayList` with pre-allocated size	`Collectors.toCollection(() -> new ArrayList<>(size))`
Collect to a `Set`	`Collectors.toSet()`
Collect to a `Set` with better iteration performance	`Collectors.toCollection(() -> new LinkedHashSet<>())`
Collect to a case-insensitive `Set<String>`	`Collectors.toCollection(() -> new TreeSet<>(String.CASE_INSENSITIVE_ORDER))`
Collect to an `EnumSet<AnEnum>` (best performance for enums)	`Collectors.toCollection(() -> EnumSet.noneOf(AnEnum.class))`

Collect to a `Map<K,V>` with unique keys	`Collectors.toMap(keyFunc,valFunc)`
Map MyObject.getter() to unique MyObject	`Collectors.toMap(MyObject::getter, Function.identity())`
Map MyObject.getter() to multiple MyObjects	`Collectors.groupingBy(MyObject::getter)`

Section 57.6: Using Streams to Implement Mathematical Functions

`Streams`, and especially `IntStreams`, are an elegant way of implementing summation terms (∑). The ranges of the `Stream` can be used as the bounds of the summation.

E.g., Madhava's approximation of Pi is given by the formula (Source: wikipedia):

$$\pi = \sqrt{12}\sum_{k=0}^{\infty}\frac{(-3)^{-k}}{2k+1} = \sqrt{12}\sum_{k=0}^{\infty}\frac{(-\frac{1}{3})^{k}}{2k+1} = \sqrt{12}\left(\frac{1}{1\cdot 3^0} - \frac{1}{3\cdot 3^1} + \frac{1}{5\cdot 3^2} - \frac{1}{7\cdot 3^3} + \cdots\right)$$

This can be calculated with an arbitrary precision. E.g., for 101 terms:

```
double pi = Math.sqrt(12) *
            IntStream.rangeClosed(0, 100)
                     .mapToDouble(k -> Math.pow(-3, -1 * k) / (2 * k + 1))
                     .sum();
```

Note: With `double`'s precision, selecting an upper bound of 29 is sufficient to get a result that's indistinguishable from `Math.Pi`.

Section 57.7: Flatten Streams with flatMap()

A `Stream` of items that are in turn streamable can be flattened into a single continuous `Stream`:

Array of List of Items can be converted into a single List.

```
List<String> list1 =  Arrays.asList("one", "two");
      List<String> list2 =  Arrays.asList("three","four","five");
      List<String> list3 =  Arrays.asList("six");
          List<String> finalList = Stream.of(list1, list2,
list3).flatMap(Collection::stream).collect(Collectors.toList());
System.out.println(finalList);

// [one, two, three, four, five, six]
```

Map containing List of Items as values can be Flattened to a Combined List

```
Map<String, List<Integer>> map = new LinkedHashMap<>();
map.put("a", Arrays.asList(1, 2, 3));
map.put("b", Arrays.asList(4, 5, 6));

List<Integer> allValues = map.values() // Collection<List<Integer>>
        .stream()                      // Stream<List<Integer>>
        .flatMap(List::stream)         // Stream<Integer>
        .collect(Collectors.toList());

System.out.println(allValues);
// [1, 2, 3, 4, 5, 6]
```

`List` of `Map` can be flattened into a single continuous `Stream`

```
List<Map<String, String>> list = new ArrayList<>();
Map<String,String> map1 = new HashMap();
map1.put("1", "one");
map1.put("2", "two");

Map<String,String> map2 = new HashMap();
map2.put("3", "three");
map2.put("4", "four");
list.add(map1);
list.add(map2);


Set<String> output= list.stream()  //  Stream<Map<String, String>>
    .map(Map::values)              // Stream<List<String>>
    .flatMap(Collection::stream)   // Stream<String>
    .collect(Collectors.toSet());  //Set<String>
// [one, two, three,four]
```

Section 57.8: Parallel Stream

Note: Before deciding which `Stream` to use please have a look at ParallelStream vs Sequential Stream behavior.

When you want to perform `Stream` operations concurrently, you could use either of these ways.

```
List<String> data = Arrays.asList("One", "Two", "Three", "Four", "Five");
Stream<String> aParallelStream = data.stream().parallel();
```

Or:

```
Stream<String> aParallelStream = data.parallelStream();
```

To execute the operations defined for the parallel stream, call a terminal operator:

```
aParallelStream.forEach(System.out::println);
```

(A possible) output from the parallel `Stream`:

> Three
> Four
> One
> Two
> Five

The order might change as all the elements are processed in parallel (Which *may* make it faster). Use `parallelStream` when ordering does not matter.

Performance impact

In case networking is involved, parallel `Streams` may degrade the overall performance of an application because all parallel `Streams` use a common fork-join thread pool for the network.

On the other hand, parallel `Streams` may significantly improve performance in many other cases, depending of the number of available cores in the running CPU at the moment.

Section 57.9: Creating a Stream

All java `Collection<E>`s have `stream()` and `parallelStream()` methods from which a `Stream<E>` can be constructed:

```
Collection<String> stringList = new ArrayList<>();
Stream<String> stringStream = stringList.parallelStream();
```

A `Stream<E>` can be created from an array using one of two methods:

```
String[] values = { "aaa", "bbbb", "ddd", "cccc" };
Stream<String> stringStream = Arrays.stream(values);
Stream<String> stringStreamAlternative = Stream.of(values);
```

The difference between `Arrays.stream()` and `Stream.of()` is that `Stream.of()` has a varargs parameter, so it can be used like:

```
Stream<Integer> integerStream = Stream.of(1, 2, 3);
```

There are also primitive `Streams` that you can use. For example:

```
IntStream intStream = IntStream.of(1, 2, 3);
DoubleStream doubleStream = DoubleStream.of(1.0, 2.0, 3.0);
```

These primitive streams can also be constructed using the `Arrays.stream()` method:

```
IntStream intStream = Arrays.stream(new int[]{ 1, 2, 3 });
```

It is possible to create a `Stream` from an array with a specified range.

```
int[] values= new int[]{1, 2, 3, 4, 5};
IntStream intStram = Arrays.stream(values, 1, 3);
```

Note that any primitive stream can be converted to boxed type stream using the `boxed` method :

```
Stream<Integer> integerStream = intStream.boxed();
```

This can be useful in some case if you want to collect the data since primitive stream does not have any `collect` method that takes a `Collector` as argument.

Reusing intermediate operations of a stream chain

Stream is closed when ever terminal operation is called. Reusing the stream of intermediate operations, when only terminal operation is only varying. we could create a stream supplier to construct a new stream with all intermediate operations already set up.

```
Supplier<Stream<String>> streamSupplier = () -> Stream.of("apple", "banana","orange", "grapes",
"melon","blueberry","blackberry")
.map(String::toUpperCase).sorted();

  streamSupplier.get().filter(s ->   s.startsWith("A")).forEach(System.out::println);

// APPLE

  streamSupplier.get().filter(s -> s.startsWith("B")).forEach(System.out::println);
```

```
// BANANA
// BLACKBERRY
// BLUEBERRY
```

`int[]` arrays can be converted to `List<Integer>` using streams

```
int[] ints = {1,2,3};
List<Integer> list = IntStream.of(ints).boxed().collect(Collectors.toList());
```

Section 57.10: Finding Statistics about Numerical Streams

Java 8 provides classes called `IntSummaryStatistics`, `DoubleSummaryStatistics` and `LongSummaryStatistics` which give a state object for collecting statistics such as `count`, `min`, `max`, `sum`, and `average`.

Version ≥ Java SE 8

```
List<Integer> naturalNumbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
IntSummaryStatistics stats = naturalNumbers.stream()
                                           .mapToInt((x) -> x)
                                           .summaryStatistics();
System.out.println(stats);
```

Which will result in:

Version ≥ Java SE 8

```
IntSummaryStatistics{count=10, sum=55, min=1, max=10, average=5.500000}
```

Section 57.11: Converting an iterator to a stream

Use `Spliterators.spliterator()` or `Spliterators.spliteratorUnknownSize()` to convert an iterator to a stream:

```
Iterator<String> iterator = Arrays.asList("A", "B", "C").iterator();
Spliterator<String> spliterator = Spliterators.spliteratorUnknownSize(iterator, 0);
Stream<String> stream = StreamSupport.stream(spliterator, false);
```

Section 57.12: Using IntStream to iterate over indexes

`Streams` of elements usually do not allow access to the index value of the current item. To iterate over an array or `ArrayList` while having access to indexes, use `IntStream.range(start, endExclusive)`.

```
String[] names = { "Jon", "Darin", "Bauke", "Hans", "Marc" };

IntStream.range(0, names.length)
    .mapToObj(i -> String.format("#%d %s", i + 1, names[i]))
    .forEach(System.out::println);
```

The `range(start, endExclusive)` method returns another `IntStream` and the `mapToObj(mapper)` returns a stream of `String`.

Output:

> #1 Jon
> #2 Darin
> #3 Bauke
> #4 Hans

> #5 Marc

This is very similar to using a normal `for` loop with a counter, but with the benefit of pipelining and parallelization:

```
for (int i = 0; i < names.length; i++) {
    String newName = String.format("#%d %s", i + 1, names[i]);
    System.out.println(newName);
}
```

Section 57.13: Concatenate Streams

Variable declaration for examples:

```
Collection<String> abc = Arrays.asList("a", "b", "c");
Collection<String> digits = Arrays.asList("1", "2", "3");
Collection<String> greekAbc = Arrays.asList("alpha", "beta", "gamma");
```

Example 1 - Concatenate two `Streams`

```
final Stream<String> concat1 = Stream.concat(abc.stream(), digits.stream());

concat1.forEach(System.out::print);
// prints: abc123
```

Example 2 - Concatenate more than two `Streams`

```
final Stream<String> concat2 = Stream.concat(
    Stream.concat(abc.stream(), digits.stream()),
    greekAbc.stream());

System.out.println(concat2.collect(Collectors.joining(", ")));
// prints: a, b, c, 1, 2, 3, alpha, beta, gamma
```

Alternatively to simplify the nested `concat()` syntax the `Streams` can also be concatenated with `flatMap()`:

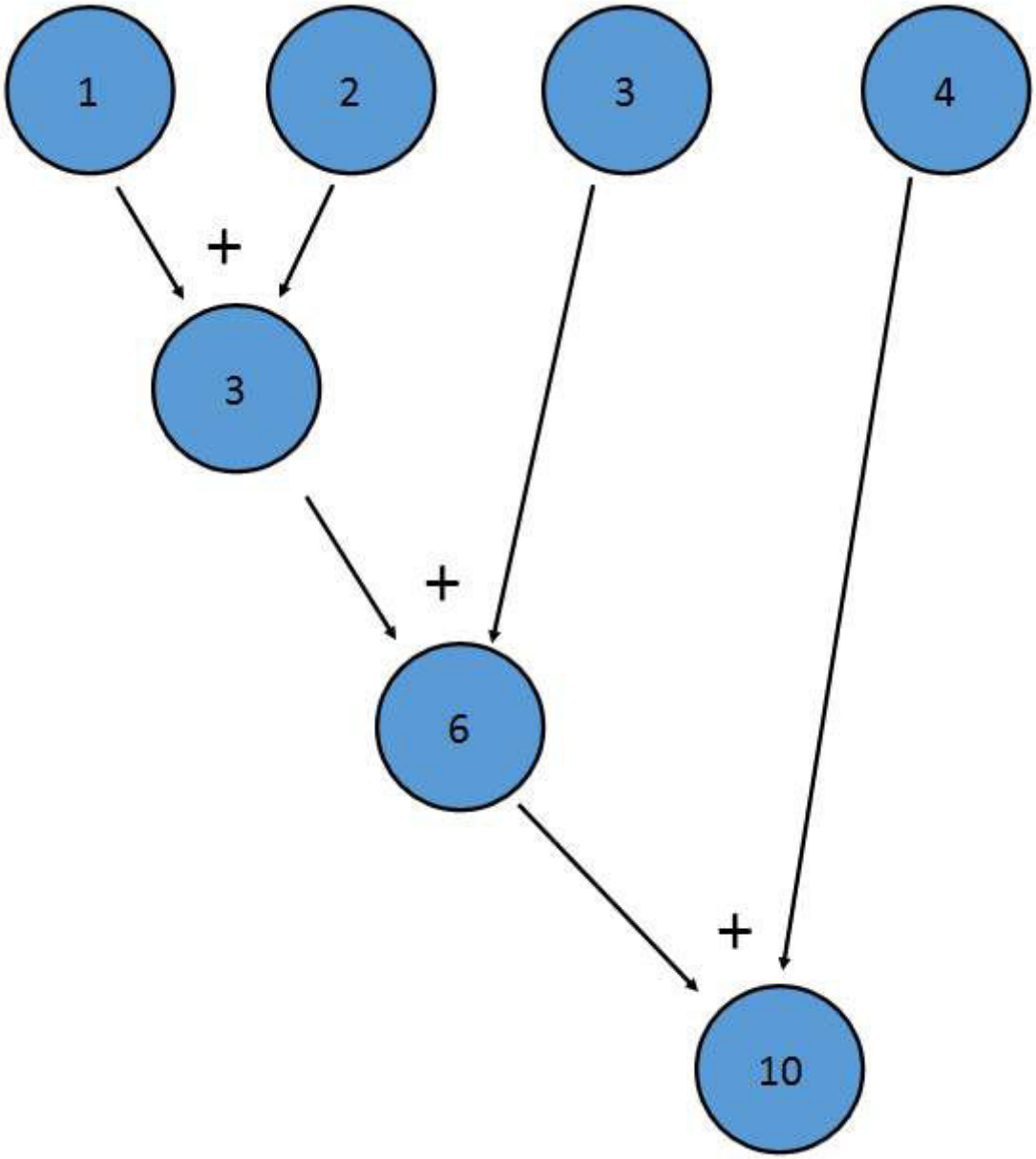
```
final Stream<String> concat3 = Stream.of(
    abc.stream(), digits.stream(), greekAbc.stream())
    .flatMap(s -> s);
    // or `.flatMap(Function.identity());` (java.util.function.Function)

System.out.println(concat3.collect(Collectors.joining(", ")));
// prints: a, b, c, 1, 2, 3, alpha, beta, gamma
```

Be careful when constructing `Streams` from repeated concatenation, because accessing an element of a deeply concatenated `Stream` can result in deep call chains or even a `StackOverflowException`.

Section 57.14: Reduction with Streams

Reduction is the process of applying a binary operator to every element of a stream to result in one value.

The `sum()` method of an `IntStream` is an example of a reduction; it applies addition to every term of the Stream, resulting in one final value:



This is equivalent to `(((1+2)+3)+4)`

The `reduce` method of a Stream allows one to create a custom reduction. It is possible to use the `reduce` method to implement the `sum()` method:

```
IntStream istr;

//Initialize istr

OptionalInt istr.reduce((a,b)->a+b);
```

The `Optional` version is returned so that empty Streams can be handled appropriately.

Another example of reduction is combining a `Stream<LinkedList<T>>` into a single `LinkedList<T>`:

```
Stream<LinkedList<T>> listStream;

//Create a Stream<LinkedList<T>>
```

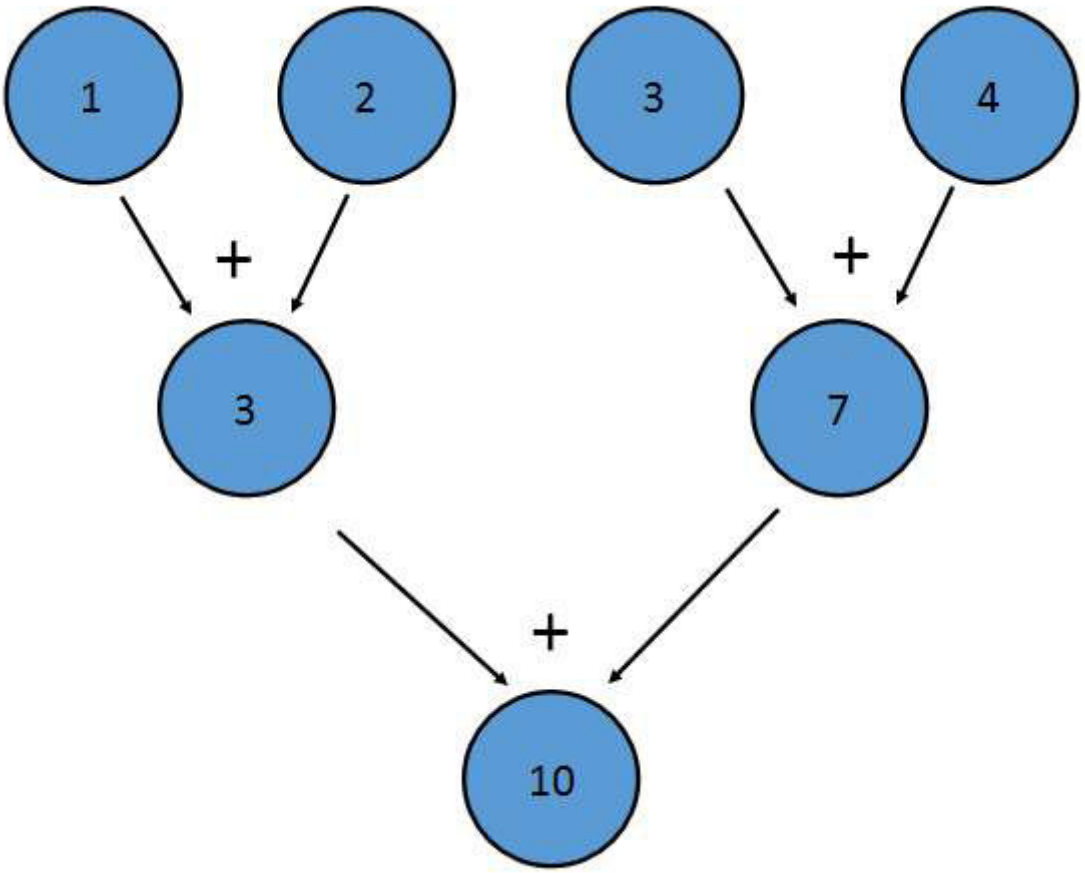
```
Optional<LinkedList<T>> bigList = listStream.reduce((LinkedList<T> list1, LinkedList<T> list2)->{
    LinkedList<T> retList = new LinkedList<T>();
    retList.addAll(list1);
    retList.addAll(list2);
    return retList;
});
```

You can also provide an *identity element*. For example, the identity element for addition is 0, as `x+0==x`. For multiplication, the identity element is 1, as `x*1==x`. In the case above, the identity element is an empty `LinkedList<T>`, because if you add an empty list to another list, the list that you are "adding" to doesn't change:

```
Stream<LinkedList<T>> listStream;

//Create a Stream<LinkedList<T>>

LinkedList<T> bigList = listStream.reduce(new LinkedList<T>(), (LinkedList<T> list1, LinkedList<T>
list2)->{
    LinkedList<T> retList = new LinkedList<T>();
    retList.addAll(list1);
    retList.addAll(list2);
    return retList;
});
```

Note that when an identity element is provided, the return value is not wrapped in an `Optional`—if called on an empty stream, `reduce()` will return the identity element.

The binary operator must also be *associative*, meaning that `(a+b)+c==a+(b+c)`. This is because the elements may be reduced in any order. For example, the above addition reduction could be performed like this:



This reduction is equivalent to writing `((1+2)+(3+4))`. The property of associativity also allows Java to reduce the `Stream` in parallel—a portion of the Stream can be reduced by each processor, with a reduction combining the result of each processor at the end.

Section 57.15: Using Streams of Map.Entry to Preserve Initial Values after Mapping

When you have a `Stream` you need to map but want to preserve the initial values as well, you can map the `Stream` to a `Map.Entry<K,V>` using a utility method like the following:

```
public static <K, V> Function<K, Map.Entry<K, V>> entryMapper(Function<K, V> mapper){
    return (k)->new AbstractMap.SimpleEntry<>(k, mapper.apply(k));
}
```

Then you can use your converter to process `Streams` having access to both the original and mapped values:

```
Set<K> mySet;
Function<K, V> transformer = SomeClass::transformerMethod;
Stream<Map.Entry<K, V>> entryStream = mySet.stream()
    .map(entryMapper(transformer));
```

You can then continue to process that `Stream` as normal. This avoids the overhead of creating an intermediate collection.

Section 57.16: IntStream to String

Java does not have a *Char Stream*, so when working with `String`s and constructing a `Stream` of `Character`s, an option is to get a `IntStream` of code points using `String.codePoints()` method. So `IntStream` can be obtained as below:

```
public IntStream stringToIntStream(String in) {
  return in.codePoints();
}
```

It is a bit more involved to do the conversion other way around i.e. IntStreamToString. That can be done as follows:

```
public String intStreamToString(IntStream intStream) {
  return intStream.collect(StringBuilder::new, StringBuilder::appendCodePoint,
StringBuilder::append).toString();
}
```

Section 57.17: Finding the First Element that Matches a Predicate

It is possible to find the first element of a `Stream` that matches a condition.

For this example, we will find the first `Integer` whose square is over `50000`.

```
IntStream.iterate(1, i -> i + 1) // Generate an infinite stream 1,2,3,4...
    .filter(i -> (i*i) > 50000) // Filter to find elements where the square is >50000
    .findFirst(); // Find the first filtered element
```

This expression will return an `OptionalInt` with the result.

Note that with an infinite `Stream`, Java will keep checking each element until it finds a result. With a finite `Stream`, if Java runs out of elements but still can't find a result, it returns an empty `OptionalInt`.

Section 57.18: Using Streams and Method References to Write Self-Documenting Processes

Method references make excellent self-documenting code, and using method references with `Streams` makes complicated processes simple to read and understand. Consider the following code:

```
public interface Ordered {
    default int getOrder(){
        return 0;
    }
}

public interface Valued<V extends Ordered> {
    boolean hasPropertyTwo();
    V getValue();
}

public interface Thing<V extends Ordered> {
    boolean hasPropertyOne();
    Valued<V> getValuedProperty();
}

public <V extends Ordered> List<V> myMethod(List<Thing<V>> things) {
    List<V> results = new ArrayList<V>();
    for (Thing<V> thing : things) {
        if (thing.hasPropertyOne()) {
            Valued<V> valued = thing.getValuedProperty();
            if (valued != null && valued.hasPropertyTwo()){
                V value = valued.getValue();
                if (value != null){
                    results.add(value);
                }
            }
        }
    }
    results.sort((a, b)->{
        return Integer.compare(a.getOrder(), b.getOrder());
    });
    return results;
}
```

This last method rewritten using `Streams` and method references is much more legible and each step of the process is quickly and easily understood - it's not just shorter, it also shows at a glance which interfaces and classes are responsible for the code in each step:

```
public <V extends Ordered> List<V> myMethod(List<Thing<V>> things) {
    return things.stream()
        .filter(Thing::hasPropertyOne)
        .map(Thing::getValuedProperty)
        .filter(Objects::nonNull)
        .filter(Valued::hasPropertyTwo)
        .map(Valued::getValue)
        .filter(Objects::nonNull)
        .sorted(Comparator.comparing(Ordered::getOrder))
        .collect(Collectors.toList());
```

```
}
```

Section 57.19: Converting a Stream of Optional to a Stream of Values

You may need to convert a `Stream` emitting `Optional` to a `Stream` of values, emitting only values from existing `Optional`. (ie: without **`null`** value and not dealing with `Optional.empty()`).

```
Optional<String> op1 = Optional.empty();
Optional<String> op2 = Optional.of("Hello World");

List<String> result = Stream.of(op1, op2)
                            .filter(Optional::isPresent)
                            .map(Optional::get)
                            .collect(Collectors.toList());

System.out.println(result); //[Hello World]
```

Section 57.20: Get a Slice of a Stream

Example: Get a `Stream` of 30 elements, containing 21st to 50th (inclusive) element of a collection.

```
final long n = 20L; // the number of elements to skip
final long maxSize = 30L; // the number of elements the stream should be limited to
final Stream<T> slice = collection.stream().skip(n).limit(maxSize);
```

Notes:

- `IllegalArgumentException` is thrown if `n` is negative or `maxSize` is negative
- both `skip(long)` and `limit(long)` are intermediate operations
- if a stream contains fewer than `n` elements then `skip(n)` returns an empty stream
- both `skip(long)` and `limit(long)` are cheap operations on sequential stream pipelines, but can be quite expensive on ordered parallel pipelines

Section 57.21: Create a Map based on a Stream

Simple case without duplicate keys

```
Stream<String> characters = Stream.of("A", "B", "C");

Map<Integer, String> map = characters
            .collect(Collectors.toMap(element -> element.hashCode(), element -> element));
// map = {65=A, 66=B, 67=C}
```

To make things more declarative, we can use static method in `Function` interface - `Function.identity()`. We can replace this lambda `element -> element` with `Function.identity()`.

Case where there might be duplicate keys

The javadoc for `Collectors.toMap` states:

> If the mapped keys contains duplicates (according to `Object.equals(Object)`), an `IllegalStateException` is thrown when the collection operation is performed. If the mapped keys may have duplicates, use `toMap(Function, Function, BinaryOperator)` instead.

```
Stream<String> characters = Stream.of("A", "B", "B", "C");

Map<Integer, String> map = characters
            .collect(Collectors.toMap(
                element -> element.hashCode(),
                element -> element,
                (existingVal, newVal) -> (existingVal + newVal)));

// map = {65=A, 66=BB, 67=C}
```

The `BinaryOperator` passed to `Collectors.toMap(...)` generates the value to be stored in the case of a collision. It can:

- return the old value, so that the first value in the stream takes precedence,
- return the new value, so that the last value in the stream takes precedence, or
- combine the old and new values

Grouping by value

You can use `Collectors.groupingBy` when you need to perform the equivalent of a database cascaded "group by" operation. To illustrate, the following creates a map in which people's names are mapped to surnames:

```
List<Person> people = Arrays.asList(
    new Person("Sam", "Rossi"),
    new Person("Sam", "Verdi"),
    new Person("John", "Bianchi"),
    new Person("John", "Rossi"),
    new Person("John", "Verdi")
);

Map<String, List<String>> map = people.stream()
        .collect(
                // function mapping input elements to keys
                Collectors.groupingBy(Person::getName,
                // function mapping input elements to values,
                // how to store values
                Collectors.mapping(Person::getSurname, Collectors.toList()))
        );

// map = {John=[Bianchi, Rossi, Verdi], Sam=[Rossi, Verdi]}
```

Live on Ideone

Section 57.22: Joining a stream to a single String

A use case that comes across frequently, is creating a `String` from a stream, where the stream-items are separated by a certain character. The `Collectors.joining()` method can be used for this, like in the following example:

```
Stream<String> fruitStream = Stream.of("apple", "banana", "pear", "kiwi", "orange");

String result = fruitStream.filter(s -> s.contains("a"))
           .map(String::toUpperCase)
           .sorted()
           .collect(Collectors.joining(", "));

System.out.println(result);
```

Output:

> APPLE, BANANA, ORANGE, PEAR

The `Collectors.joining()` method can also cater for pre- and postfixes:

```
String result = fruitStream.filter(s -> s.contains("e"))
           .map(String::toUpperCase)
           .sorted()
           .collect(Collectors.joining(", ", "Fruits: ", "."));

System.out.println(result);
```

Output:

> Fruits: APPLE, ORANGE, PEAR.

Live on Ideone

Section 57.23: Sort Using Stream

```
List<String> data = new ArrayList<>();
data.add("Sydney");
data.add("London");
data.add("New York");
data.add("Amsterdam");
data.add("Mumbai");
data.add("California");

System.out.println(data);

List<String> sortedData = data.stream().sorted().collect(Collectors.toList());

System.out.println(sortedData);
```

Output:

```
[Sydney, London, New York, Amsterdam, Mumbai, California]
[Amsterdam, California, London, Mumbai, New York, Sydney]
```

It's also possible to use different comparison mechanism as there is a overloaded `sorted` version which takes a comparator as its argument.

Also, you can use a lambda expression for sorting:

```
List<String> sortedData2 = data.stream().sorted((s1,s2) ->
s2.compareTo(s1)).collect(Collectors.toList());
```

This would output

```
[Sydney, New York, Mumbai, London, California, Amsterdam]
```

You can use `Comparator.reverseOrder()` to have a comparator that imposes the `reverse` of the natural ordering.

```
List<String> reverseSortedData =
```

```
data.stream().sorted(Comparator.reverseOrder()).collect(Collectors.toList());
```

Section 57.24: Streams of Primitives

Java provides specialized `Streams` for three types of primitives `IntStream` (for **`int`**s), `LongStream` (for **`long`**s) and `DoubleStream` (for **`double`**s). Besides being optimized implementations for their respective primitives, they also provide several specific terminal methods, typically for mathematical operations. E.g.:

```
IntStream is = IntStream.of(10, 20, 30);
double average = is.average().getAsDouble(); // average is 20.0
```

Section 57.25: Stream operations categories

Stream operations fall into two main categories, intermediate and terminal operations, and two sub-categories, stateless and stateful.

Intermediate Operations:

An intermediate operation is always *lazy*, such as a simple `Stream.map`. It is not invoked until the stream is actually consumed. This can be verified easily:

```
Arrays.asList(1, 2 ,3).stream().map(i -> {
    throw new RuntimeException("not gonna happen");
    return i;
});
```

Intermediate operations are the common building blocks of a stream, chained after the source and are usually followed by a terminal operation triggering the stream chain.

Terminal Operations

Terminal operations are what triggers the consumption of a stream. Some of the more common are `Stream.`**`forEach`** or `Stream.collect`. They are usually placed after a chain of intermediate operations and are almost always *eager*.

Stateless Operations

Statelessness means that each item is processed without the context of other items. Stateless operations allow for memory-efficient processing of streams. Operations like `Stream.map` and `Stream.filter` that do not require information on other items of the stream are considered to be stateless.

Stateful operations

Statefulness means the operation on each item depends on (some) other items of the stream. This requires a state to be preserved. Statefulness operations may break with long, or infinite, streams. Operations like `Stream.sorted` require the entirety of the stream to be processed before any item is emitted which will break in a long enough stream of items. This can be demonstrated by a long stream (**run at your own risk**):

```
// works - stateless stream
long BIG_ENOUGH_NUMBER = 999999999;
```

```
IntStream.iterate(0, i -> i + 1).limit(BIG_ENOUGH_NUMBER).forEach(System.out::println);
```

This will cause an out-of-memory due to statefulness of `Stream.sorted`:

```
// Out of memory - stateful stream
IntStream.iterate(0, i -> i + 1).limit(BIG_ENOUGH_NUMBER).sorted().forEach(System.out::println);
```

Section 57.26: Collect Results of a Stream into an Array

Analog to get a collection for a `Stream` by `collect()` an array can be obtained by the `Stream.toArray()` method:

```
List<String> fruits = Arrays.asList("apple", "banana", "pear", "kiwi", "orange");

String[] filteredFruits = fruits.stream()
    .filter(s -> s.contains("a"))
    .toArray(String[]::new);

// prints: [apple, banana, pear, orange]
System.out.println(Arrays.toString(filteredFruits));
```

`String[]::new` is a special kind of method reference: a constructor reference.

Section 57.27: Generating random Strings using Streams

It is sometimes useful to create random `Strings`, maybe as Session-ID for a web-service or an initial password after registration for an application. This can be easily achieved using `Streams`.

First we need to initialize a random number generator. To enhance security for the generated `Strings`, it is a good idea to use `SecureRandom`.

Note: Creating a `SecureRandom` is quite expensive, so it is best practice to only do this once and call one of its `setSeed()` methods from time to time to reseed it.

```
private static final SecureRandom rng = new SecureRandom(SecureRandom.generateSeed(20));
//20 Bytes as a seed is rather arbitrary, it is the number used in the JavaDoc example
```

When creating random `Strings`, we usually want them to use only certain characters (e.g. only letters and digits). Therefore we can create a method returning a `boolean` which can later be used to filter the `Stream`.

```
//returns true for all chars in 0-9, a-z and A-Z
boolean useThisCharacter(char c){
    //check for range to avoid using all unicode Letter (e.g. some chinese symbols)
    return c >= '0' && c <= 'z' && Character.isLetterOrDigit(c);
}
```

Next we can utilize the RNG to generate a random String of specific length containing the charset which pass our `useThisCharacter` check.

```
public String generateRandomString(long length){
    //Since there is no native CharStream, we use an IntStream instead
    //and convert it to a Stream<Character> using mapToObj.
    //We need to specify the boundaries for the int values to ensure they can safely be cast to char
    Stream<Character> randomCharStream = rng.ints(Character.MIN_CODE_POINT,
Character.MAX_CODE_POINT).mapToObj(i -> (char)i).filter(c -> this::useThisCharacter).limit(length);

    //now we can use this Stream to build a String utilizing the collect method.
```

```
    String randomString = randomCharStream.collect(StringBuilder::new, StringBuilder::append,
StringBuilder::append).toString();
    return randomString;
}
```

Chapter 58: InputStreams and OutputStreams

Section 58.1: Closing Streams

Most streams must be closed when you are done with them, otherwise you could introduce a memory leak or leave a file open. It is important that streams are closed even if an exception is thrown.

Version ≥ Java SE 7

```
try(FileWriter fw = new FileWriter("outfilename");
    BufferedWriter bw = new BufferedWriter(fw);
    PrintWriter out = new PrintWriter(bw))
{
    out.println("the text");
    //more code
    out.println("more text");
    //more code
} catch (IOException e) {
    //handle this however you
}
```

Remember: try-with-resources guarantees, that the resources have been closed when the block is exited, whether that happens with the usual control flow or because of an exception.

Version ≤ Java SE 6

Sometimes, try-with-resources is not an option, or maybe you're supporting older version of Java 6 or earlier. In this case, proper handling is to use a **`finally`** block:

```
FileWriter fw = null;
BufferedWriter bw = null;
PrintWriter out = null;
try {
    fw = new FileWriter("myfile.txt");
    bw = new BufferedWriter(fw);
    out = new PrintWriter(bw);
    out.println("the text");
    out.close();
} catch (IOException e) {
    //handle this however you want
}
finally {
    try {
        if(out != null)
            out.close();
    } catch (IOException e) {
        //typically not much you can do here...
    }
}
```

Note that closing a wrapper stream will also close its underlying stream. This means you cannot wrap a stream, close the wrapper and then continue using the original stream.

Section 58.2: Reading InputStream into a String

Sometimes you may wish to read byte-input into a String. To do this you will need to find something that converts

between **byte** and the "native Java" UTF-16 Codepoints used as **char**. That is done with a `InputStreamReader`.

To speed the process up a bit, it's "usual" to allocate a buffer, so that we don't have too much overhead when reading from Input.

Version ≥ Java SE 7

```
public String inputStreamToString(InputStream inputStream) throws Exception {
     StringWriter writer = new StringWriter();

     char[] buffer = new char[1024];
     try (Reader reader = new BufferedReader(new InputStreamReader(inputStream, "UTF-8"))) {
          int n;
          while ((n = reader.read(buffer)) != -1) {
               // all this code does is redirect the output of `reader` to `writer` in
               // 1024 byte chunks
               writer.write(buffer, 0, n);
          }
     }
     return writer.toString();
}
```

Transforming this example to Java SE 6 (and lower)-compatible code is left out as an exercise for the reader.

Section 58.3: Wrapping Input/Output Streams

`OutputStream` and `InputStream` have many different classes, each of them with a unique functionality. By wrapping a stream around another, you gain the functionality of both streams.

You can wrap a stream any number of times, just take note of the ordering.

Useful combinations

Writing characters to a file while using a buffer

```
File myFile = new File("targetFile.txt");
PrintWriter writer = new PrintWriter(new BufferedOutputStream(new FileOutputStream(myFile)));
```

Compressing and encrypting data before writing to a file while using a buffer

```
Cipher cipher = ... // Initialize cipher
File myFile = new File("targetFile.enc");
BufferedOutputStream outputStream = new BufferedOutputStream(new DeflaterOutputStream(new
CipherOutputStream(new FileOutputStream(myFile), cipher)));
```

List of Input/Output Stream wrappers

Wrapper	Description
BufferedOutputStream/ BufferedInputStream	While `OutputStream` writes data one byte at a time, `BufferedOutputStream` writes data in chunks. This reduces the number of system calls, thus improving performance.
DeflaterOutputStream/ DeflaterInputStream	Performs data compression.
InflaterOutputStream/ InflaterInputStream	Performs data decompression.
CipherOutputStream/ CipherInputStream	Encrypts/Decrypts data.
DigestOutputStream/ DigestInputStream	Generates Message Digest to verify data integrity.
CheckedOutputStream/ CheckedInputStream	Generates a CheckSum. CheckSum is a more trivial version of Message Digest.

DataOutputStream/ DataInputStream	Allows writing of primitive data types and Strings. Meant for writing bytes. Platform independent.
PrintStream	Allows writing of primitive data types and Strings. Meant for writing bytes. Platform dependent.
OutputStreamWriter	Converts a OutputStream into a Writer. An OutputStream deals with bytes while Writers deals with characters
PrintWriter	Automatically calls OutputStreamWriter. Allows writing of primitive data types and Strings. Strictly for writing characters and best for writing characters

Section 58.4: DataInputStream Example

```
package com.streams;
import java.io.*;
public class DataStreamDemo {
  public static void main(String[] args) throws IOException {
    InputStream input = new FileInputStream("D:\\datastreamdemo.txt");
    DataInputStream inst = new DataInputStream(input);
    int count = input.available();
    byte[] arr = new byte[count];
    inst.read(arr);
    for (byte byt : arr) {
      char ki = (char) byt;
      System.out.print(ki+"-");
    }
  }
}
```

Section 58.5: Writing bytes to an OutputStream

Writing bytes to an `OutputStream` one byte at a time

```
OutputStream stream = object.getOutputStream();

byte b = 0x00;
stream.write( b );
```

Writing a byte array

```
byte[] bytes = new byte[] { 0x00, 0x00 };

stream.write( bytes );
```

Writing a section of a byte array

```
int offset = 1;
int length = 2;
byte[] bytes = new byte[] { 0xFF, 0x00, 0x00, 0xFF };

stream.write( bytes, offset, length );
```

Section 58.6: Copying Input Stream to Output Stream

This function copies data between two streams -

```
void copy(InputStream in, OutputStream out) throws IOException {
```

```
    byte[] buffer = new byte[8192];
    while ((bytesRead = in.read(buffer)) > 0) {
        out.write(buffer, 0, bytesRead);
    }
}
```

Example -

```
 // reading from System.in and writing to System.out
copy(System.in, System.out);
```

Chapter 59: Readers and Writers

Readers and Writers and their respective subclasses provide simple I/O for text / character-based data.

Section 59.1: BufferedReader

Introduction

The `BufferedReader` class is a wrapper for other `Reader` classes that serves two main purposes:

1. A `BufferedReader` provides buffering for the wrapped `Reader`. This allows an application to read characters one at a time without undue I/O overheads.
2. A `BufferedReader` provides functionality for reading text a line at a time.

Basics of using a BufferedReader

The normal pattern for using a `BufferedReader` is as follows. First, you obtain the `Reader` that you want to read characters from. Next you instantiate a `BufferedReader` that wraps the `Reader`. Then you read character data. Finally you close the `BufferedReader` which close the wrapped `Reader. For example:

```
File someFile = new File(...);
int aCount = 0;
try (FileReader fr = new FileReader(someFile);
     BufferedReader br = new BufferedReader(fr)) {
    // Count the number of 'a' characters.
    int ch;
    while ((ch = br.read()) != -1) {
        if (ch == 'a') {
            aCount++;
        }
    }
    System.out.println("There are " + aCount + " 'a' characters in " + someFile);
}
```

You can apply this pattern to any `Reader`

Notes:

1. We have used Java 7 (or later) *try-with-resources* to ensure that the underlying reader is always closed. This avoids a potential resource leak. In earlier versions of Java, you would explicitly close the `BufferedReader` in a **`finally`** block.
2. The code inside the **`try`** block is virtually identical to what we would use if we read directly from the `FileReader`. In fact, a `BufferedReader` functions exactly like the `Reader` that it wraps would behave. The difference is that *this* version is a lot more efficient.

The BufferedReader buffer size
The BufferedReader.readLine() method
Example: reading all lines of a File into a List

This is done by getting each line in a file, and adding it into a `List<String>`. The list is then returned:

```
public List<String> getAllLines(String filename) throws IOException {
    List<String> lines = new ArrayList<String>();
```

```
    try (BufferedReader br = new BufferedReader(new FileReader(filename))) {
        String line = null;
        while ((line = reader.readLine) != null) {
            lines.add(line);
        }
    }
    return lines;
}
```

Java 8 provides a more concise way to do this using the `lines()` method:

```
public List<String> getAllLines(String filename) throws IOException {
    try (BufferedReader br = new BufferedReader(new FileReader(filename))) {
        return br.lines().collect(Collectors.toList());
    }
    return Collections.empty();
}
```

Section 59.2: StringWriter Example

Java StringWriter class is a character stream that collects output from string buffer, which can be used to construct a string.

The StringWriter class extends the Writer class.

In StringWriter class, system resources like network sockets and files are not used, therefore closing the StringWriter is not necessary.

```
import java.io.*;
public class StringWriterDemo {
    public static void main(String[] args) throws IOException {
        char[] ary = new char[1024];
        StringWriter writer = new StringWriter();
        FileInputStream input = null;
        BufferedReader buffer = null;
        input = new FileInputStream("c://stringwriter.txt");
        buffer = new BufferedReader(new InputStreamReader(input, "UTF-8"));
        int x;
        while ((x = buffer.read(ary)) != -1) {
                  writer.write(ary, 0, x);
        }
        System.out.println(writer.toString());
        writer.close();
        buffer.close();
    }
}
```

The above example helps us to know simple example of StringWriter using BufferedReader to read file data from the stream.

Chapter 60: Preferences

Section 60.1: Using preferences

`Preferences` can be used to store user settings that reflect a user's personal application settings, e.g. their editor font, whether they prefer the application to be started in full-screen mode, whether they checked a "don't show this again" checkbox and things like that.

```
public class ExitConfirmer {
    private static boolean confirmExit() {
        Preferences preferences = Preferences.userNodeForPackage(ExitConfirmer.class);
        boolean doShowDialog = preferences.getBoolean("showExitConfirmation", true); // true is
default value

        if (!doShowDialog) {
            return true;
        }

        //
        // Show a dialog here...
        //
        boolean exitWasConfirmed = ...; // whether the user clicked OK or Cancel
        boolean doNotShowAgain = ...; // get value from "Do not show again" checkbox

        if (exitWasConfirmed && doNotShowAgain) {
            // Exit was confirmed and the user chose that the dialog should not be shown again
            // Save these settings to the Preferences object so the dialog will not show again next
time
            preferences.putBoolean("showExitConfirmation", false);
        }

        return exitWasConfirmed;
    }

    public static void exit() {
        if (confirmExit()) {
            System.exit(0);
        }
    }
}
```

Section 60.2: Adding event listeners

There are two types of events emitted by a `Preferences` object: `PreferenceChangeEvent` and `NodeChangeEvent`.

PreferenceChangeEvent

A `PreferenceChangeEvent` gets emitted by a `Properties` object every time one of the node's key-value-pairs changes. `PreferenceChangeEvents` can be listened for with a `PreferenceChangeListener`:

`Version ≥ Java SE 8`

```
preferences.addPreferenceChangeListener(evt -> {
    String newValue = evt.getNewValue();
    String changedPreferenceKey = evt.getKey();
    Preferences changedNode = evt.getNode();
});
```

`Version < Java SE 8`

```
preferences.addPreferenceChangeListener(new PreferenceChangeListener() {
    @Override
    public void preferenceChange(PreferenceChangeEvent evt) {
        String newValue = evt.getNewValue();
        String changedPreferenceKey = evt.getKey();
        Preferences changedNode = evt.getNode();
    }
});
```

This listener will not listen to changed key-value pairs of child nodes.

NodeChangeEvent

This event will be fired whenever a child node of a `Properties` node is added or removed.

```
preferences.addNodeChangeListener(new NodeChangeListener() {
    @Override
    public void childAdded(NodeChangeEvent evt) {
        Preferences addedChild = evt.getChild();
        Preferences parentOfAddedChild = evt.getParent();
    }

    @Override
    public void childRemoved(NodeChangeEvent evt) {
        Preferences removedChild = evt.getChild();
        Preferences parentOfRemovedChild = evt.getParent();
    }
});
```

Section 60.3: Getting sub-nodes of Preferences

`Preferences` objects always represent a specific node in a whole `Preferences` tree, kind of like this:

rRoot ├── **com** │ └── **mycompany** │ └── **myapp** │ ├── darkApplicationMode=true │ ├── showExitConfirmation=false │ └── windowMaximized=true └── **org** └── **myorganization** └── **anotherapp** ├── defaultFont=Helvetica ├── defaultSavePath=/home/matt/Documents └── **exporting** ├── defaultFormat=pdf └── openInBrowserAfterExport=false

To select the `/com/mycompany/myapp` node:

1. By convention, based on the package of a class:

   ```
   package com.mycompany.myapp;

   // ...

   // Because this class is in the com.mycompany.myapp package, the node
   // /com/mycompany/myapp will be returned.
   Preferences myApp = Preferences.userNodeForPackage(getClass());
   ```

2. By relative path:

   ```
   Preferences myApp = Preferences.userRoot().node("com/mycompany/myapp");
   ```

   Using a relative path (a path not starting with a `/`) will cause the path to be resolved relative to the parent node it is resolved on. For example, the following example will return the node of the path

`/one/two/three/com/mycompany/myapp`:

```
Preferences prefix = Preferences.userRoot().node("one/two/three");
Preferences myAppWithPrefix = prefix.node("com/mycompany/myapp");
// prefix          is /one/two/three
// myAppWithPrefix is /one/two/three/com/mycompany/myapp
```

3. By absolute path:

```
Preferences myApp = Preferences.userRoot().node("/com/mycompany/myapp");
```

Using an absolute path on the root node will not be different from using a relative path. The difference is that, if called on a sub-node, the path will be resolved relative to the root node.

```
Preferences prefix = Preferences.userRoot().node("one/two/three");
Preferences myAppWitoutPrefix = prefix.node("/com/mycompany/myapp");
// prefix            is /one/two/three
// myAppWitoutPrefix is /com/mycompany/myapp
```

Section 60.4: Coordinating preferences access across multiple application instances

All instances of `Preferences` are always thread-safe across the threads of a single Java Virtual Machine (JVM). Because `Preferences` can be shared across multiple JVMs, there are special methods that deal with synchronizing changes across virtual machines.

If you have an application which is supposed to run in a **single instance** only, then **no external synchronization** is required.

If you have an application which runs in **multiple instances** on a single system and therefore `Preferences` access needs to be coordinated between the JVMs on the system, then the **`sync()` method** of any `Preferences` node may be used to ensure changes to the `Preferences` node are visible to other JVMs on the system:

```
// Warning: don't use this if your application is intended
// to only run a single instance on a machine once
// (this is probably the case for most desktop applications)
try {
    preferences.sync();
} catch (BackingStoreException e) {
    // Deal with any errors while saving the preferences to the backing storage
    e.printStackTrace();
}
```

Section 60.5: Exporting preferences

`Preferences` nodes can be exported into a XML document representing that node. The resulting XML tree can be imported again. The resulting XML document will remember whether it was exported from the user or system `Preferences`.

To export a single node, but **not its child nodes**:

Version ≥ Java SE 7

```
try (OutputStream os = ...) {
    preferences.exportNode(os);
} catch (IOException ioe) {
```

```
    // Exception whilst writing data to the OutputStream
    ioe.printStackTrace();
} catch (BackingStoreException bse) {
    // Exception whilst reading from the backing preferences store
    bse.printStackTrace();
}
```

Version < Java SE 7

```
OutputStream os = null;
try {
    os = ...;
    preferences.exportSubtree(os);
} catch (IOException ioe) {
    // Exception whilst writing data to the OutputStream
    ioe.printStackTrace();
} catch (BackingStoreException bse) {
    // Exception whilst reading from the backing preferences store
    bse.printStackTrace();
} finally {
    if (os != null) {
        try {
            os.close();
        } catch (IOException ignored) {}
    }
}
```

To export a single node **with its child nodes**:

Version ≥ Java SE 7

```
try (OutputStream os = ...) {
    preferences.exportNode(os);
} catch (IOException ioe) {
    // Exception whilst writing data to the OutputStream
    ioe.printStackTrace();
} catch (BackingStoreException bse) {
    // Exception whilst reading from the backing preferences store
    bse.printStackTrace();
}
```

Version < Java SE 7

```
OutputStream os = null;
try {
    os = ...;
    preferences.exportSubtree(os);
} catch (IOException ioe) {
    // Exception whilst writing data to the OutputStream
    ioe.printStackTrace();
} catch (BackingStoreException bse) {
    // Exception whilst reading from the backing preferences store
    bse.printStackTrace();
} finally {
    if (os != null) {
        try {
            os.close();
        } catch (IOException ignored) {}
    }
}
```

Section 60.6: Importing preferences

`Preferences` nodes can be imported from a XML document. Importing is meant to be used in conjunction with the

exporting functionality of `Preferences`, since it creates the correct corresponding XML documents.

The XML documents will remember whether they were exported from the user or system `Preferences`. Therefore, they can be imported into their respective `Preferences` trees again, without you having to figure out or know where they came from. The static function will automatically find out whether the XML document was exported from the user or system `Preferences` and will automatically import them into the tree they were exported from.

Version ≥ Java SE 7

```
try (InputStream is = ...) {
    // This is a static call on the Preferences class
    Preferences.importPreferences(is);
} catch (IOException ioe) {
    // Exception whilst reading data from the InputStream
    ioe.printStackTrace();
} catch (InvalidPreferencesFormatException ipfe) {
    // Exception whilst parsing the XML document tree
    ipfe.printStackTrace();
}
```

Version < Java SE 7

```
InputStream is = null;
try {
    is = ...;
    // This is a static call on the Preferences class
    Preferences.importPreferences(is);
} catch (IOException ioe) {
    // Exception whilst reading data from the InputStream
    ioe.printStackTrace();
} catch (InvalidPreferencesFormatException ipfe) {
    // Exception whilst parsing the XML document tree
    ipfe.printStackTrace();
} finally {
    if (is != null) {
        try {
            is.close();
        } catch (IOException ignored) {}
    }
}
```

Section 60.7: Removing event listeners

Event listeners can be removed again from any `Properties` node, but the instance of the listener has to be kept around for that.

Version ≥ Java SE 8

```
Preferences preferences = Preferences.userNodeForPackage(getClass());

PreferenceChangeListener listener = evt -> {
    System.out.println(evt.getKey() + " got new value " + evt.getNewValue());
};
preferences.addPreferenceChangeListener(listener);

//
// later...
//

preferences.removePreferenceChangeListener(listener);
```

Version < Java SE 8

```
Preferences preferences = Preferences.userNodeForPackage(getClass());
```

```
PreferenceChangeListener listener = new PreferenceChangeListener() {
    @Override
    public void preferenceChange(PreferenceChangeEvent evt) {
        System.out.println(evt.getKey() + " got new value " + evt.getNewValue());
    }
};
preferences.addPreferenceChangeListener(listener);

//
// later...
//

preferences.removePreferenceChangeListener(listener);
```

The same applies for `NodeChangeListener`.

Section 60.8: Getting preferences values

A value of a `Preferences` node can be of the type `String`, **`boolean`**, **`byte`**`[]`, **`double`**, **`float`**, **`int`** or **`long`**. All invocations must provide a default value, in case the specified value is not present in the `Preferences` node.

```
Preferences preferences = Preferences.userNodeForPackage(getClass());

String someString = preferences.get("someKey", "this is the default value");
boolean someBoolean = preferences.getBoolean("someKey", true);
byte[] someByteArray = preferences.getByteArray("someKey", new byte[0]);
double someDouble = preferences.getDouble("someKey", 887284.4d);
float someFloat = preferences.getFloat("someKey", 38723.3f);
int someInt = preferences.getInt("someKey", 13232);
long someLong = preferences.getLong("someKey", 2827637868234L);
```

Section 60.9: Setting preferences values

To store a value into the `Preferences` node, one of the `putXXX()` methods is used. A value of a `Preferences` node can be of the type `String`, **`boolean`**, **`byte`**`[]`, **`double`**, **`float`**, **`int`** or **`long`**.

```
Preferences preferences = Preferences.userNodeForPackage(getClass());

preferences.put("someKey", "some String value");
preferences.putBoolean("someKey", false);
preferences.putByteArray("someKey", new byte[0]);
preferences.putDouble("someKey", 187398123.4454d);
preferences.putFloat("someKey", 298321.445f);
preferences.putInt("someKey", 77637);
preferences.putLong("someKey", 2873984729834L);
```

Chapter 61: Collection Factory Methods

Method w/ Parameter	Description
`List.of(E e)`	A generic type that can be a class or interface.
`Set.of(E e)`	A generic type that can be a class or interface.
`Map.of(K k, V v)`	A key-value pair of generic types that can each be a class or interface.
`Map.of(Map.Entry<? extends K, ? extends V> entry)`	A `Map.Entry` instance where its key can be `K` or one of its children, and its value can be `V` or any of its children.

The arrival of Java 9 brings many new features to Java's Collections API, one of which being collection factory methods. These methods allow for easy initialization of **immutable** collections, whether they be empty or nonempty.

Note that these factory methods are only available for the following interfaces: `List<E>`, `Set<E>`, and `Map<K, V>`

Section 61.1: List<E> Factory Method Examples

- `List<Integer> immutableEmptyList = List.of();`
  - Initializes an empty, immutable `List<Integer>`.
- `List<Integer> immutableList = List.of(1, 2, 3, 4, 5);`
  - Initializes an immutable `List<Integer>` with five initial elements.
- `List<Integer> mutableList = new ArrayList<>(immutableList);`
  - Initializes a mutable `List<Integer>` from an immutable `List<Integer>`.

Section 61.2: Set<E> Factory Method Examples

- `Set<Integer> immutableEmptySet = Set.of();`
  - Initializes an empty, immutable `Set<Integer>`.
- `Set<Integer> immutableSet = Set.of(1, 2, 3, 4, 5);`
  - Initializes an immutable `Set<Integer>` with five initial elements.
- `Set<Integer> mutableSet = new HashSet<>(immutableSet);`
  - Initializes a mutable `Set<Integer>` from an immutable `Set<Integer>`.

Section 61.3: Map<K, V> Factory Method Examples

- `Map<Integer, Integer> immutableEmptyMap = Map.of();`
  - Initializes an empty, immutable `Map<Integer, Integer>`.
- `Map<Integer, Integer> immutableMap = Map.of(1, 2, 3, 4);`
  - Initializes an immutable `Map<Integer, Integer>` with two initial key-value entries.
- `Map<Integer, Integer> immutableMap = Map.ofEntries(Map.entry(1, 2), Map.entry(3, 4));`
  - Initializes an immutable `Map<Integer, Integer>` with two initial key-value entries.
- `Map<Integer, Integer> mutableMap = new HashMap<>(immutableMap);`
  - Initializes a mutable `Map<Integer, Integer>` from an immutable `Map<Integer, Integer>`.

Chapter 62: Alternative Collections

Section 62.1: Multimap in Guava, Apache and Eclipse Collections

This multimap allows duplicate key-value pairs. JDK analogs are HashMap<K, List>, HashMap<K, Set> and so on.

Key's order	Value's order	Duplicate	Analog key	Analog value	Guava	Apache	Eclipse (GS) Collections	JDK
not defined	Insertion-order	yes	HashMap	ArrayList	ArrayListMultimap	`MultiValueMap`	`FastListMultimap`	`HashMap<K, ArrayList<V>>`
not defined	not defined	no	HashMap	HashSet	HashMultimap	`MultiValueMap.multiValueMap( new HashMap<K, Set>(), HashSet.class);`	`UnifiedSetMultimap`	`HashMap<K, HashSet<V>>`
not defined	sorted	no	HashMap	TreeSet	`Multimaps.newMultimap( HashMap, Supplier <TreeSet>)`	`MultiValueMap.multiValueMap( new HashMap<K, Set>(), TreeSet.class)`	`TreeSortedSet-Multimap`	`HashMap<K, TreeSet<V>>`
Insertion-order	Insertion-order	yes	LinkedHashMap	ArrayList	LinkedListMultimap	MultiValueMap. multiValueMap(new LinkedHashMap<K, List>(), ArrayList.class);		LinkedHashMap< K, ArrayList>
Insertion-order	Insertion-order	no	LinkedHashMap	LinkedHashSet	LinkedHashMultimap	`MultiValueMap.multiValueMap(new LinkedHashMap<K, Set>(), LinkedHashSet.class)`		`LinkedHashMap<K, LinkedHashSet<V>>`
sorted	sorted	no	TreeMap	TreeSet	TreeMultimap	`MultiValueMap.multiValueMap( new TreeMap<K, Set>(),TreeSet.class)`		`TreeMap<K, TreeSet<V>>`

Examples using Multimap

Task: Parse "Hello World! Hello All! Hi World!" string to separate words and print all indexes of every word using MultiMap (for example, Hello=[0, 2], World!=[1, 5] and so on)

1. MultiValueMap from Apache

```
    String INPUT_TEXT = "Hello World! Hello All! Hi World!";
    // Parse text to words and index
    List<String> words = Arrays.asList(INPUT_TEXT.split(" "));
    // Create Multimap
    MultiMap<String, Integer> multiMap = new MultiValueMap<String, Integer>();


    // Fill Multimap
    int i = 0;
    for(String word: words) {
        multiMap.put(word, i);
        i++;
    }

    // Print all words
    System.out.println(multiMap); // print {Hi=[4], Hello=[0, 2], World!=[1, 5], All!=[3]} - in
random orders
    // Print all unique words
    System.out.println(multiMap.keySet());    // print [Hi, Hello, World!, All!] - in random orders

    // Print all indexes
    System.out.println("Hello = " + multiMap.get("Hello"));    // print [0, 2]
    System.out.println("World = " + multiMap.get("World!"));    // print [1, 5]
    System.out.println("All = " + multiMap.get("All!"));    // print [3]
    System.out.println("Hi = " + multiMap.get("Hi"));    // print [4]
    System.out.println("Empty = " + multiMap.get("Empty"));    // print null

    // Print count unique words
```

```
    System.out.println(multiMap.keySet().size());    //print 4
```

2. HashBiMap from GS / Eclipse Collection

```
    String[] englishWords = {"one", "two", "three","ball","snow"};
    String[] russianWords = {"jeden", "dwa", "trzy", "kula", "snieg"};

    // Create Multiset
    MutableBiMap<String, String> biMap = new HashBiMap(englishWords.length);
    // Create English-Polish dictionary
    int i = 0;
    for(String englishWord: englishWords) {
        biMap.put(englishWord, russianWords[i]);
        i++;
    }

    // Print count words
    System.out.println(biMap); // print {two=dwa, ball=kula, one=jeden, snow=snieg, three=trzy} - in
random orders
    // Print all unique words
    System.out.println(biMap.keySet());    // print [snow, two, one, three, ball] - in random orders
    System.out.println(biMap.values());    // print [dwa, kula, jeden, snieg, trzy] - in random
orders

    // Print translate by words
    System.out.println("one = " + biMap.get("one"));    // print one = jeden
    System.out.println("two = " + biMap.get("two"));    // print two = dwa
    System.out.println("kula = " + biMap.inverse().get("kula"));    // print kula = ball
    System.out.println("snieg = " + biMap.inverse().get("snieg"));    // print snieg = snow
    System.out.println("empty = " + biMap.get("empty"));    // print empty = null

    // Print count word's pair
    System.out.println(biMap.size());    //print 5
```

3. HashMultiMap from Guava

```
 String INPUT_TEXT = "Hello World! Hello All! Hi World!";
 // Parse text to words and index
 List<String> words = Arrays.asList(INPUT_TEXT.split(" "));
 // Create Multimap
 Multimap<String, Integer> multiMap = HashMultimap.create();

 // Fill Multimap
 int i = 0;
 for(String word: words) {
     multiMap.put(word, i);
     i++;
 }

 // Print all words
 System.out.println(multiMap); // print {Hi=[4], Hello=[0, 2], World!=[1, 5], All!=[3]} - keys
and values in random orders
 // Print all unique words
 System.out.println(multiMap.keySet());    // print [Hi, Hello, World!, All!] - in random
orders

 // Print all indexes
 System.out.println("Hello = " + multiMap.get("Hello"));    // print [0, 2]
 System.out.println("World = " + multiMap.get("World!"));    // print [1, 5]
 System.out.println("All = " + multiMap.get("All!"));    // print [3]
 System.out.println("Hi = " + multiMap.get("Hi"));    // print [4]
```

```
System.out.println("Empty = " + multiMap.get("Empty"));    // print []

// Print count all words
System.out.println(multiMap.size());    //print 6

// Print count unique words
System.out.println(multiMap.keySet().size());    //print 4
```

Nore examples:

I. Apache Collection:

1. MultiValueMap
2. MultiValueMapLinked
3. MultiValueMapTree

II. GS / Eclipse Collection

1. FastListMultimap
2. HashBagMultimap
3. TreeSortedSetMultimap
4. UnifiedSetMultimap

III. Guava

1. HashMultiMap
2. LinkedHashMultimap
3. LinkedListMultimap
4. TreeMultimap
5. ArrayListMultimap

Section 62.2: Apache HashBag, Guava HashMultiset and Eclipse HashBag

A Bag/ultiset stores each object in the collection together with a count of occurrences. Extra methods on the interface allow multiple copies of an object to be added or removed at once. JDK analog is HashMap<T, Integer>, when values is count of copies this key.

Type	Guava	Apache Commons Collections	GS Collections	JDK
Order not defined	HashMultiset	HashBag	HashBag	HashMap
Sorted	TreeMultiset	TreeBag	TreeBag	TreeMap
Insertion-order	LinkedHashMultiset	-	-	LinkedHashMap
Concurrent variant	ConcurrentHashMultiset	SynchronizedBag	SynchronizedBag	`Collections.synchronizedMap(HashMap<String, Integer>)`
Concurrent and sorted	-	SynchronizedSortedBag	SynchronizedSortedBag	`Collections.synchronizedSortedMap(TreeMap<String,Integer>)`
Immutable collection	ImmutableMultiset	UnmodifiableBag	UnmodifiableBag	`Collections.unmodifiableMap(HashMap<String, Integer>)]`
Immutable and sorted	ImmutableSortedMultiset	UnmodifiableSortedBag	UnmodifiableSortedBag	`Collections.unmodifiableSortedMap(TreeMap<String, Integer>)`

Examples:

1. Using SynchronizedSortedBag from Apache:

```
// Parse text to separate words
String INPUT_TEXT = "Hello World! Hello All! Hi World!";
```

```
    // Create Multiset
    Bag bag = SynchronizedSortedBag.synchronizedBag(new TreeBag(Arrays.asList(INPUT_TEXT.split("
")))));

    // Print count words
    System.out.println(bag); // print [1:All!,2:Hello,1:Hi,2:World!]- in natural (alphabet) order
    // Print all unique words
    System.out.println(bag.uniqueSet());    // print [All!, Hello, Hi, World!]- in natural
(alphabet) order


    // Print count occurrences of words
    System.out.println("Hello = " + bag.getCount("Hello"));    // print 2
    System.out.println("World = " + bag.getCount("World!"));    // print 2
    System.out.println("All = " + bag.getCount("All!"));    // print 1
    System.out.println("Hi = " + bag.getCount("Hi"));    // print 1
    System.out.println("Empty = " + bag.getCount("Empty"));    // print 0

    // Print count all words
    System.out.println(bag.size());    //print 6

    // Print count unique words
    System.out.println(bag.uniqueSet().size());    //print 4
```

2. Using TreeBag from Eclipse(GC):

```
    // Parse text to separate words
    String INPUT_TEXT = "Hello World! Hello All! Hi World!";
    // Create Multiset
    MutableSortedBag<String> bag =  TreeBag.newBag(Arrays.asList(INPUT_TEXT.split(" ")));

    // Print count words
    System.out.println(bag); // print [All!, Hello, Hello, Hi, World!, World!]- in natural order
    // Print all unique words
    System.out.println(bag.toSortedSet());    // print [All!, Hello, Hi, World!]- in natural order

    // Print count occurrences of words
    System.out.println("Hello = " + bag.occurrencesOf("Hello"));    // print 2
    System.out.println("World = " + bag.occurrencesOf("World!"));    // print 2
    System.out.println("All = " + bag.occurrencesOf("All!"));    // print 1
    System.out.println("Hi = " + bag.occurrencesOf("Hi"));    // print 1
    System.out.println("Empty = " + bag.occurrencesOf("Empty"));    // print 0

    // Print count all words
    System.out.println(bag.size());    //print 6

    // Print count unique words
    System.out.println(bag.toSet().size());    //print 4
```

3. Using LinkedHashMultiset from Guava:

```
    // Parse text to separate words
    String INPUT_TEXT = "Hello World! Hello All! Hi World!";
    // Create Multiset
    Multiset<String> multiset = LinkedHashMultiset.create(Arrays.asList(INPUT_TEXT.split(" ")));

    // Print count words
    System.out.println(multiset); // print [Hello x 2, World! x 2, All!, Hi]- in predictable
iteration order
    // Print all unique words
    System.out.println(multiset.elementSet());    // print [Hello, World!, All!, Hi] - in
predictable iteration order

    // Print count occurrences of words
```

```
    System.out.println("Hello = " + multiset.count("Hello"));    // print 2
    System.out.println("World = " + multiset.count("World!"));    // print 2
    System.out.println("All = " + multiset.count("All!"));    // print 1
    System.out.println("Hi = " + multiset.count("Hi"));    // print 1
    System.out.println("Empty = " + multiset.count("Empty"));    // print 0

    // Print count all words
    System.out.println(multiset.size());    //print 6

    // Print count unique words
    System.out.println(multiset.elementSet().size());    //print 4
```

More examples:

I. Apache Collection:

1. HashBag - order not defined
2. SynchronizedBag - concurrent and order not defined
3. SynchronizedSortedBag - - concurrent and sorted order
4. TreeBag - sorted order

II. GS / Eclipse Collection

5. MutableBag - order not defined
6. MutableSortedBag - sorted order

III. Guava

7. HashMultiset - order not defined
8. TreeMultiset - sorted order
9. LinkedHashMultiset - insertion order
10. ConcurrentHashMultiset - concurrent and order not defined

Section 62.3: Compare operation with collections - Create collections

Compare operation with collections - Create collections

1. Create List

Description	JDK	guava	gs-collections
Create empty list	`new ArrayList<>()`	`Lists.newArrayList()`	`FastList.newList()`
Create list from values	`Arrays.asList("1", "2", "3")`	`Lists.newArrayList("1", "2", "3")`	`FastList.newListWith("1", "2", "3")`
Create list with capacity = 100	`new ArrayList<>(100)`	`Lists.newArrayListWithCapacity(100)`	`FastList.newList(100)`
Create list from any collectin	`new ArrayList<>(collection)`	`Lists.newArrayList(collection)`	`FastList.newList(collection)`
Create list from any Iterable	-	`Lists.newArrayList(iterable)`	`FastList.newList(iterable)`
Create list from Iterator	-	`Lists.newArrayList(iterator)`	-

Create list from array	`Arrays.asList(array)`	`Lists.newArrayList(array)`	`FastList.newListWith(array)`
Create list using factory	-	-	`FastList.newWithNValues(10, () -> "1")`

Examples:

```
    System.out.println("createArrayList start");
    // Create empty list
    List<String> emptyGuava = Lists.newArrayList(); // using guava
    List<String> emptyJDK = new ArrayList<>(); // using JDK
    MutableList<String>  emptyGS = FastList.newList(); // using gs

    // Create list with 100 element
    List < String > exactly100 = Lists.newArrayListWithCapacity(100); // using guava
    List<String> exactly100JDK = new ArrayList<>(100); // using JDK
    MutableList<String>  empty100GS = FastList.newList(100); // using gs

    // Create list with about 100 element
    List<String> approx100 = Lists.newArrayListWithExpectedSize(100); // using guava
    List<String> approx100JDK = new ArrayList<>(115); // using JDK
    MutableList<String>  approx100GS = FastList.newList(115); // using gs

    // Create list with some elements
    List<String> withElements = Lists.newArrayList("alpha", "beta", "gamma"); // using guava
    List<String> withElementsJDK = Arrays.asList("alpha", "beta", "gamma"); // using JDK
    MutableList<String>  withElementsGS = FastList.newListWith("alpha", "beta", "gamma"); // using
gs

    System.out.println(withElements);
    System.out.println(withElementsJDK);
    System.out.println(withElementsGS);

    // Create list from any Iterable interface (any collection)
    Collection<String> collection = new HashSet<>(3);
    collection.add("1");
    collection.add("2");
    collection.add("3");

    List<String> fromIterable = Lists.newArrayList(collection); // using guava
    List<String> fromIterableJDK = new ArrayList<>(collection); // using JDK
    MutableList<String>  fromIterableGS = FastList.newList(collection); // using gs

    System.out.println(fromIterable);
    System.out.println(fromIterableJDK);
    System.out.println(fromIterableGS);
    /* Attention: JDK create list only from Collection, but guava and gs can create list from
Iterable and Collection */

    // Create list from any Iterator
    Iterator<String> iterator = collection.iterator();
    List<String> fromIterator = Lists.newArrayList(iterator); // using guava
    System.out.println(fromIterator);

    // Create list from any array
    String[] array = {"4", "5", "6"};
    List<String> fromArray = Lists.newArrayList(array); // using guava
    List<String> fromArrayJDK = Arrays.asList(array); // using JDK
    MutableList<String>  fromArrayGS = FastList.newListWith(array); // using gs
    System.out.println(fromArray);
```

```
    System.out.println(fromArrayJDK);
    System.out.println(fromArrayGS);

    // Create list using fabric
    MutableList<String>  fromFabricGS = FastList.newWithNValues(10, () ->
String.valueOf(Math.random())); // using gs
    System.out.println(fromFabricGS);

    System.out.println("createArrayList end");
```

2 Create Set

Description	JDK	guava	gs-collections
Create empty set	`new HashSet<>()`	`Sets.newHashSet()`	`UnifiedSet.newSet()`
Creatre set from values	`new HashSet<>(Arrays.asList("alpha", "beta", "gamma"))`	`Sets.newHashSet("alpha", "beta", "gamma")`	`UnifiedSet.newSetWith("alpha", "beta", "gamma")`
Create set from any collections	`new HashSet<>(collection)`	`Sets.newHashSet(collection)`	`UnifiedSet.newSet(collection)`
Create set from any Iterable	-	`Sets.newHashSet(iterable)`	`UnifiedSet.newSet(iterable)`
Create set from any Iterator	-	`Sets.newHashSet(iterator)`	-
Create set from Array	`new HashSet<>(Arrays.asList(array))`	`Sets.newHashSet(array)`	`UnifiedSet.newSetWith(array)`

Examples:

```
    System.out.println("createHashSet start");
    // Create empty set
    Set<String> emptyGuava = Sets.newHashSet(); // using guava
    Set<String> emptyJDK = new HashSet<>(); // using JDK
    Set<String> emptyGS = UnifiedSet.newSet(); // using gs

    // Create set with 100 element
    Set<String> approx100 = Sets.newHashSetWithExpectedSize(100); // using guava
    Set<String> approx100JDK = new HashSet<>(130); // using JDK
    Set<String> approx100GS = UnifiedSet.newSet(130); // using gs

    // Create set from some elements
    Set<String> withElements =  Sets.newHashSet("alpha", "beta", "gamma"); // using guava
    Set<String> withElementsJDK = new HashSet<>(Arrays.asList("alpha", "beta", "gamma")); // using
JDK
    Set<String> withElementsGS  = UnifiedSet.newSetWith("alpha", "beta", "gamma"); // using gs

    System.out.println(withElements);
    System.out.println(withElementsJDK);
    System.out.println(withElementsGS);

    // Create set from any Iterable interface (any collection)
    Collection<String> collection = new ArrayList<>(3);
    collection.add("1");
    collection.add("2");
    collection.add("3");

    Set<String> fromIterable = Sets.newHashSet(collection); // using guava
    Set<String> fromIterableJDK = new HashSet<>(collection); // using JDK
    Set<String> fromIterableGS  = UnifiedSet.newSet(collection); // using gs
```

```
    System.out.println(fromIterable);
    System.out.println(fromIterableJDK);
    System.out.println(fromIterableGS);
    /* Attention: JDK create set only from Collection, but guava and gs can create set from Iterable
and Collection */

    // Create set from any Iterator
    Iterator<String> iterator = collection.iterator();
    Set<String> fromIterator = Sets.newHashSet(iterator); // using guava
    System.out.println(fromIterator);

    // Create set from any array
    String[] array = {"4", "5", "6"};
    Set<String> fromArray = Sets.newHashSet(array); // using guava
    Set<String> fromArrayJDK = new HashSet<>(Arrays.asList(array)); // using JDK
    Set<String> fromArrayGS  = UnifiedSet.newSetWith(array); // using gs
    System.out.println(fromArray);
    System.out.println(fromArrayJDK);
    System.out.println(fromArrayGS);

    System.out.println("createHashSet end");
```

3 Create Map

Description	JDK	guava	gs-collections
Create empty map	`new HashMap<>()`	`Maps.newHashMap()`	`UnifiedMap.newMap()`
Create map with capacity = 130	`new HashMap<>(130)`	`Maps.newHashMapWithExpectedSize(100)`	`UnifiedMap.newMap(130)`
Create map from other map	`new HashMap<>(map)`	`Maps.newHashMap(map)`	`UnifiedMap.newMap(map)`
Create map from keys	-	-	`UnifiedMap.newWithKeysValues("1", "a", "2", "b")`

Examples:

```
    System.out.println("createHashMap start");
    // Create empty map
    Map<String, String> emptyGuava = Maps.newHashMap(); // using guava
    Map<String, String> emptyJDK = new HashMap<>(); // using JDK
    Map<String, String> emptyGS = UnifiedMap.newMap(); // using gs

    // Create map with about 100 element
    Map<String, String> approx100 = Maps.newHashMapWithExpectedSize(100); // using guava
    Map<String, String> approx100JDK = new HashMap<>(130); // using JDK
    Map<String, String> approx100GS = UnifiedMap.newMap(130); // using gs

    // Create map from another map
    Map<String, String> map = new HashMap<>(3);
    map.put("k1","v1");
    map.put("k2","v2");
    Map<String, String> withMap =  Maps.newHashMap(map); // using guava
    Map<String, String> withMapJDK = new HashMap<>(map); // using JDK
    Map<String, String> withMapGS = UnifiedMap.newMap(map); // using gs

    System.out.println(withMap);
    System.out.println(withMapJDK);
    System.out.println(withMapGS);

    // Create map from keys
```

```
    Map<String, String> withKeys =  UnifiedMap.newWithKeysValues("1", "a", "2", "b");
    System.out.println(withKeys);

    System.out.println("createHashMap end");
```

More examples: CreateCollectionTest

1. CollectionCompare
2. CollectionSearch
3. JavaTransform

Chapter 63: Concurrent Collections

A *concurrent collection* is a [collection][1] which permits access by more than one thread at the same time. Different threads can typically iterate through the contents of the collection and add or remove elements. The collection is responsible for ensuring that the collection doesn't become corrupt. [1]: http://stackoverflow.com/documentation/java/90/collections#t=20161222193649729848 4

Section 63.1: Thread-safe Collections

By default, the various Collection types are not thread-safe.

However, it's fairly easy to make a collection thread-safe.

```
List<String> threadSafeList = Collections.synchronizedList(new ArrayList<String>());
Set<String> threadSafeSet = Collections.synchronizedSet(new HashSet<String>());
Map<String, String> threadSafeMap = Collections.synchronizedMap(new HashMap<String, String>());
```

When you make a thread-safe collection, you should never access it through the original collection, only through the thread-safe wrapper.

Version ≥ Java SE 5

Starting in Java 5, `java.util.collections` has several new thread-safe collections that don't need the various `Collections.synchronized` methods.

```
List<String> threadSafeList = new CopyOnWriteArrayList<String>();
Set<String> threadSafeSet = new ConcurrentHashSet<String>();
Map<String, String> threadSafeMap = new ConcurrentHashMap<String, String>();
```

Section 63.2: Insertion into ConcurrentHashMap

```
public class InsertIntoConcurrentHashMap
{

    public static void main(String[] args)
    {
        ConcurrentHashMap<Integer, SomeObject> concurrentHashMap = new ConcurrentHashMap<>();

        SomeObject value = new SomeObject();
        Integer key = 1;

        SomeObject previousValue = concurrentHashMap.putIfAbsent(1, value);
        if (previousValue != null)
        {
            //Then some other value was mapped to key = 1. 'value' that was passed to
            //putIfAbsent method is NOT inserted, hence, any other thread which calls
            //concurrentHashMap.get(1) would NOT receive a reference to the 'value'
            //that your thread attempted to insert. Decide how you wish to handle
            //this situation.
        }

       else
        {
            //'value' reference is mapped to key = 1.
        }
    }
```

```
}
```

Section 63.3: Concurrent Collections

Concurrent collections are a generalization of thread-safe collections, that allow for a broader usage in a concurrent environment.

While thread-safe collections have safe element addition or removal from multiple threads, they do not necessarily have safe iteration in the same context (one may not be able to safely iterate through the collection in one thread, while another one modifies it by adding/removing elements).

This is where concurrent collections are used.

As iteration is often the base implementation of several bulk methods in collections, like `addAll`, `removeAll`, or also collection copying (through a constructor, or other means), sorting, ... the use case for concurrent collections is actually pretty large.

For example, the Java SE 5 `java.util.concurrent.CopyOnWriteArrayList` is a thread safe and concurrent `List` implementation, its javadoc states :

> The "snapshot" style iterator method uses a reference to the state of the array at the point that the iterator was created. This array never changes during the lifetime of the iterator, so interference is impossible and the iterator is guaranteed not to throw ConcurrentModificationException.

Therefore, the following code is safe :

```
public class ThreadSafeAndConcurrent {

public static final List<Integer> LIST = new CopyOnWriteArrayList<>();

public static void main(String[] args) throws InterruptedException {
    Thread modifier = new Thread(new ModifierRunnable());
    Thread iterator = new Thread(new IteratorRunnable());
    modifier.start();
    iterator.start();
    modifier.join();
    iterator.join();
}

public static final class ModifierRunnable implements Runnable {
    @Override
    public void run() {
        try {
            for (int i = 0; i < 50000; i++) {
                LIST.add(i);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

public static final class IteratorRunnable implements Runnable {
    @Override
    public void run() {
        try {
```

```
            for (int i = 0; i < 10000; i++) {
                long total = 0;
                for(Integer inList : LIST) {
                    total += inList;
                }
                System.out.println(total);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
}
```

Another concurrent collection regarding iteration is `ConcurrentLinkedQueue`, which states :

> Iterators are weakly consistent, returning elements reflecting the state of the queue at some point at or since the creation of the iterator. They do not throw java.util.ConcurrentModificationException, and may proceed concurrently with other operations. Elements contained in the queue since the creation of the iterator will be returned exactly once.

One should check the javadocs to see if a collection is concurrent, or not. The attributes of the iterator returned by the `iterator()` method ("fail fast", "weakly consistent", ...) is the most important attribute to look for.

Thread safe but non concurrent examples

In the above code, changing the `LIST` declaration to

```
public static final List<Integer> LIST = Collections.synchronizedList(new ArrayList<>());
```

Could (and statistically will on most modern, multi CPU/core architectures) lead to exceptions.

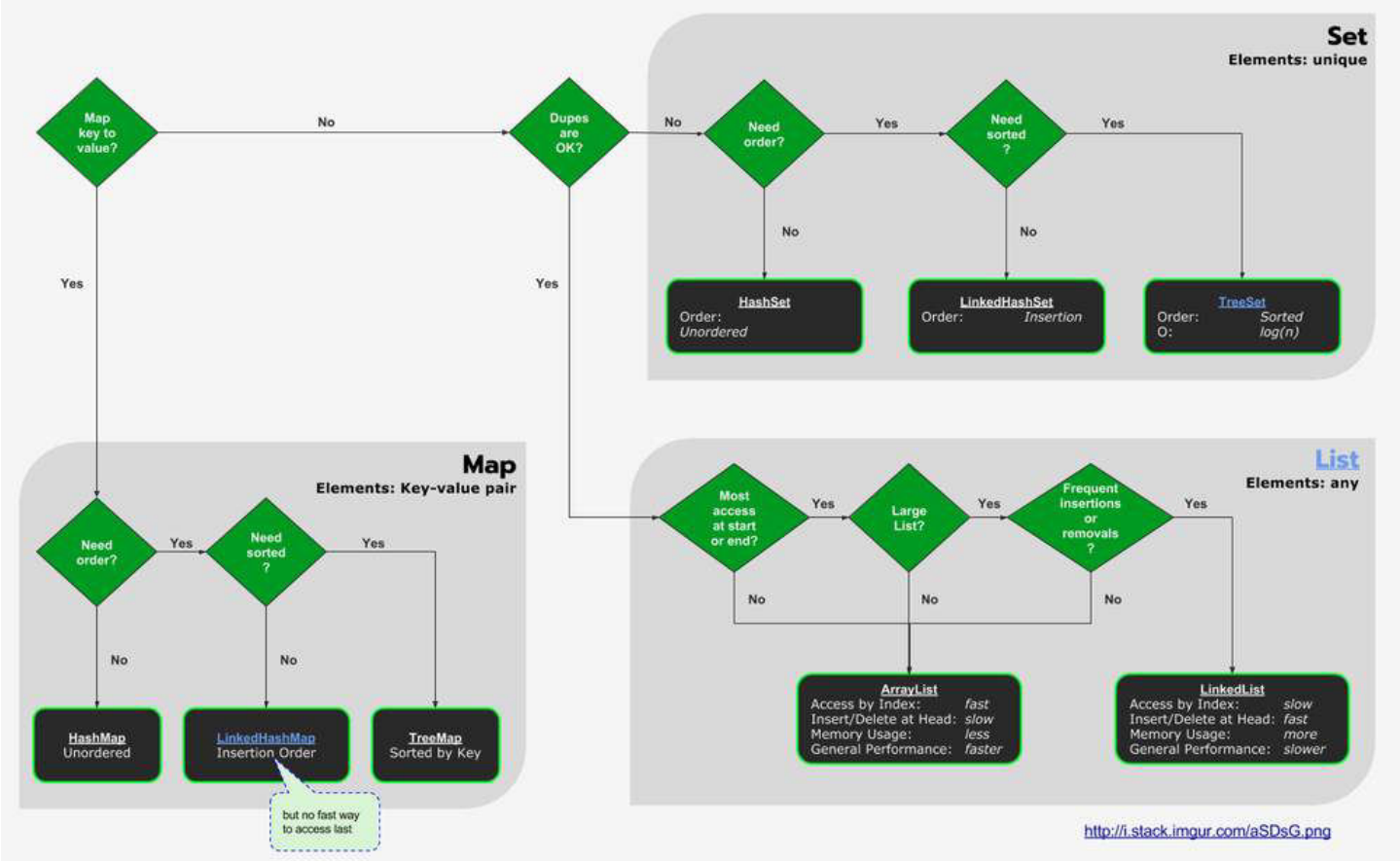
Synchronized collections from the `Collections` utility methods are thread safe for addition/removal of elements, but not iteration (unless the underlying collection being passed to it already is).

Chapter 64: Choosing Collections

Java offers a wide variety of Collections. Choosing which Collection to use can be tricky. See the Examples section for an easy-to-follow flowchart to choose the right Collection for the job.

Section 64.1: Java Collections Flowchart

Use the following flowchart to choose the right Collection for the job.



This flowchart was based off [http://i.stack.imgur.com/aSDsG.png).

Chapter 65: super keyword

Section 65.1: Super keyword use with examples

super keyword performs important role in three places

1. Constructor Level
2. Method Level
3. Variable Level

Constructor Level

`super` keyword is used to call parent class constructor. This constructor can be default constructor or parameterized constructor.

- Default constructor : `super();`
- Parameterized constructor : `super(int no, double amount, String name);`

```
class Parentclass
{
   Parentclass(){
      System.out.println("Constructor of Superclass");
   }
}
class Subclass extends Parentclass
{
   Subclass(){
    /* Compile adds super() here at the first line
     * of this constructor implicitly
     */
    System.out.println("Constructor of Subclass");
   }
   Subclass(int n1){
    /* Compile adds super() here at the first line
     * of this constructor implicitly
     */
    System.out.println("Constructor with arg");
   }
   void display(){
    System.out.println("Hello");
   }
   public static void main(String args[]){
    // Creating object using default constructor
    Subclass obj= new Subclass();
    //Calling sub class method
       obj.display();
       //Creating object 2 using arg constructor
       Subclass obj2= new Subclass(10);
       obj2.display();
  }
}
```

Note: `super()` must be the first statement in constructor otherwise we will get the compilation error message.

Method Level

`super` keyword can also be used in case of method overriding. `super` keyword can be used to invoke or call parent class method.

```
class Parentclass
{
   //Overridden method
   void display(){
    System.out.println("Parent class method");
   }
}
class Subclass extends Parentclass
{
   //Overriding method
   void display(){
    System.out.println("Child class method");
   }
   void printMsg(){
    //This would call Overriding method
    display();
    //This would call Overridden method
    super.display();
   }
   public static void main(String args[]){
    Subclass obj= new Subclass();
    obj.printMsg();
   }
}
```

Note:If there is not method overriding then we do not need to use `super` keyword to call parent class method.

Variable Level

`super` is used to refer immediate parent class instance variable. In case of inheritance, there may be possibility of base class and derived class may have similar data members.In order to differentiate between the data member of base/parent class and derived/child class, in the context of derived class the base class data members must be preceded by `super` keyword.

```
//Parent class or Superclass
class Parentclass
{
    int num=100;
}
//Child class or subclass
class Subclass extends Parentclass
{
    /* I am declaring the same variable
     * num in child class too.
     */
    int num=110;
    void printNumber(){
     System.out.println(num); //It will print value 110
     System.out.println(super.num); //It will print value 100
    }
    public static void main(String args[]){
       Subclass obj= new Subclass();
       obj.printNumber();
    }
}
```

Note: If we are not writing **`super`** keyword before the base class data member name then it will be referred as current class data member and base class data member are hidden in the context of derived class.

Chapter 66: Serialization

Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Section 66.1: Basic Serialization in Java

What is Serialization

Serialization is the process of converting an object's state (including its references) to a sequence of bytes, as well as the process of rebuilding those bytes into a live object at some future time. Serialization is used when you want to persist the object. It is also used by Java RMI to pass objects between JVMs, either as arguments in a method invocation from a client to a server or as return values from a method invocation, or as exceptions thrown by remote methods. In general, serialization is used when we want the object to exist beyond the lifetime of the JVM.

`java.io.Serializable` is a marker interface (has no body). It is just used to "mark" Java classes as serializable.

The serialization runtime associates with each serializable class a version number, called a `serialVersionUID`, which is used during *de*-serialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization. If the receiver has loaded a class for the object that has a different `serialVersionUID` than that of the corresponding sender's class, then deserialization will result in an `InvalidClassException`. A serializable class can declare its own `serialVersionUID` explicitly by declaring a field named `serialVersionUID` that must be **`static`**, **`final`**, and of type **`long`**:

```
ANY-ACCESS-MODIFIER static final long serialVersionUID = 1L;
```

How to make a class eligible for serialization

To persist an object the respective class must implement the `java.io.Serializable` interface.

```
import java.io.Serializable;

public class SerialClass implements Serializable {

    private static final long serialVersionUID = 1L;
    private Date currentTime;

    public SerialClass() {
        currentTime = Calendar.getInstance().getTime();
    }

    public Date getCurrentTime() {
        return currentTime;
    }
}
```

How to write an object into a file

Now we need to write this object to a file system. We use `java.io.ObjectOutputStream` for this purpose.

```
import java.io.FileOutputStream;
```

```
import java.io.ObjectOutputStream;
import java.io.IOException;

public class PersistSerialClass {

    public static void main(String [] args) {
        String filename = "time.ser";
        SerialClass time = new SerialClass(); //We will write this object to file system.
        try {
            ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(filename));
            out.writeObject(time); //Write byte stream to file system.
            out.close();
        } catch(IOException ex){
            ex.printStackTrace();
        }
     }
}
```

How to recreate an object from its serialized state

The stored object can be read from file system at later time using `java.io.ObjectInputStream` as shown below:

```
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.io.IOException;
import java.io.java.lang.ClassNotFoundException;

 public class ReadSerialClass {

    public static void main(String [] args) {
        String filename = "time.ser";
        SerialClass time = null;

        try {
            ObjectInputStream in = new ObjectInputStream(new FileInputStream(filename));
            time = (SerialClass)in.readObject();
            in.close();
        } catch(IOException ex){
            ex.printStackTrace();
        } catch(ClassNotFoundException cnfe){
            cnfe.printStackTrace();
        }
        // print out restored time
        System.out.println("Restored time: " + time.getTime());
     }
}
```

The serialized class is in binary form. The deserialization can be problematic if the class definition changes: see the Versioning of Serialized Objects chapter of the Java Serialization Specification for details.

Serializing an object serializes the entire object graph of which it is the root, and operates correctly in the presence of cyclic graphs. A `reset()` method is provided to force the `ObjectOutputStream` to forget about objects that have already been serialized.

Transient-fields - Serialization

Section 66.2: Custom Serialization

In this example we want to create a class that will generate and output to console, a random number between a

range of two integers which are passed as arguments during the initialization.

```
    public class SimpleRangeRandom implements Runnable {
    private int min;
    private int max;

    private Thread thread;

    public SimpleRangeRandom(int min, int max){
        this.min = min;
        this.max = max;
        thread = new Thread(this);
        thread.start();
    }

    @Override
 private void WriteObject(ObjectOutputStreamout) throws IO Exception;
    private void ReadObject(ObjectInputStream in) throws IOException, ClassNotFoundException;
    public void run() {
        while(true) {
            Random rand = new Random();
            System.out.println("Thread: " + thread.getId() + " Random:" + rand.nextInt(max - min));
            try {
                Thread.sleep(10000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    }
```

Now if we want to make this class Serializable there will be some problems. The Thread is one of the certain system-level classes that are not Serializable. So we need to declare the thread as **transient**. By doing this we will be able to serialize the objects of this class but we will still have an issue. As you can see in the constructor we set the min and the max values of our randomizer and after this we start the thread which is responsible for generating and printing the random value. Thus when restoring the persisted object by calling the ***readObject()*** the constructor will not run again as there is no creation of a new object. In that case we need to develop a **Custom Serialization** by providing two methods inside the class. Those methods are:

```
private void writeObject(ObjectOutputStream out) throws IOException;
private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException;
```

Thus by adding our implementation in the ***readObject()*** we can initiate and start our thread:

```
class RangeRandom implements Serializable, Runnable {

private int min;
private int max;

private transient Thread thread;
//transient should be any field that either cannot be serialized e.g Thread or any field you do not
want serialized

public RangeRandom(int min, int max){
    this.min = min;
    this.max = max;
    thread = new Thread(this);
    thread.start();
}
```

```
@Override
public void run() {
    while(true) {
        Random rand = new Random();
        System.out.println("Thread: " + thread.getId() + " Random:" + rand.nextInt(max - min));
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

private void writeObject(ObjectOutputStream oos) throws IOException {
    oos.defaultWriteObject();
}

private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
    in.defaultReadObject();
    thread = new Thread(this);
    thread.start();
}
}
```

Here is the main for our example:

```
public class Main {
public static void main(String[] args) {
    System.out.println("Hello");
    RangeRandom rangeRandom = new RangeRandom(1,10);

    FileOutputStream fos = null;
    ObjectOutputStream out = null;
    try
    {
        fos = new FileOutputStream("test");
        out = new ObjectOutputStream(fos);
        out.writeObject(rangeRandom);
        out.close();
    }
    catch(IOException ex)
    {
        ex.printStackTrace();
    }


    RangeRandom rangeRandom2 = null;
       FileInputStream fis = null;
       ObjectInputStream in = null;
       try
       {
             fis = new FileInputStream("test");
             in = new ObjectInputStream(fis);
           rangeRandom2 = (RangeRandom)in.readObject();
             in.close();
           }
       catch(IOException ex)
       {
             ex.printStackTrace();
           }
       catch(ClassNotFoundException ex)
```

```
        {
              ex.printStackTrace();
            }

}
}
```

If you run the main you will see that there are two threads running for each RangeRandom instance and that is because the ***Thread.start()*** method is now in both the constructor and the ***readObject()***.

Section 66.3: Versioning and serialVersionUID

When you implement `java.io.Serializable` interface to make a class serializable, the compiler looks for a **`static final`** field named `serialVersionUID` of type **`long`**. If the class doesn't have this field declared explicitly then the compiler will create one such field and assign it with a value which comes out of a implementation dependent computation of `serialVersionUID`. This computation depends upon various aspects of the class and it follows the Object Serialization Specifications given by Sun. But, the value is not guaranteed to be the same across all compiler implementations.

This value is used for checking the compatibility of the classes with respect to serialization and this is done while de-serializing a saved object. The Serialization Runtime verifies that `serialVersionUID` read from the de-serialized data and the `serialVersionUID` declared in the class are exactly the same. If that is not the case, it throws an `InvalidClassException`.

It's highly recommended that you explicitly declare and initialize the static, final field of type long and named 'serialVersionUID' in all your classes you want to make Serializable instead of relying on the default computation of the value for this field even if you are not gonna use versioning. **'serialVersionUID' computation is extremely sensitive and may vary from one compiler implementation to another and hence you may turn up getting the `InvalidClassException` even for the same class just because you used different compiler implementations on the sender and the receiver ends of the serialization process.**

```
public class Example implements Serializable {
    static final long serialVersionUID = 1L /*or some other value*/;
    //...
}
```

As long as `serialVersionUID` is the same, Java Serialization can handle different versions of a class. Compatible and incompatible changes are;

Compatible Changes

- **Adding fields :** When the class being reconstituted has a field that does not occur in the stream, that field in the object will be initialized to the default value for its type. If class-specific initialization is needed, the class may provide a readObject method that can initialize the field to nondefault values.
- **Adding classes :** The stream will contain the type hierarchy of each object in the stream. Comparing this hierarchy in the stream with the current class can detect additional classes. Since there is no information in the stream from which to initialize the object, the class's fields will be initialized to the default values.
- **Removing classes :** Comparing the class hierarchy in the stream with that of the current class can detect that a class has been deleted. In this case, the fields and objects corresponding to that class are read from the stream. Primitive fields are discarded, but the objects referenced by the deleted class are created, since they may be referred to later in the stream. They will be garbage-collected when the stream is garbage-collected or reset.
- **Adding writeObject/readObject methods :** If the version reading the stream has these methods then readObject is expected, as usual, to read the required data written to the stream by the default serialization.

It should call defaultReadObject first before reading any optional data. The writeObject method is expected as usual to call defaultWriteObject to write the required data and then may write optional data.

- **Adding java.io.Serializable :** This is equivalent to adding types. There will be no values in the stream for this class so its fields will be initialized to default values. The support for subclassing nonserializable classes requires that the class's supertype have a no-arg constructor and the class itself will be initialized to default values. If the no-arg constructor is not available, the InvalidClassException is thrown.
- **Changing the access to a field :** The access modifiers public, package, protected, and private have no effect on the ability of serialization to assign values to the fields.
- **Changing a field from static to nonstatic or transient to nontransient :** When relying on default serialization to compute the serializable fields, this change is equivalent to adding a field to the class. The new field will be written to the stream but earlier classes will ignore the value since serialization will not assign values to static or transient fields.

Incompatible Changes

- **Deleting fields :** If a field is deleted in a class, the stream written will not contain its value. When the stream is read by an earlier class, the value of the field will be set to the default value because no value is available in the stream. However, this default value may adversely impair the ability of the earlier version to fulfill its contract.
- **Moving classes up or down the hierarchy :** This cannot be allowed since the data in the stream appears in the wrong sequence.
- **Changing a nonstatic field to static or a nontransient field to transient :** When relying on default serialization, this change is equivalent to deleting a field from the class. This version of the class will not write that data to the stream, so it will not be available to be read by earlier versions of the class. As when deleting a field, the field of the earlier version will be initialized to the default value, which can cause the class to fail in unexpected ways.
- **Changing the declared type of a primitive field :** Each version of the class writes the data with its declared type. Earlier versions of the class attempting to read the field will fail because the type of the data in the stream does not match the type of the field.
- Changing the writeObject or readObject method so that it no longer writes or reads the default field data or changing it so that it attempts to write it or read it when the previous version did not. The default field data must consistently either appear or not appear in the stream.
- Changing a class from Serializable to Externalizable or vice versa is an incompatible change since the stream will contain data that is incompatible with the implementation of the available class.
- Changing a class from a non-enum type to an enum type or vice versa since the stream will contain data that is incompatible with the implementation of the available class.
- Removing either Serializable or Externalizable is an incompatible change since when written it will no longer supply the fields needed by older versions of the class.
- Adding the writeReplace or readResolve method to a class is incompatible if the behavior would produce an object that is incompatible with any older version of the class.

Section 66.4: Serialization with Gson

Serialization with Gson is easy and will output correct JSON.

```
public class Employe {

    private String firstName;
    private String lastName;
    private int age;
    private BigDecimal salary;
    private List<String> skills;
```

```
    //getters and setters
}
```

(Serialization)

```
//Skills
List<String> skills = new LinkedList<String>();
skills.add("leadership");
skills.add("Java Experience");

//Employe
Employe obj = new Employe();
obj.setFirstName("Christian");
obj.setLastName("Lusardi");
obj.setAge(25);
obj.setSalary(new BigDecimal("10000"));
obj.setSkills(skills);

//Serialization process
Gson gson = new Gson();
String json = gson.toJson(obj);
//{"firstName":"Christian","lastName":"Lusardi","age":25,"salary":10000,"skills":["leadership","Java
Experience"]}
```

Note that you can not serialize objects with circular references since that will result in infinite recursion.

(Deserialization)

```
//it's very simple...
//Assuming that json is the previous String object....

Employe obj2 = gson.fromJson(json, Employe.class); // obj2 is just like obj
```

Section 66.5: Custom JSON Deserialization with Jackson

We consume rest API as a JSON format and then unmarshal it to a POJO. Jackson's org.codehaus.jackson.map.ObjectMapper "just works" out of the box and we really don't do anything in most cases. But sometimes we need custom deserializer to fulfill our custom needs and this tutorial will guide you through the process of creating your own custom deserializer.

Let's say we have following entities.

```
public class User {
    private Long id;
    private String name;
    private String email;

    //getter setter are omitted for clarity
}
```

And

```
public class Program {
    private Long id;
    private String name;
    private User createdBy;
    private String contents;
```

```
    //getter setter are omitted for clarity
}
```

Let's serialize/marshal an object first.

```
User user = new User();
user.setId(1L);
user.setEmail("example@example.com");
user.setName("Bazlur Rahman");

Program program = new Program();
program.setId(1L);
program.setName("Program @# 1");
program.setCreatedBy(user);
program.setContents("Some contents");

ObjectMapper objectMapper = new ObjectMapper();
```

final String json = objectMapper.writeValueAsString(program); System.out.println(json);

The above code will produce following JSON-

```
{
    "id": 1,
    "name": "Program @# 1",
    "createdBy": {
        "id": 1,
        "name": "Bazlur Rahman",
        "email": "example@example.com"
    },
    "contents": "Some contents"
}
```

Now can do the opposite very easily. If we have this JSON, we can unmarshal to a program object using ObjectMapper as following –

Now let's say, this is not the real case, we are going to have a different JSON from an API which doesn't match with our `Program` class.

```
{
"id": 1,
"name": "Program @# 1",
"ownerId": 1
"contents": "Some contents"
}
```

Look at the JSON string, you can see, it has a different field that is owenerId.

Now if you want to serialize this JSON as we did earlier, you will have exceptions.

There are two ways to avoid exceptions and have this serialized –

Ignore the unknown fields

Ignore the `onwerId`. Add the following annotation in the Program class

```
@JsonIgnoreProperties(ignoreUnknown = true)
public class Program {}
```

Write custom deserializer

But there are cases when you actually need this `owerId` field. Let's say you want to relate it as an id of the `User` class.

In such case, you need to write a custom deserializer-

As you can see, first you have to access the `JsonNode` from the `JonsParser`. And then you can easily extract information from a `JsonNode` using the `get()` method. and you have to make sure about the field name. It should be the exact name, spelling mistake will cause exceptions.

And finally, you have to register your ProgramDeserializer to the `ObjectMapper`.

```
ObjectMapper mapper = new ObjectMapper();
SimpleModule module = new SimpleModule();
module.addDeserializer(Program.class, new ProgramDeserializer());

mapper.registerModule(module);

String newJsonString = "{\"id\":1,\"name\":\"Program @# 1\",\"ownerId\":1,\"contents\":\"Some
contents\"}";
final Program program2 = mapper.readValue(newJsonString, Program.class);
```

Alternatively, you can use annotation to register the deserializer directly –

```
@JsonDeserialize(using = ProgramDeserializer.class)
public class Program {
}
```

Chapter 67: Optional

`Optional` is a container object which may or may not contain a non-null value. If a value is present, `isPresent()` will return `true` and `get()` will return the value.

Additional methods that depend on the presence of the contained value are provided, such as `orElse()`, which returns a default value if value not present, and `ifPresent()` which executes a block of code if the value is present.

Section 67.1: Map

Use the `map()` method of `Optional` to work with values that might be `null` without doing explicit `null` checks:

(Note that the `map()` and `filter()` operations are evaluated immediately, unlike their Stream counterparts which are only evaluated upon a *terminal operation*.)

Syntax:

```
public <U> Optional<U> map(Function<? super T,? extends U> mapper)
```

Code examples:

```
String value = null;

return Optional.ofNullable(value).map(String::toUpperCase).orElse("NONE");
// returns "NONE"
```

```
String value = "something";

return Optional.ofNullable(value).map(String::toUpperCase).orElse("NONE");
// returns "SOMETHING"
```

Because Optional.map() returns an empty optional when its mapping function returns null, you can chain several map() operations as a form of null-safe dereferencing. This is also known as **Null-safe chaining**.

Consider the following example:

```
String value = foo.getBar().getBaz().toString();
```

Any of `getBar`, `getBaz`, and `toString` can potentially throw a `NullPointerException`.

Here is an alternative way to get the value from `toString()` using `Optional`:

```
String value = Optional.ofNullable(foo)
                       .map(Foo::getBar)
                       .map(Bar::getBaz)
                       .map(Baz::toString)
                       .orElse("");
```

This will return an empty string if any of the mapping functions returned null.

Below is an another example, but slightly different. It will print the value only if none of the mapping functions returned null.

```
Optional.ofNullable(foo)
        .map(Foo::getBar)
        .map(Bar::getBaz)
```

```
        .map(Baz::toString)
        .ifPresent(System.out::println);
```

Section 67.2: Return default value if Optional is empty

Don't just use `Optional.get()` since that may throw `NoSuchElementException`. The `Optional.orElse(T)` and `Optional.orElseGet(Supplier<? extends T>)` methods provide a way to supply a default value in case the Optional is empty.

```
String value = "something";

return Optional.ofNullable(value).orElse("defaultValue");
// returns "something"

return Optional.ofNullable(value).orElseGet(() -> getDefaultValue());
// returns "something" (never calls the getDefaultValue() method)
```

```
String value = null;

return Optional.ofNullable(value).orElse("defaultValue");
// returns "defaultValue"

return Optional.ofNullable(value).orElseGet(() -> getDefaultValue());
// calls getDefaultValue() and returns its results
```

The crucial difference between the `orElse` and `orElseGet` is that the latter is only evaluated when the Optional is empty while the argument supplied to the former one is evaluated even if the Optional is not empty. The `orElse` should therefore only be used for constants and never for supplying value based on any sort of computation.

Section 67.3: Throw an exception, if there is no value

Use the `orElseThrow()` method of `Optional` to get the contained value or throw an exception, if it hasn't been set. This is similar to calling `get()`, except that it allows for arbitrary exception types. The method takes a supplier that must return the exception to be thrown.

In the first example, the method simply returns the contained value:

```
Optional optional = Optional.of("something");

return optional.orElseThrow(IllegalArgumentException::new);
// returns "something" string
```

In the second example, the method throws an exception because a value hasn't been set:

```
Optional optional = Optional.empty();

return optional.orElseThrow(IllegalArgumentException::new);
// throws IllegalArgumentException
```

You can also use the lambda syntax if throwing an exception with message is needed:

```
optional.orElseThrow(() -> new IllegalArgumentException("Illegal"));
```

Section 67.4: Lazily provide a default value using a Supplier

The *normal* `orElse` method takes an `Object`, so you might wonder why there is an option to provide a `Supplier`

here (the `orElseGet` method).

Consider:

```
String value = "something";
return Optional.ofNullable(value)
               .orElse(getValueThatIsHardToCalculate()); // returns "something"
```

It would still call `getValueThatIsHardToCalculate()` even though it's result is not used as the optional is not empty.

To avoid this penalty you supply a supplier:

```
String value = "something";
return Optional.ofNullable(value)
               .orElseGet(() -> getValueThatIsHardToCalculate()); // returns "something"
```

This way `getValueThatIsHardToCalculate()` will only be called if the `Optional` is empty.

Section 67.5: Filter

`filter()` is used to indicate that you would like the value *only if* it matches your predicate.

Think of it like `if (!somePredicate(x)) { x = null; }`.

Code examples:

```
String value = null;
Optional.ofNullable(value) // nothing
        .filter(x -> x.equals("cool string"))// this is never run since value is null
        .isPresent(); // false
```

```
String value = "cool string";
Optional.ofNullable(value) // something
        .filter(x -> x.equals("cool string"))// this is run and passes
        .isPresent(); // true
```

```
String value = "hot string";
Optional.ofNullable(value) // something
        .filter(x -> x.equals("cool string"))// this is run and fails
        .isPresent(); // false
```

Section 67.6: Using Optional containers for primitive number types

`OptionalDouble`, `OptionalInt` and `OptionalLong` work like `Optional`, but are specifically designed to wrap primitive types:

```
OptionalInt presentInt = OptionalInt.of(value);
OptionalInt absentInt = OptionalInt.empty();
```

Because numeric types do have a value, there is no special handling for null. Empty containers can be checked with:

```
presentInt.isPresent(); // Is true.
absentInt.isPresent(); // Is false.
```

Similarly, shorthands exist to aid value management:

```
// Prints the value since it is provided on creation.
presentInt.ifPresent(System.out::println);

// Gives the other value as the original Optional is empty.
int finalValue = absentInt.orElseGet(this::otherValue);

// Will throw a NoSuchElementException.
int nonexistentValue = absentInt.getAsInt();
```

Section 67.7: Run code only if there is a value present

```
Optional<String> optionalWithValue = Optional.of("foo");
optionalWithValue.ifPresent(System.out::println);//Prints "foo".

Optional<String> emptyOptional = Optional.empty();
emptyOptional.ifPresent(System.out::println);//Does nothing.
```

Section 67.8: FlatMap

`flatMap` is similar to `map`. The difference is described by the javadoc as follows:

> This method is similar to `map(Function)`, but the provided mapper is one whose result is already an `Optional`, and if invoked, `flatMap` does not wrap it with an additional `Optional`.

In other words, when you chain a method call that returns an `Optional`, using `Optional.flatMap` avoids creating nested `Optionals`.

For example, given the following classes:

```
public class Foo {
    Optional<Bar> getBar(){
        return Optional.of(new Bar());
    }
}

public class Bar {
}
```

If you use `Optional.map`, you will get a nested `Optional`; i.e. `Optional<Optional<Bar>>`.

```
Optional<Optional<Bar>> nestedOptionalBar =
    Optional.of(new Foo())
        .map(Foo::getBar);
```

However, if you use `Optional.flatMap`, you will get a simple `Optional`; i.e. `Optional<Bar>`.

```
Optional<Bar> optionalBar =
    Optional.of(new Foo())
        .flatMap(Foo::getBar);
```

Chapter 68: Object References

Section 68.1: Object References as method parameters

This topic explains the concept of an *object reference*; it is targeted at people who are new to programming in Java. You should already be familiar with some terms and meanings: class definition, main method, object instance, and the calling of methods "on" an object, and passing parameters to methods.

```
public class Person {

  private String name;

  public void setName(String name) { this.name = name; }

  public String getName() { return name; }

  public static void main(String [] arguments) {
    Person person = new Person();
    person.setName("Bob");

    int i = 5;
    setPersonName(person, i);

    System.out.println(person.getName() + " " + i);
  }

  private static void setPersonName(Person person, int num) {
    person.setName("Linda");
    num = 99;
  }
}
```

To be fully competent in Java programming, you should be able to explain this example to someone else off the top of your head. Its concepts are fundamental to understanding how Java works.

As you can see, we have a `main` that instantiates an object to the variable `person`, and calls a method to set the `name` field in that object to `"Bob"`. Then it calls another method, and passes `person` as one of two parameters; the other parameter is an integer variable, set to 5.

The method called sets the `name` value on the passed object to "Linda', and sets the integer variable passed to 99, then returns.

So what would get printed?

```
Linda 5
```

So why does the change made to `person` take effect in `main`, but the change made to the integer does not?

When the call is made, the main method passes an *object reference* for `person` to the `setPersonName` method; any change that `setAnotherName` makes to that object is part of that object, and so those changes are still part of that object when the method returns.

Another way of saying the same thing: `person` points to an object (stored on the heap, if you're interested). Any change the method makes to that object are made "on that object", and are not affected by whether the method making the change is still active or has returned. When the method returns, any changes made to the object are still stored on that object.

Contrast this with the integer that is passed. Since this is a *primitive* int (and not an Integer object instance), it is passed "by value", meaning its value is provided to the method, not a pointer to the original integer passed in. The method can change it for the method's own purposes, but that does not affect the variable used when the method call is made.

In Java, all primitives are passed by value. Objects are passed by reference, which means that a pointer to the object is passed as the parameter to any methods that take them.

One less-obvious thing this means: it is not possible for a called method to create a *new* object and return it as one of the parameters. The only way for a method to return an object that is created, directly or indirectly, by the method call, is as a return value from the method. Let's first see how that would not work, and then how it would work.

Let's add another method to our little example here:

```
private static void getAnotherObjectNot(Person person) {
  person = new Person();
  person.setName("George");
}
```

And, back in the `main`, below the call to `setAnotherName`, let's put a call to this method and another println call:

```
getAnotherObjectNot(person);
System.out.println(person.getName());
```

Now the program would print out:

```
Linda 5
Linda
```

What happened to the object that had George? Well, the parameter that was passed in was a pointer to Linda; when the `getAnotherObjectNot` method created a new object, it replaced the reference to the Linda object with a reference to the George object. The Linda object still exists (on the heap), the `main` method can still access it, but the `getAnotherObjectNot` method wouldn't be able to do anything with it after that, because it has no reference to it. It would appear that the writer of the code intended for the method to create a new object and pass it back, but if so, it didn't work.

If that is what the writer wanted to do, he would need to return the newly created object from the method, something like this:

```
private static Person getAnotherObject() {
  Person person = new Person();
  person.setName("Mary");
  return person;
}
```

Then call it like this:

```
Person mary;
mary = getAnotherObject();
System.out.println(mary.getName());
```

And the entire program output would now be:

```
Linda 5
```

```
Linda
Mary
```

Here is the entire program, with both additions:

```
public class Person {
  private String name;

  public void setName(String name) { this.name = name; }
  public String getName() { return name; }

  public static void main(String [] arguments) {
    Person person = new Person();
    person.setName("Bob");

    int i = 5;
    setPersonName(person, i);
    System.out.println(person.getName() + " " + i);

    getAnotherObjectNot(person);
    System.out.println(person.getName());

    Person person;
    person = getAnotherObject();
    System.out.println(person.getName());
  }

  private static void setPersonName(Person person, int num) {
    person.setName("Linda");
    num = 99;
  }

  private static void getAnotherObjectNot(Person person) {
    person = new Person();
    person.setMyName("George");
  }

  private static person getAnotherObject() {
    Person person = new Person();
    person.setMyName("Mary");
    return person;
  }
}
```

Chapter 69: Exceptions and exception handling

Objects of type `Throwable` and its subtypes can be sent up the stack with the **`throw`** keyword and caught with **`try`**...**`catch`** statements.

Section 69.1: Catching an exception with try-catch

An exception can be caught and handled using the **`try...catch`** statement. (In fact **`try`** statements take other forms, as described in other examples about **`try...catch...finally`** and `try-with-resources`.)

Try-catch with one catch block

The most simple form looks like this:

```
try {
    doSomething();
} catch (SomeException e) {
    handle(e);
}
// next statement
```

The behavior of a simple **`try...catch`** is as follows:

- The statements in the **`try`** block are executed.
- If no exception is thrown by the statements in the **`try`** block, then control passes to the next statement after the **`try...catch`**.
- If an exception is thrown within the **`try`** block.
    - The exception object is tested to see if it is an instance of `SomeException` or a subtype.
    - If it is, then the **`catch`** block will *catch* the exception:
        - The variable `e` is bound to the exception object.
        - The code within the **`catch`** block is executed.
        - If that code throws an exception, then the newly thrown exception is propagated in place of the original one.
        - Otherwise, control passes to the next statement after the **`try...catch`**.
    - If it is not, the original exception continues to propagate.

Try-catch with multiple catches

A **`try...catch`** can also have multiple **`catch`** blocks. For example:

```
try {
    doSomething();
} catch (SomeException e) {
    handleOneWay(e)
} catch (SomeOtherException e) {
    handleAnotherWay(e);
}
// next statement
```

If there are multiple **`catch`** blocks, they are tried one at a time starting with the first one, until a match is found for the exception. The corresponding handler is executed (as above), and then control is passed to the next statement after the **`try...catch`** statement. The **`catch`** blocks after the one that matches are always skipped, *even if the handler code throws an exception*.

The "top down" matching strategy has consequences for cases where the exceptions in the `catch` blocks are not disjoint. For example:

```
try {
    throw new RuntimeException("test");
} catch (Exception e) {
    System.out.println("Exception");
} catch (RuntimeException e) {
    System.out.println("RuntimeException");
}
```

This code snippet will output "Exception" rather than "RuntimeException". Since `RuntimeException` is a subtype of `Exception`, the first (more general) `catch` will be matched. The second (more specific) `catch` will never be executed.

The lesson to learn from this is that the most specific `catch` blocks (in terms of the exception types) should appear first, and the most general ones should be last. (Some Java compilers will warn you if a `catch` can never be executed, but this is not a compilation error.)

Multi-exception catch blocks

Version ≥ Java SE 7

Starting with Java SE 7, a single `catch` block can handle a list of unrelated exceptions. The exception type are listed, separated with a vertical bar (`|`) symbol. For example:

```
try {
    doSomething();
} catch (SomeException | SomeOtherException e) {
    handleSomeException(e);
}
```

The behavior of a multi-exception catch is a simple extension for the single-exception case. The `catch` matches if the thrown exception matches (at least) one of the listed exceptions.

There is some additional subtlety in the specification. The type of `e` is a synthetic *union* of the exception types in the list. When the value of `e` is used, its static type is the least common supertype of the type union. However, if `e` is rethrown within the `catch` block, the exception types that are thrown are the types in the union. For example:

```
public void method() throws IOException, SQLException
    try {
        doSomething();
    } catch (IOException | SQLException e) {
        report(e);
        throw e;
    }
```

In the above, `IOException` and `SQLException` are checked exceptions whose least common supertype is `Exception`. This means that the `report` method must match `report(Exception)`. However, the compiler knows that the `throw` can throw only an `IOException` or an `SQLException`. Thus, `method` can be declared as `throws IOException, SQLException` rather than `throws Exception`. (Which is a good thing: see Pitfall - Throwing Throwable, Exception, Error or RuntimeException.)

Section 69.2: The try-with-resources statement

Version ≥ Java SE 7

As the try-catch-final statement example illustrates, resource cleanup using a **`finally`** clause requires a significant amount of "boiler-plate" code to implement the edge-cases correctly. Java 7 provides a much simpler way to deal with this problem in the form of the *try-with-resources* statement.

What is a resource?

Java 7 introduced the `java.lang.AutoCloseable` interface to allow classes to be managed using the *try-with-resources* statement. Instances of classes that implement `AutoCloseable` are referred to as *resources*. These typically need to be disposed of in a timely fashion rather than relying on the garbage collector to dispose of them.

The `AutoCloseable` interface defines a single method:

```
public void close() throws Exception
```

A `close()` method should dispose of the resource in an appropriate fashion. The specification states that it should be safe to call the method on a resource that has already been disposed of. In addition, classes that implement `Autocloseable` are *strongly encouraged* to declare the `close()` method to throw a more specific exception than `Exception`, or no exception at all.

A wide range of standard Java classes and interfaces implement `AutoCloseable`. These include:

- `InputStream`, `OutputStream` and their subclasses
- `Reader`, `Writer` and their subclasses
- `Socket` and `ServerSocket` and their subclasses
- `Channel` and its subclasses, and
- the JDBC interfaces `Connection`, `Statement` and `ResultSet` and their subclasses.

Application and third party classes may do this as well.

The basic try-with-resource statement

The syntax of a *try-with-resources* is based on classical *try-catch*, *try-finally* and *try-catch-finally* forms. Here is an example of a "basic" form; i.e. the form without a **`catch`** or **`finally`**.

```
try (PrintStream stream = new PrintStream("hello.txt")) {
    stream.println("Hello world!");
}
```

The resources to be manage are declared as variables in the `(...)` section after the **`try`** clause. In the example above, we declare a resource variable `stream` and initialize it to a newly created `PrintStream`.

Once the resource variables have been initialized, the **`try`** block is executed. When that completes, `stream.close()` will be called automatically to ensure that the resource does not leak. Note that the `close()` call happens no matter how the block completes.

The enhanced try-with-resource statements

The *try-with-resources* statement can be enhanced with **`catch`** and **`finally`** blocks, as with the pre-Java 7 *try-catch-finally* syntax. The following code snippet adds a **`catch`** block to our previous one to deal with the `FileNotFoundException` that the `PrintStream` constructor can throw:

```
try (PrintStream stream = new PrintStream("hello.txt")) {
    stream.println("Hello world!");
} catch (FileNotFoundException ex) {
    System.err.println("Cannot open the file");
```

```
} finally {
    System.err.println("All done");
}
```

If either the resource initialization or the try block throws the exception, then the `catch` block will be executed. The `finally` block will always be executed, as with a conventional *try-catch-finally* statement.

There are a couple of things to note though:

- The resource variable is *out of scope* in the `catch` and `finally` blocks.
- The resource cleanup will happen before the statement tries to match the `catch` block.
- If the automatic resource cleanup threw an exception, then that *could* be caught in one of the `catch` blocks.

Managing multiple resources

The code snippets above show a single resource being managed. In fact, *try-with-resources* can manage multiple resources in one statement. For example:

```
try (InputStream is = new FileInputStream(file1);
     OutputStream os = new FileOutputStream(file2)) {
    // Copy 'is' to 'os'
}
```

This behaves as you would expect. Both `is` and `os` are closed automatically at the end of the `try` block. There are a couple of points to note:

- The initializations occur in the code order, and later resource variable initializers can use of the values of the earlier ones.
- All resource variables that were successfully initialized will be cleaned up.
- Resource variables are cleaned up in *reverse order* of their declarations.

Thus, in the above example, `is` is initialized before `os` and cleaned up after it, and `is` will be cleaned up if there is an exception while initializing `os`.

Equivalence of try-with-resource and classical try-catch-finally

The Java Language Specification specifies the behavior of *try-with-resource* forms in terms of the classical *try-catch-finally* statement. (Please refer to the JLS for the full details.)

For example, this basic *try-with-resource* :

```
try (PrintStream stream = new PrintStream("hello.txt")) {
    stream.println("Hello world!");
}
```

is defined to be equivalent to this *try-catch-finally*:

```
// Note that the constructor is not part of the try-catch statement
PrintStream stream = new PrintStream("hello.txt");

// This variable is used to keep track of the primary exception thrown
// in the try statement. If an exception is thrown in the try block,
// any exception thrown by AutoCloseable.close() will be suppressed.
Throwable primaryException = null;

// The actual try block
try {
```

```
    stream.println("Hello world!");
} catch (Throwable t) {
    // If an exception is thrown, remember it for the finally block
    primaryException = t;
    throw t;
} finally {
    if (primaryException == null) {
        // If no exception was thrown so far, exceptions thrown in close() will
        // not be caught and therefore be passed on to the enclosing code.
        stream.close();
    } else {
        // If an exception has already been thrown, any exception thrown in
        // close() will be suppressed as it is likely to be related to the
        // previous exception. The suppressed exception can be retrieved
        // using primaryException.getSuppressed().
        try {
            stream.close();
        } catch (Throwable suppressedException) {
            primaryException.addSuppressed(suppressedException);
        }
    }
}
```

(The JLS specifies that the actual `t` and `primaryException` variables will be invisible to normal Java code.)

The enhanced form of *try-with-resources* is specified as an equivalence with the basic form. For example:

```
try (PrintStream stream = new PrintStream(fileName)) {
    stream.println("Hello world!");
} catch (NullPointerException ex) {
    System.err.println("Null filename");
} finally {
    System.err.println("All done");
}
```

is equivalent to:

```
try {
    try (PrintStream stream = new PrintStream(fileName)) {
        stream.println("Hello world!");
    }
} catch (NullPointerException ex) {
    System.err.println("Null filename");
} finally {
    System.err.println("All done");
}
```

Section 69.3: Custom Exceptions

Under most circumstances, it is simpler from a code-design standpoint to use existing generic `Exception` classes when throwing exceptions. This is especially true if you only need the exception to carry a simple error message. In that case, RuntimeException is usually preferred, since it is not a checked Exception. Other exception classes exist for common classes of errors:

- UnsupportedOperationException - a certain operation is not supported
- IllegalArgumentException - an invalid parameter value was passed to a method
- IllegalStateException - your API has internally reached a condition that should never happen, or which occurs as a result of using your API in an invalid way

Cases where you **do** want to use a custom exception class include the following:

- You are writing an API or library for use by others, and you want to allow users of your API to be able to specifically catch and handle exceptions from your API, and *be able to differentiate those exceptions from other, more generic exceptions*.
- You are throwing exceptions for a **specific kind of error** in one part of your program, which you want to catch and handle in another part of your program, and you want to be able to differentiate these errors from other, more generic errors.

You can create your own custom exceptions by extending `RuntimeException` for an unchecked exception, or checked exception by extending any `Exception` *which is not also subclass of RuntimeException*, because:

> Subclasses of Exception that are not also subclasses of RuntimeException are checked exceptions

```
public class StringTooLongException extends RuntimeException {
    // Exceptions can have methods and fields like other classes
    // those can be useful to communicate information to pieces of code catching
    // such an exception
    public final String value;
    public final int maximumLength;

    public StringTooLongException(String value, int maximumLength){
        super(String.format("String exceeds maximum Length of %s: %s", maximumLength, value));
        this.value = value;
        this.maximumLength = maximumLength;
    }
}
```

Those can be used just as predefined exceptions:

```
void validateString(String value){
    if (value.length() > 30){
        throw new StringTooLongException(value, 30);
    }
}
```

And the fields can be used where the exception is caught and handled:

```
void anotherMethod(String value){
    try {
        validateString(value);
    } catch(StringTooLongException e){
        System.out.println("The string '" + e.value +
                "' was longer than the max of " + e.maximumLength );
    }
}
```

Keep in mind that, according to Oracle's Java Documentation:

> [...] If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.

More:

- Why does RuntimeException not require an explicit exception handling?

Section 69.4: Handling InterruptedException

`InterruptedException` is a confusing beast - it shows up in seemingly innocuous methods like `Thread.sleep()`, but handling it incorrectly leads to hard-to-manage code that behaves poorly in concurrent environments.

At its most basic, if an `InterruptedException` is caught it means someone, somewhere, called `Thread.interrupt()` on the thread your code is currently running in. You might be inclined to say "It's my code! I'll never interrupt it!" and therefore do something like this:

```
// Bad. Don't do this.
try {
  Thread.sleep(1000);
} catch (InterruptedException e) {
  // disregard
}
```

But this is exactly the wrong way to handle an "impossible" event occurring. If you know your application will never encounter an `InterruptedException` you should treat such an event as a serious violation of your program's assumptions and exit as quickly as possible.

The proper way to handle an "impossible" interrupt is like so:

```
// When nothing will interrupt your code
try {
  Thread.sleep(1000);
} catch (InterruptedException e) {
  Thread.currentThread().interrupt();
  throw new AssertionError(e);
}
```

This does two things; it first restores the interrupt status of the thread (as if the `InterruptedException` had not been thrown in the first place), and then it throws an `AssertionError` indicating the basic invariants of your application have been violated. If you know for certain that you'll never interrupt the thread this code runs in this is safe since the **`catch`** block should never be reached.

Using Guava's `Uninterruptibles` class helps simplify this pattern; calling `Uninterruptibles.sleepUninterruptibly()` disregards the interrupted state of a thread until the sleep duration has expired (at which point it's restored for later calls to inspect and throw their own `InterruptedException`). If you know you'll never interrupt such code this safely avoids needing to wrap your sleep calls in a try-catch block.

More often, however, you cannot guarantee that your thread will never be interrupted. In particular if you're writing code that will be executed by an `Executor` or some other thread-management it's critical that your code responds promptly to interrupts, otherwise your application will stall or even deadlock.

In such cases the best thing to do is generally to allow the `InterruptedException` to propagate up the call stack, adding a **`throws`** `InterruptedException` to each method in turn. This may seem kludgy but it's actually a desirable property - your method's signatures now indicates to callers that it will respond promptly to interrupts.

```
// Let the caller determine how to handle the interrupt if you're unsure
public void myLongRunningMethod() throws InterruptedException {
  ...
}
```

In limited cases (e.g. while overriding a method that doesn't `throw` any checked exceptions) you can reset the interrupted status without raising an exception, expecting whatever code is executed next to handle the interrupt. This delays handling the interruption but doesn't suppress it entirely.

```
// Suppresses the exception but resets the interrupted state letting later code
// detect the interrupt and handle it properly.
try {
  Thread.sleep(1000);
} catch (InterruptedException e) {
  Thread.currentThread().interrupt();
  return ...; // your expectations are still broken at this point - try not to do more work.
}
```

Section 69.5: Return statements in try catch block

Although it's bad practice, it's possible to add multiple return statements in a exception handling block:

```
public static int returnTest(int number){
   try{
       if(number%2 == 0) throw new Exception("Exception thrown");
       else return x;
   }
   catch(Exception e){
       return 3;
   }
   finally{
       return 7;
   }
}
```

This method will always return 7 since the finally block associated with the try/catch block is executed before anything is returned. Now, as finally has `return 7;`, this value supersedes the try/catch return values.

If the catch block returns a primitive value and that primitive value is subsequently changed in the finally block, the value returned in the catch block will be returned and the changes from the finally block will be ignored.

The example below will print "0", not "1".

```
public class FinallyExample {

    public static void main(String[] args) {
        int n = returnTest(4);

        System.out.println(n);
    }

    public static int returnTest(int number) {

        int returnNumber = 0;

        try {
            if (number % 2 == 0)
                throw new Exception("Exception thrown");
            else
                return returnNumber;
        } catch (Exception e) {
            return returnNumber;
        } finally {
            returnNumber = 1;
```

```
        }
    }
}
```

Section 69.6: Introduction

Exceptions are errors which occur when a program is executing. Consider the Java program below which divides two integers.

```
class Division {
    public static void main(String[] args) {

        int a, b, result;

        Scanner input = new Scanner(System.in);
        System.out.println("Input two integers");

        a = input.nextInt();
        b = input.nextInt();

        result = a / b;

        System.out.println("Result = " + result);
    }
}
```

Now we compile and execute the above code, and see the output for an attempted division by zero:

```
Input two integers
7 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
at Division.main(Disivion.java:14)
```

Division by zero is an invalid operation that would produce a value that cannot be represented as an integer. Java deals with this by *throwing an **exception***. In this case, the exception is an instance of the *ArithmeticException* class.

Note: The example on creating and reading stack traces explains what the output after the two numbers means.

The utility of an *exception* is the flow control that it allows. Without using exceptions, a typical solution to this problem may be to first check if `b == 0`:

```
class Division {
    public static void main(String[] args) {

        int a, b, result;

        Scanner input = new Scanner(System.in);
        System.out.println("Input two integers");

        a = input.nextInt();
        b = input.nextInt();

        if (b == 0) {
            System.out.println("You cannot divide by zero.");
            return;
        }
```

```
        result = a / b;

        System.out.println("Result = " + result);
    }
}
```

This prints the message `You cannot divide by zero.` to the console and quits the program in a graceful way when the user tries to divide by zero. An equivalent way of dealing with this problem via *exception handling* would be to replace the `if` flow control with a `try-catch` block:

```
...

a = input.nextInt();
b = input.nextInt();

try {
    result = a / b;
}
catch (ArithmeticException e) {
    System.out.println("An ArithmeticException occurred. Perhaps you tried to divide by zero.");
    return;
}

...
```

A try catch block is executed as follows:

1. Begin executing the code in the **`try`** block.
2. If an *exception* occurs in the try block, immediately abort and check to see if this exception is *caught* by the **`catch`** block (in this case, when the Exception is an instance of `ArithmeticException`).
3. If the exception is *caught*, it is assigned to the variable `e` and the **`catch`** block is executed.
4. If either the **`try`** or **`catch`** block is completed (i.e. no uncaught exceptions occur during code execution) then continue to execute code below the `try-catch` block.

It is generally considered good practice to use *exception handling* as part of the normal flow control of an application where behavior would otherwise be undefined or unexpected. For instance, instead of returning **`null`** when a method fails, it is usually better practice to *throw an exception* so that the application making use of the method can define its own flow control for the situation via *exception handling* of the kind illustrated above. In some sense, this gets around the problem of having to return a particular *type*, as any one of multiple kinds of *exceptions* may be *thrown* to indicate the specific problem that occurred.

For more advice on how and how not to use exceptions, refer to Java Pitfalls - Exception usage

Section 69.7: The Java Exception Hierarchy - Unchecked and Checked Exceptions

All Java exceptions are instances of classes in the Exception class hierarchy. This can be represented as follows:

- `java.lang.Throwable` - This is the base class for all exception classes. Its methods and constructors implement a range of functionality common to all exceptions.
    - `java.lang.Exception` - This is the superclass of all normal exceptions.
        - various standard and custom exception classes.
        - `java.lang.RuntimeException` - This the superclass of all normal exceptions that are *unchecked exceptions*.
            - various standard and custom runtime exception classes.

- `java.lang.Error` - This is the superclass of all "fatal error" exceptions.

Notes:

1. The distinction between *checked* and *unchecked* exceptions is described below.
2. The `Throwable`, `Exception` and `RuntimeException` class should be treated as **`abstract`**; see Pitfall - Throwing Throwable, Exception, Error or RuntimeException.
3. The `Error` exceptions are thrown by the JVM in situations where it would be unsafe or unwise for an application to attempt to recover.
4. It would be unwise to declare custom subtypes of `Throwable`. Java tools and libraries may assume that `Error` and `Exception` are the only direct subtypes of `Throwable`, and misbehave if that assumption is incorrect.

Checked versus Unchecked Exceptions

One of the criticisms of exception support in some programming languages is that is difficult to know which exceptions a given method or procedure might throw. Given that an unhandled exception is liable to cause a program to crash, this can make exceptions a source of fragility.

The Java language addresses this concern with the checked exception mechanism. First, Java classifies exceptions into two categories:

- Checked exceptions typically represent anticipated events that an application should be able to deal with. For instance, `IOException` and its subtypes represent error conditions that can occur in I/O operations. Examples include, file opens failing because a file or directory does not exist, network reads and writes failing because a network connection has been broken and so on.
- Unchecked exceptions typically represent unanticipated events that an application cannot deal with. These are typically the result of a bug in the application.

(In the following, "thrown" refers to any exception thrown explicitly (by a **`throw`** statement), or implicitly (in a failed dereference, type cast and so on). Similarly, "propagated" refers to an exception that was thrown in a nested call, and not caught within that call. The sample code below will illustrate this.)

The second part of the checked exception mechanism is that there are restrictions on methods where a checked exception may occur:

- When a checked exception is thrown or propagated in a method, it *must* either be caught by the method, or listed in the method's **`throws`** clause. (The significance of the **`throws`** clause is described in this example.)
- When a checked exception is thrown or propagated in an initializer block, it must be caught the the block.
- A checked exception cannot be propagated by a method call in a field initialization expression. (There is no way to catch such an exception.)

In short, a checked exception must be either handled, or declared.

These restrictions do not apply to unchecked exceptions. This includes all cases where an exception is thrown implicitly, since all such cases throw unchecked exceptions.

Checked exception examples

These code snippets are intended to illustrate the checked exception restrictions. In each case, we show a version of the code with a compilation error, and a second version with the error corrected.

```
// This declares a custom checked exception.
public class MyException extends Exception {
    // constructors omitted.
```

```
}

// This declares a custom unchecked exception.
public class MyException2 extends RuntimeException {
    // constructors omitted.
}
```

The first example shows how explicitly thrown checked exceptions can be declared as "thrown" if they should not be handled in the method.

```
// INCORRECT
public void methodThrowingCheckedException(boolean flag) {
    int i = 1 / 0;               // Compiles OK, throws ArithmeticException
    if (flag) {
        throw new MyException();  // Compilation error
    } else {
        throw new MyException2(); // Compiles OK
    }
}

// CORRECTED
public void methodThrowingCheckedException(boolean flag) throws MyException {
    int i = 1 / 0;               // Compiles OK, throws ArithmeticException
    if (flag) {
        throw new MyException();  // Compilation error
    } else {
        throw new MyException2(); // Compiles OK
    }
}
```

The second example shows how a propagated checked exception can be dealt with.

```
// INCORRECT
public void methodWithPropagatedCheckedException() {
    InputStream is = new FileInputStream("someFile.txt");  // Compilation error
    // FileInputStream throws IOException or a subclass if the file cannot
    // be opened.  IOException is a checked exception.
    ...
}

// CORRECTED (Version A)
public void methodWithPropagatedCheckedException() throws IOException {
    InputStream is = new FileInputStream("someFile.txt");
    ...
}

// CORRECTED (Version B)
public void methodWithPropagatedCheckedException() {
    try {
        InputStream is = new FileInputStream("someFile.txt");
        ...
    } catch (IOException ex) {
        System.out.println("Cannot open file: " + ex.getMessage());
    }
}
```

The final example shows how to deal with a checked exception in a static field initializer.

```
// INCORRECT
public class Test {
```

```
    private static final InputStream is =
            new FileInputStream("someFile.txt");  // Compilation error
}

// CORRECTED
public class Test {
    private static final InputStream is;
    static {
        InputStream tmp = null;
        try {
            tmp = new FileInputStream("someFile.txt");
        } catch (IOException ex) {
            System.out.println("Cannot open file: " + ex.getMessage());
        }
        is = tmp;
    }
}
```

Note that in this last case, we also have to deal with the problems that `is` cannot be assigned to more than once, and yet also has to be assigned to, even in the case of an exception.

Section 69.8: Creating and reading stacktraces

When an exception object is created (i.e. when you **`new`** it), the `Throwable` constructor captures information about the context in which the exception was created. Later on, this information can be output in the form of a stacktrace, which can be used to help diagnose the problem that caused the exception in the first place.

Printing a stacktrace

Printing a stacktrace is simply a matter of calling the `printStackTrace()` method. For example:

```
try {
    int a = 0;
    int b = 0;
    int c = a / b;
} catch (ArithmeticException ex) {
    // This prints the stacktrace to standard output
    ex.printStackTrace();
}
```

The `printStackTrace()` method without arguments will print to the application's standard output; i.e. the current `System.out`. There are also `printStackTrace(PrintStream)` and `printStackTrace(PrintWriter)` overloads that print to a specified `Stream` or `Writer`.

Notes:

1. The stacktrace does not include the details of the exception itself. You can use the `toString()` method to get those details; e.g.

   ```
   // Print exception and stacktrace
   System.out.println(ex);
   ex.printStackTrace();
   ```

2. Stacktrace printing should be used sparingly; see Pitfall - Excessive or inappropriate stacktraces . It is often better to use a logging framework, and pass the exception object to be logged.

Understanding a stacktrace

Consider the following simple program consisting of two classes in two files. (We have shown the filenames and added line numbers for illustration purposes.)

```
File: "Main.java"
1   public class Main {
2       public static void main(String[] args) {
3           new Test().foo();
4       }
5   }

File: "Test.java"
1   class Test {
2       public void foo() {
3           bar();
4       }
5
6       public int bar() {
7           int a = 1;
8           int b = 0;
9           return a / b;
10      }
```

When these files are compiled and run, we will get the following output.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at Test.bar(Test.java:9)
at Test.foo(Test.java:3)
at Main.main(Main.java:3)
```

Let us read this one line at a time to figure out what it is telling us.

Line #1 tells us that the thread called "main" has terminated due to an uncaught exception. The full name of the exception is `java.lang.ArithmeticException`, and the exception message is "/ by zero".

If we look up the javadocs for this exception, it says:

> Thrown when an exceptional arithmetic condition has occurred. For example, an integer "divide by zero" throws an instance of this class.

Indeed, the message "/ by zero" is a strong hint that the cause of the exception is that some code has attempted to divide something by zero. But what?

The remaining 3 lines are the stack trace. Each line represents a method (or constructor) call on the call stack, and each one tells us three things:

- the name of the class and method that was being executed,
- the source code filename,
- the source code line number of the statement that was being executed

These lines of a stacktrace are listed with the frame for the current call at the top. The top frame in our example above is in the `Test.bar` method, and at line 9 of the Test.java file. That is the following line:

```
    return a / b;
```

If we look a couple of lines earlier in the file to where `b` is initialized, it is apparent that `b` will have the value zero. We can say without any doubt that this is the cause of the exception.

If we needed to go further, we can see from the stacktrace that `bar()` was called from `foo()` at line 3 of Test.java, and that `foo()` was in turn called from `Main.main()`.

Note: The class and method names in the stack frames are the internal names for the classes and methods. You will need to recognize the following unusual cases:

- A nested or inner class will look like "OuterClass$InnerClass".
- An anonymous inner class will look like "OuterClass$1", "OuterClass$2", etcetera.
- When code in a constructor, instance field initializer or an instance initializer block is being executed, the method name will be "".
- When code in a static field initializer or static initializer block is being executed, the method name will be "".

(In some versions of Java, the stacktrace formatting code will detect and elide repeated stackframe sequences, as can occur when an application fails due to excessive recursion.)

Exception chaining and nested stacktraces

`Version ≥ Java SE 1.4`

Exception chaining happens when a piece of code catches an exception, and then creates and throws a new one, passing the first exception as the cause. Here is an example:

```
File: Test,java
1   public class Test {
2      int foo() {
3           return 0 / 0;
4      }
5
6       public Test() {
7           try {
8               foo();
9           } catch (ArithmeticException ex) {
10              throw new RuntimeException("A bad thing happened", ex);
11          }
12      }
13
14      public static void main(String[] args) {
15          new Test();
16      }
17  }
```

When the above class is compiled and run, we get the following stacktrace:

```
Exception in thread "main" java.lang.RuntimeException: A bad thing happened
        at Test.<init>(Test.java:10)
        at Test.main(Test.java:15)
Caused by: java.lang.ArithmeticException: / by zero
        at Test.foo(Test.java:3)
        at Test.<init>(Test.java:8)
        ... 1 more
```

The stacktrace starts with the class name, method and call stack for the exception that (in this case) caused the application to crash. This is followed by a "Caused by:" line that reports the `cause` exception. The class name and message are reported, followed by the cause exception's stack frames. The trace ends with an "... N more" which indicates that the last N frames are the same as for the previous exception.

The "Caused by:" is only included in the output when the primary exception's `cause` is not **`null`**). Exceptions can be chained indefinitely, and in that case the stacktrace can have multiple "Caused by:" traces.

Note: the `cause` mechanism was only exposed in the `Throwable` API in Java 1.4.0. Prior to that, exception chaining needed to be implemented by the application using a custom exception field to represent the cause, and a custom `printStackTrace` method.

Capturing a stacktrace as a String

Sometimes, an application needs to be able to capture a stacktrace as a Java `String`, so that it can be used for other purposes. The general approach for doing this is to create a temporary `OutputStream` or `Writer` that writes to an in-memory buffer and pass that to the `printStackTrace(...)`.

The Apache Commons and Guava libraries provide utility methods for capturing a stacktrace as a String:

```
org.apache.commons.lang.exception.ExceptionUtils.getStackTrace(Throwable)

com.google.common.base.Throwables.getStackTraceAsString(Throwable)
```

If you cannot use third party libraries in your code base, then the following method with do the task:

```
   /**
     * Returns the string representation of the stack trace.
     *
     * @param throwable the throwable
     * @return the string.
     */
    public static String stackTraceToString(Throwable throwable) {
        StringWriter stringWriter = new StringWriter();
        throwable.printStackTrace(new PrintWriter(stringWriter));
        return stringWriter.toString();
    }
```

Note that if your intention is to analyze the stacktrace, it is simpler to use `getStackTrace()` and `getCause()` than to attempt to parse a stacktrace.

Section 69.9: Throwing an exception

The following example shows the basics of throwing an exception:

```
public void checkNumber(int number) throws IllegalArgumentException {
    if (number < 0) {
        throw new IllegalArgumentException("Number must be positive: " + number);
    }
}
```

The exception is thrown on the 3rd line. This statement can be broken down into two parts:

- **`new`** `IllegalArgumentException(...)` is creating an instance of the `IllegalArgumentException` class, with a message that describes the error that exception is reporting.
- **`throw`** `...` is then throwing the exception object.

When the exception is thrown, it causes the enclosing statements to *terminate abnormally* until the exception is *handled*. This is described in other examples.

It is good practice to both create and throw the exception object in a single statement, as shown above. It is also good practice to include a meaningful error message in the exception to help the programmer to understand the cause of the problem. However, this is not necessarily the message that you should be showing to the end user. (For a start, Java has no direct support for internationalizing exception messages.)

There are a couple more points to be made:

- We have declared the `checkNumber` as **`throws`** `IllegalArgumentException`. This was not strictly necessary, since `IllegalArgumentException` is a checked exception; see The Java Exception Hierarchy - Unchecked and Checked Exceptions. However, it is good practice to do this, and also to include the exceptions thrown a method's javadoc comments.
- Code immediately after a **`throw`** statement is *unreachable*. Hence if we wrote this:

  ```
  throw new IllegalArgumentException("it is bad");
  return;
  ```

  the compiler would report a compilation error for the **`return`** statement.

Exception chaining

Many standard exceptions have a constructor with a second `cause` argument in addition to the conventional `message` argument. The `cause` allows you to chain exceptions. Here is an example.

First we define an unchecked exception that our application is going throw when it encounters a non-recoverable error. Note that we have included a constructor that accepts a `cause` argument.

```
public class AppErrorException extends RuntimeException {
    public AppErrorException() {
        super();
    }

    public AppErrorException(String message) {
        super(message);
    }

    public AppErrorException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Next, here is some code that illustrates exception chaining.

```
public String readFirstLine(String file) throws AppErrorException {
    try (Reader r = new BufferedReader(new FileReader(file))) {
        String line = r.readLine();
        if (line != null) {
            return line;
        } else {
            throw new AppErrorException("File is empty: " + file);
        }
    } catch (IOException ex) {
        throw new AppErrorException("Cannot read file: " + file, ex);
    }
}
```

The `throw` within the `try` block detects a problem and reports it via an exception with a simple message. By contrast, the `throw` within the `catch` block is handling the `IOException` by wrapping it in a new (checked) exception. However, it is not throwing away the original exception. By passing the `IOException` as the `cause`, we record it so that it can be printed in the stacktrace, as explained in Creating and reading stacktraces.

Section 69.10: Advanced features of Exceptions

This example covers some advanced features and use-cases for Exceptions.

Examining the callstack programmatically

Version ≥ Java SE 1.4

The primary use of exception stacktraces is to provide information about an application error and its context so that the programmer can diagnose and fix the problem. Sometimes it can be used for other things. For example, a `SecurityManager` class may need to examine the call stack to decide whether the code that is making a call should be trusted.

You can use exceptions to examine the call stack programmatically as follows:

```
    Exception ex = new Exception();   // this captures the call stack
    StackTraceElement[] frames = ex.getStackTrace();
    System.out.println("This method is " + frames[0].getMethodName());
    System.out.println("Called from method " + frames[1].getMethodName());
```

There are some important caveats on this:

1. The information available in a `StackTraceElement` is limited. There is no more information available than is displayed by `printStackTrace`. (The values of the local variables in the frame are not available.)
2. The javadocs for `getStackTrace()` state that a JVM is permitted to leave out frames:

> Some virtual machines may, under some circumstances, omit one or more stack frames from the stack trace. In the extreme case, a virtual machine that has no stack trace information concerning this throwable is permitted to return a zero-length array from this method.

Optimizing exception construction

As mentioned elsewhere, constructing an exception is rather expensive because it entails capturing and recording information about all stack frames on the current thread. Sometimes, we know that that information is never going to be used for a given exception; e.g. the stacktrace will never be printed. In that case, there is an implementation trick that we can use in a custom exception to cause the information to not be captured.

The stack frame information needed for stacktraces, is captured when the `Throwable` constructors call the `Throwable.fillInStackTrace()` method. This method is `public`, which means that a subclass can override it. The trick is to override the method inherited from `Throwable` with one that does nothing; e.g.

```
public class MyException extends Exception {
    // constructors

    @Override
    public void fillInStackTrace() {
        // do nothing
    }
```

```
}
```

The problem with this approach is that an exception that overrides `fillInStackTrace()` can never capture the stacktrace, and is useless in scenarios where you need one.

Erasing or replacing the stacktrace

`Version ≥ Java SE 1.4`

In some situations, the stacktrace for an exception created in the normal way contains either incorrect information, or information that the developer does not want to reveal to the user. For these scenarios, the `Throwable.setStackTrace` can be used to replace the array of `StackTraceElement` objects that holds the information.

For example, the following can be used to discard an exception's stack information:

```
exception.setStackTrace(new StackTraceElement[0]);
```

Suppressed exceptions

`Version ≥ Java SE 7`

Java 7 introduced the *try-with-resources* construct, and the associated concept of exception suppression. Consider the following snippet:

```
try (Writer w = new BufferedWriter(new FileWriter(someFilename))) {
    // do stuff
    int temp = 0 / 0;    // throws an ArithmeticException
}
```

When the exception is thrown, the **`try`** will call `close()` on the `w` which will flush any buffered output and then close the `FileWriter`. But what happens if an `IOException` is thrown while flushing the output?

What happens is that any exception that is thrown while cleaning up a resource is *suppressed*. The exception is caught, and added to the primary exception's suppressed exception list. Next the *try-with-resources* will continue with the cleanup of the other resources. Finally, primary exception will be rethrown.

A similar pattern occurs if an exception it thrown during the resource initialization, or if the **`try`** block completes normally. The first exception thrown becomes the primary exception, and subsequent ones arising from cleanup are suppressed.

The suppressed exceptions can be retrieved from the primary exception object by calling `getSuppressedExceptions`.

Section 69.11: The try-finally and try-catch-finally statements

The **`try...catch...finally`** statement combines exception handling with clean-up code. The **`finally`** block contains code that will be executed in all circumstances. This makes them suitable for resource management, and other kinds of cleanup.

Try-finally

Here is an example of the simpler (**`try...finally`**) form:

```
try {
    doSomething();
} finally {
    cleanUp();
```

```
}
```

The behavior of the **`try...finally`** is as follows:

- The code in the **`try`** block is executed.
- If no exception was thrown in the **`try`** block:
    - The code in the **`finally`** block is executed.
    - If the **`finally`** block throws an exception, that exception is propagated.
    - Otherwise, control passes to the next statement after the **`try...finally`**.
- If an exception was thrown in the try block:
    - The code in the **`finally`** block is executed.
    - If the **`finally`** block throws an exception, that exception is propagated.
    - Otherwise, the original exception continues to propagate.

The code within **`finally`** block will always be executed. (The only exceptions are if `System.exit(int)` is called, or if the JVM panics.) Thus a **`finally`** block is the correct place code that always needs to be executed; e.g. closing files and other resources or releasing locks.

try-catch-finally

Our second example shows how **`catch`** and **`finally`** can be used together. It also illustrates that cleaning up resources is not straightforward.

```
// This code snippet writes the first line of a file to a string
String result = null;
Reader reader = null;
try {
    reader = new BufferedReader(new FileReader(fileName));
    result = reader.readLine();
} catch (IOException ex) {
    Logger.getLogger.warn("Unexpected IO error", ex);  // logging the exception
} finally {
    if (reader != null) {
        try {
            reader.close();
        } catch (IOException ex) {
            // ignore / discard this exception
        }
    }
}
```

The complete set of (hypothetical) behaviors of **`try...catch...finally`** in this example are too complicated to describe here. The simple version is that the code in the **`finally`** block will always be executed.

Looking at this from the perspective of resource management:

- We declare the "resource" (i.e. `reader` variable) before the **`try`** block so that it will be in scope for the **`finally`** block.
- By putting the **`new`** `FileReader(...)`, the **`catch`** is able to handle any `IOError` exception from thrown when opening the file.
- We need a `reader.close()` in the **`finally`** block because there are some exception paths that we cannot intercept either in the **`try`** block or in **`catch`** block.
- However, since an exception *might* have been thrown before `reader` was initialized, we also need an explicit **`null`** test.
- Finally, the `reader.close()` call might (hypothetically) throw an exception. We don't care about that, but if we don't catch the exception at source, we would need to deal with it further up the call stack.

Version ≥ Java SE 7

Java 7 and later provide an alternative try-with-resources syntax which significantly simplifies resource clean-up.

Section 69.12: The 'throws' clause in a method declaration

Java's *checked exception* mechanism requires the programmer to declare that certain methods *could* throw specifed checked exceptions. This is done using the `throws` clause. For example:

```
public class OddNumberException extends Exception { // a checked exception
}

public void checkEven(int number) throws OddNumberException {
    if (number % 2 != 0) {
        throw new OddNumberException();
    }
}
```

The `throws OddNumberException` declares that a call to `checkEven` *could* throw an exception that is of type `OddNumberException`.

A `throws` clause can declare a list of types, and can include unchecked exceptions as well as checked exceptions.

```
public void checkEven(Double number)
        throws OddNumberException, ArithmeticException {
    if (!Double.isFinite(number)) {
        throw new ArithmeticException("INF or NaN");
    } else if (number % 2 != 0) {
        throw new OddNumberException();
    }
}
```

What is the point of declaring unchecked exceptions as thrown?

The `throws` clause in a method declaration serves two purposes:

1. It tells the compiler which exceptions are thrown so that the compiler can report uncaught (checked) exceptions as errors.
2. It tells a programmer who is writing code that calls the method what exceptions to expect. For this purpose, it often makes to senses to include unchecked exceptions in a `throws` list.

Note: that the `throws` list is also used by the javadoc tool when generating API documentation, and by a typical IDE's "hover text" method tips.

Throws and method overriding

The `throws` clause forms part of a method's signature for the purpose of method overriding. An override method can be declared with the same set of checked exceptions as thrown by the overridden method, or with a subset. However the override method cannot add extra checked exceptions. For example:

```
@Override
public void checkEven(int number) throws NullPointerException // OK—NullPointerException is an
unchecked exception
    ...
```

```
@Override
public void checkEven(Double number) throws OddNumberException // OK—identical to the superclass
    ...

class PrimeNumberException extends OddNumberException {}
class NonEvenNumberException extends OddNumberException {}

@Override
public void checkEven(int number) throws PrimeNumberException, NonEvenNumberException // OK—these
are both subclasses

@Override
public void checkEven(Double number) throws IOExcepion         // ERROR
```

The reason for this rule is that if an overridden method can throw a checked exception that the overridden method could not throw, that would break type substitutability.

Chapter 70: Calendar and its Subclasses

Section 70.1: Creating Calendar objects

`Calendar` objects can be created by using `getInstance()` or by using the constructor `GregorianCalendar`.

It's important to notice that months in `Calendar` are zero based, which means that JANUARY is represented by an `int` value 0. In order to provide a better code, always use `Calendar` constants, such as `Calendar.JANUARY` to avoid misunderstandings.

```
Calendar calendar = Calendar.getInstance();
Calendar gregorianCalendar = new GregorianCalendar();
Calendar gregorianCalendarAtSpecificDay = new GregorianCalendar(2016, Calendar.JANUARY, 1);
Calendar gregorianCalendarAtSpecificDayAndTime = new GregorianCalendar(2016, Calendar.JANUARY, 1,
6, 55, 10);
```

Note: Always use the month constants: The numeric representation is misleading, e.g. `Calendar.JANUARY` has the value `0`

Section 70.2: Increasing / Decreasing calendar fields

`add()` and `roll()` can be used to increase/decrease `Calendar` fields.

```
Calendar calendar = new GregorianCalendar(2016, Calendar.MARCH, 31); // 31 March 2016
```

The `add()` method affects all fields, and behaves effectively if one were to add or subtract actual dates from the calendar

```
calendar.add(Calendar.MONTH, -6);
```

The above operation removes six months from the calendar, taking us back to 30 September 2015.

To change a particular field without affecting the other fields, use `roll()`.

```
calendar.roll(Calendar.MONTH, -6);
```

The above operation removes six months from the current *month*, so the month is identified as September. No other fields have been adjusted; the year has not changed with this operation.

Section 70.3: Subtracting calendars

To get a difference between two `Calendar`s, use `getTimeInMillis()` method:

```
Calendar c1 = Calendar.getInstance();
Calendar c2 = Calendar.getInstance();
c2.set(Calendar.DATE, c2.get(Calendar.DATE) + 1);

System.out.println(c2.getTimeInMillis() - c1.getTimeInMillis()); //outputs 86400000 (24 * 60 * 60 *
1000)
```

Section 70.4: Finding AM/PM

With Calendar class it is easy to find AM or PM.

```
Calendar cal = Calendar.getInstance();
cal.setTime(new Date());
if (cal.get(Calendar.AM_PM) == Calendar.PM)
    System.out.println("It is PM");
```

Chapter 71: Using the static keyword

Section 71.1: Reference to non-static member from static context

Static variables and methods are not part of an instance, There will always be a single copy of that variable no matter how many objects you create of a particular class.

For example you might want to have an immutable list of constants, it would be a good idea to keep it static and initialize it just once inside a static method. This would give you a significant performance gain if you are creating several instances of a particular class on a regular basis.

Furthermore you can also have a static block in a class as well. You can use it to assign a default value to a static variable. They are executed only once when the class is loaded into memory.

Instance variable as the name suggest are dependent on an instance of a particular object, they live to serve the whims of it. You can play around with them during a particular life cycle of an object.

All the fields and methods of a class used inside a static method of that class must be static or local. If you try to use instance (non-static) variables or methods, your code will not compile.

```
public class Week {
    static int daysOfTheWeek = 7; // static variable
    int dayOfTheWeek; // instance variable

    public static int getDaysLeftInWeek(){
        return Week.daysOfTheWeek-dayOfTheWeek; // this will cause errors
    }

    public int getDaysLeftInWeek(){
        return Week.daysOfTheWeek-dayOfTheWeek; // this is valid
    }

    public static int getDaysLeftInTheWeek(int today){
        return Week.daysOfTheWeek-today; // this is valid
    }

}
```

Section 71.2: Using static to declare constants

As the **`static`** keyword is used for accessing fields and methods without an instantiated class, it can be used to declare constants for use in other classes. These variables will remain constant across every instantiation of the class. By convention, **`static`** variables are always `ALL_CAPS` and use underscores rather than camel case. ex:

```
static E STATIC_VARIABLE_NAME
```

As constants cannot change, **`static`** can also be used with the **`final`** modifier:

For example, to define the mathematical constant of pi:

```
public class MathUtilities {

    static final double PI = 3.14159265358
```

```
}
```

Which can be used in any class as a constant, for example:

```
public class MathCalculations {

    //Calculates the circumference of a circle
    public double calculateCircumference(double radius) {
        return (2 * radius * MathUtilities.PI);
    }

}
```

Chapter 72: Properties Class

The properties object contains key and value pair both as a string. The java.util.Properties class is the subclass of Hashtable.

It can be used to get property value based on the property key. The Properties class provides methods to get data from properties file and store data into properties file. Moreover, it can be used to get properties of system.

Advantage of properties file

Recompilation is not required, if information is changed from properties file: If any information is changed from

Section 72.1: Loading properties

To load a properties file bundled with your application:

```
public class Defaults {

    public static Properties loadDefaults() {
        try (InputStream bundledResource =
            Defaults.class.getResourceAsStream("defaults.properties")) {

            Properties defaults = new Properties();
            defaults.load(bundledResource);
            return defaults;
        } catch (IOException e) {
            // Since the resource is bundled with the application,
            // we should never get here.
            throw new UncheckedIOException(
                "defaults.properties not properly packaged"
                + " with application", e);
        }
    }

}
```

Section 72.2: Saving Properties as XML

Storing Properties in a XML File

The way you store properties files as XML files is very similar to the way you would store them as `.properties` files. Just instead of using the `store()` you would use `storeToXML()`.

```
public void saveProperties(String location) throws IOException{
    // make new instance of properties
    Properties prop = new Properties();

    // set the property values
    prop.setProperty("name", "Steve");
    prop.setProperty("color", "green");
    prop.setProperty("age", "23");

    // check to see if the file already exists
    File file = new File(location);
    if (!file.exists()){
        file.createNewFile();
    }
```

```
    // save the properties
    prop.storeToXML(new FileOutputStream(file), "testing properties with xml");
}
```

When you open the file it will look like this.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
<comment>testing properties with xml</comment>
<entry key="age">23</entry>
<entry key="color">green</entry>
<entry key="name">Steve</entry>
</properties>
```

Loading Properties from a XML File

Now to load this file as a `properties` you need to call the `loadFromXML()` instead of the `load()` that you would use with regular `.properties` files.

```
public static void loadProperties(String location) throws FileNotFoundException, IOException{
    // make new properties instance to load the file into
    Properties prop = new Properties();

    // check to make sure the file exists
    File file = new File(location);
    if (file.exists()){
        // load the file
        prop.loadFromXML(new FileInputStream(file));

        // print out all the properties
        for (String name : prop.stringPropertyNames()){
            System.out.println(name + "=" + prop.getProperty(name));
        }
    } else {
        System.err.println("Error: No file found at: " + location);
    }
}
```

When you run this code you will get the following in the console:

```
age=23
color=green
name=Steve
```

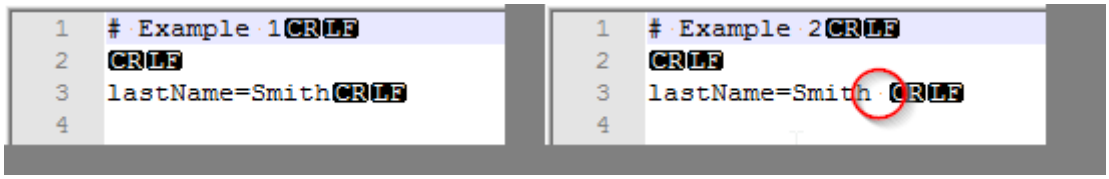
Section 72.3: Property files caveat: trailing whitespace

Take a close look at these two property files which are seemingly completely identical:

```
# Example 1

lastName=Smith

```

```
# Example 2

lastName=Smith

```

except they are really not identical:



(screenshots are from Notepad++)

Since trailing whitespace is preserved the value of `lastName` would be `"Smith"` in the first case and `"Smith "` in the second case.

Very rarely this is what users expect and one and can only speculate why this is the default behavior of `Properties` class. It is however easy to create an enhanced version of `Properties` that fixes this problem. The following class, **TrimmedProperties**, does just that. It is a drop-in replacement for standard Properties class.

```
import java.io.FileInputStream;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.Reader;
import java.util.Map.Entry;
import java.util.Properties;

/**
 * Properties class where values are trimmed for trailing whitespace if the
 * properties are loaded from a file.
 *
 * <p>
 * In the standard {@link java.util.Properties Properties} class trailing
 * whitespace is always preserved. When loading properties from a file such
 * trailing whitespace is almost always <i>unintentional</i>. This class fixes
 * this problem. The trimming of trailing whitespace only takes place if the
 * source of input is a file and only where the input is line oriented (meaning
 * that for example loading from XML file is <i>not</i> changed by this class).
 * For this reason this class is almost in all cases a safe drop-in replacement
 * for the standard <tt>Properties</tt>
 * class.
 *
 * <p>
 * Whitespace is defined here as any of space (U+0020) or tab (U+0009).
 * *
 */
public class TrimmedProperties extends Properties {

    /**
     * Reads a property list (key and element pairs) from the input byte stream.
     *
     * <p>Behaves exactly as {@link java.util.Properties#load(java.io.InputStream) }
     * with the exception that trailing whitespace is trimmed from property values
     * if <tt>inStream</tt> is an instance of <tt>FileInputStream</tt>.
     *
     * @see java.util.Properties#load(java.io.InputStream)
     * @param inStream the input stream.
     * @throws IOException if an error occurred when reading from the input stream.
     */
    @Override
    public void load(InputStream inStream) throws IOException {
        if (inStream instanceof FileInputStream) {
            // First read into temporary props using the standard way
            Properties tempProps = new Properties();
            tempProps.load(inStream);
```

```
            // Now trim and put into target
            trimAndLoad(tempProps);
        } else {
            super.load(inStream);
        }
    }

    /**
     * Reads a property list (key and element pairs) from the input character stream in a simple
line-oriented format.
     *
     * <p>Behaves exactly as {@link java.util.Properties#load(java.io.Reader)}
     * with the exception that trailing whitespace is trimmed on property values
     * if <tt>reader</tt> is an instance of <tt>FileReader</tt>.
     *
     * @see java.util.Properties#load(java.io.Reader) }
     * @param reader the input character stream.
     * @throws IOException if an error occurred when reading from the input stream.
     */
    @Override
    public void load(Reader reader) throws IOException {
        if (reader instanceof FileReader) {
            // First read into temporary props using the standard way
            Properties tempProps = new Properties();
            tempProps.load(reader);
            // Now trim and put into target
            trimAndLoad(tempProps);
        } else {
            super.load(reader);
        }
    }

    private void trimAndLoad(Properties p) {
        for (Entry<Object, Object> entry : p.entrySet()) {
            if (entry.getValue() instanceof String) {
                put(entry.getKey(), trimTrailing((String) entry.getValue()));
            } else {
                put(entry.getKey(), entry.getValue());
            }
        }
    }

    /**
     * Trims trailing space or tabs from a string.
     *
     * @param str
     * @return
     */
    public static String trimTrailing(String str) {
        if (str != null) {
            // read str from tail until char is no longer whitespace
            for (int i = str.length() - 1; i >= 0; i--) {
                if ((str.charAt(i) != ' ') && (str.charAt(i) != '\t')) {
                    return str.substring(0, i + 1);
                }
            }
        }
        return str;
    }
}
```

Chapter 73: Lambda Expressions

Lambda expressions provide a clear and concise way of implementing a single-method interface using an expression. They allow you to reduce the amount of code you have to create and maintain. While similar to anonymous classes, they have no type information by themselves. Type inference needs to happen.

Method references implement functional interfaces using existing methods rather than expressions. They belong to the lambda family as well.

Section 73.1: Introduction to Java lambdas

Functional Interfaces

Lambdas can only operate on a functional interface, which is an interface with just one abstract method. Functional interfaces can have any number of `default` or `static` methods. (For this reason, they are sometimes referred to as Single Abstract Method Interfaces, or SAM Interfaces).

```
interface Foo1 {
    void bar();
}

interface Foo2 {
    int bar(boolean baz);
}

interface Foo3 {
    String bar(Object baz, int mink);
}

interface Foo4 {
    default String bar() { // default so not counted
        return "baz";
    }
    void quux();
}
```

When declaring a functional interface the `@FunctionalInterface` annotation can be added. This has no special effect, but a compiler error will be generated if this annotation is applied to an interface which is not functional, thus acting as a reminder that the interface should not be changed.

```
@FunctionalInterface
interface Foo5 {
    void bar();
}

@FunctionalInterface
interface BlankFoo1 extends Foo3 { // inherits abstract method from Foo3
}

@FunctionalInterface
interface Foo6 {
    void bar();
    boolean equals(Object obj); // overrides one of Object's method so not counted
}
```

Conversely, this is **not** a functional interface, as it has more than **one abstract** method:

```
interface BadFoo {
    void bar();
    void quux(); // <-- Second method prevents lambda: which one should
                 // be considered as lambda?
}
```

This is **also not** a functional interface, as it does not have any methods:

```
interface BlankFoo2 { }
```

Take note of the following. Suppose you have

```
interface Parent { public int parentMethod(); }
```

and

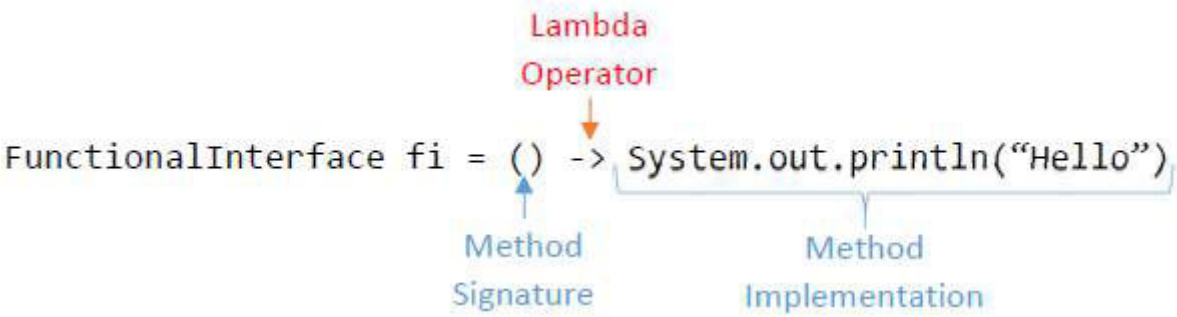
```
interface Child extends Parent { public int ChildMethod(); }
```

Then `Child` **cannot** be a functional interface since it has two specified methods.

Java 8 also provides a number of generic templated functional interfaces in the package `java.util.function`. For example, the built-in interface `Predicate<T>` wraps a single method which inputs a value of type `T` and returns a **`boolean`**.

Lambda Expressions

The basic structure of a Lambda expression is:



`fi` will then hold a singleton instance of a class, similar to an anonymous class, which implements `FunctionalInterface` and where the one method's definition is `{ System.out.println("Hello"); }`. In other words, the above is mostly equivalent to:

```
FunctionalInterface fi = new FunctionalInterface() {
    @Override
    public void theOneMethod() {
        System.out.println("Hello");
    }
};
```

The lambda is only "mostly equivalent" to the anonymous class because in a lambda, the meaning of expressions like **`this`**, **`super`** or `toString()` reference the class within which the assignment takes place, not the newly created object.

You cannot specify the name of the method when using a lambda—but you shouldn't need to, because a functional interface must have only one abstract method, so Java overrides that one.

In cases where the type of the lambda is not certain, (e.g. overloaded methods) you can add a cast to the lambda to tell the compiler what its type should be, like so:

```
Object fooHolder = (Foo1) () -> System.out.println("Hello");
System.out.println(fooHolder instanceof Foo1); // returns true
```

If the functional interface's single method takes parameters, the local formal names of these should appear between the brackets of the lambda. There is no need to declare the type of the parameter or return as these are taken from the interface (although it is not an error to declare the parameter types if you want to). Thus, these two examples are equivalent:

```
Foo2 longFoo = new Foo2() {
    @Override
    public int bar(boolean baz) {
        return baz ? 1 : 0;
    }
};
Foo2 shortFoo = (x) -> { return x ? 1 : 0; };
```

The parentheses around the argument can be omitted if the function only has one argument:

```
Foo2 np = x -> { return x ? 1 : 0; }; // okay
Foo3 np2 = x, y -> x.toString() + y // not okay
```

Implicit Returns

If the code placed inside a lambda is a Java *expression* rather than a *statement*, it is treated as a method which returns the value of the expression. Thus, the following two are equivalent:

```
IntUnaryOperator addOneShort = (x) -> (x + 1);
IntUnaryOperator addOneLong = (x) -> { return (x + 1); };
```

Accessing Local Variables (value closures)

Since lambdas are syntactic shorthand for anonymous classes, they follow the same rules for accessing local variables in the enclosing scope; the variables must be treated as `final` and not modified inside the lambda.

```
IntUnaryOperator makeAdder(int amount) {
    return (x) -> (x + amount); // Legal even though amount will go out of scope
                                // because amount is not modified
}

IntUnaryOperator makeAccumulator(int value) {
    return (x) -> { value += x; return value; }; // Will not compile
}
```

If it is necessary to wrap a changing variable in this way, a regular object that keeps a copy of the variable should be used. Read more in Java Closures with lambda expressions.

Accepting Lambdas

Because a lambda is an implementation of an interface, nothing special needs to be done to make a method accept a lambda: any function which takes a functional interface can also accept a lambda.

```
public void passMeALambda(Foo1 f) {
    f.bar();
}
passMeALambda(() -> System.out.println("Lambda called"));
```

The Type of a Lambda Expression

A lambda expression, by itself, does not have a specific type. While it is true that the types and number of parameters, along with the type of a return value can convey some type information, such information will only constrain what types it can be assigned to. The lambda receives a type when it is assigned to a functional interface type in one of the following ways:

- Direct assignment to a functional type, e.g. `myPredicate = s -> s.isEmpty()`
- Passing it as a parameter that has a functional type, e.g. `stream.filter(s -> s.isEmpty())`
- Returning it from a function that returns a functional type, e.g. `return s -> s.isEmpty()`
- Casting it to a functional type, e.g. `(Predicate<String>) s -> s.isEmpty()`

Until any such assignment to a functional type is made, the lambda does not have a definite type. To illustrate, consider the lambda expression `o -> o.isEmpty()`. The same lambda expression can be assigned to many different functional types:

```
Predicate<String> javaStringPred = o -> o.isEmpty();
Function<String, Boolean> javaFunc = o -> o.isEmpty();
Predicate<List> javaListPred = o -> o.isEmpty();
Consumer<String> javaStringConsumer = o -> o.isEmpty(); // return value is ignored!
com.google.common.base.Predicate<String> guavaPredicate = o -> o.isEmpty();
```

Now that they are assigned, the examples shown are of completely different types even though the lambda expressions looked the same, and they cannot be assigned to each other.

Section 73.2: Using Lambda Expressions to Sort a Collection

Sorting lists

Prior to Java 8, it was necessary to implement the `java.util.Comparator` interface with an anonymous (or named) class when sorting a list1:

Version ≥ Java SE 1.2

```
List<Person> people = ...
Collections.sort(
    people,
    new Comparator<Person>() {
        public int compare(Person p1, Person p2){
            return p1.getFirstName().compareTo(p2.getFirstName());
        }
    }
);
```

Starting with Java 8, the anonymous class can be replaced with a lambda expression. Note that the types for the parameters `p1` and `p2` can be left out, as the compiler will infer them automatically:

```
Collections.sort(
    people,
    (p1, p2) -> p1.getFirstName().compareTo(p2.getFirstName())
);
```

The example can be simplified by using `Comparator.comparing` and method references expressed using the `::`

(double colon) symbol.

```
Collections.sort(
    people,
    Comparator.comparing(Person::getFirstName)
);
```

A static import allows us to express this more concisely, but it is debatable whether this improves overall readability:

```
import static java.util.Collections.sort;
import static java.util.Comparator.comparing;
//...
sort(people, comparing(Person::getFirstName));
```

Comparators built this way can also be chained together. For example, after comparing people by their first name, if there are people with the same first name, the `thenComparing` method with also compare by last name:

```
sort(people, comparing(Person::getFirstName).thenComparing(Person::getLastName));
```

1 - Note that Collections.sort(...) only works on collections that are subtypes of `List`. The `Set` and `Collection` APIs do not imply any ordering of the elements.

Sorting maps

You can sort the entries of a `HashMap` by value in a similar fashion. (Note that a `LinkedHashMap` must be used as the target. The keys in an ordinary `HashMap` are unordered.)

```
Map<String, Integer> map = new HashMap();  // ... or any other Map class
// populate the map
map = map.entrySet()
    .stream()
    .sorted(Map.Entry.<String, Integer>comparingByValue())
    .collect(Collectors.toMap(k -> k.getKey(), v -> v.getValue(),
                              (k, v) -> k, LinkedHashMap::new));
```

Section 73.3: Method References

Method references allow predefined static or instance methods that adhere to a compatible functional interface to be passed as arguments instead of an anonymous lambda expression.

Assume that we have a model:

```
class Person {
    private final String name;
    private final String surname;

    public Person(String name, String surname){
        this.name = name;
        this.surname = surname;
    }

    public String getName(){ return name; }
    public String getSurname(){ return surname; }
}
```

```
List<Person> people = getSomePeople();
```

Instance method reference (to an arbitrary instance)

```
people.stream().map(Person::getName)
```

The equivalent lambda:

```
people.stream().map(person -> person.getName())
```

In this example, a method reference to the instance method `getName()` of type `Person`, is being passed. Since it's known to be of the collection type, the method on the instance (known later) will be invoked.

Instance method reference (to a specific instance)

```
people.forEach(System.out::println);
```

Since `System.out` is an instance of `PrintStream`, a method reference to this specific instance is being passed as an argument.

The equivalent lambda:

```
people.forEach(person -> System.out.println(person));
```

Static method reference

Also for transforming streams we can apply references to static methods:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
numbers.stream().map(String::valueOf)
```

This example passes a reference to the static `valueOf()` method on the `String` type. Therefore, the instance object in the collection is passed as an argument to `valueOf()`.

The equivalent lambda:

```
 numbers.stream().map(num -> String.valueOf(num))
```

Reference to a constructor

```
List<String> strings = Arrays.asList("1", "2", "3");
strings.stream().map(Integer::new)
```

Read Collect Elements of a Stream into a Collection to see how to collect elements to collection.

The single String argument constructor of the `Integer` type is being used here, to construct an integer given the string provided as the argument. In this case, as long as the string represents a number, the stream will be mapped to Integers. The equivalent lambda:

```
strings.stream().map(s -> new Integer(s));
```

Cheat-Sheet

Method Reference Format	Code	Equivalent Lambda
Static method	`TypeName::method`	`(args) -> TypeName.method(args)`

Non-static method (on instance*)	`instance::method (args) -> instance.method(args)`
Non-static method (no instance)	`TypeName::method (instance, args) -> instance.method(args)`
Constructor**	`TypeName::new (args) -> new TypeName(args)`
Array constructor	`TypeName[]::new (int size) -> new TypeName[size]`

* `instance` can be any expression that evaluates to a reference to an instance, e.g. `getInstance()::method`, `this::method`

** If `TypeName` is a non-static inner class, constructor reference is only valid within the scope of an outer class instance

Section 73.4: Implementing multiple interfaces

Sometimes you may want to have a lambda expression implementing more than one interface. This is mostly useful with marker interfaces (such as java.io.Serializable) since they don't add abstract methods.

For example, you want to create a `TreeSet` with a custom `Comparator` and then serialize it and send it over the network. The trivial approach:

```
TreeSet<Long> ts = new TreeSet<>((x, y) -> Long.compare(y, x));
```

doesn't work since the lambda for the comparator does not implement `Serializable`. You can fix this by using intersection types and explicitly specifying that this lambda needs to be serializable:

```
TreeSet<Long> ts = new TreeSet<>(
    (Comparator<Long> & Serializable) (x, y) -> Long.compare(y, x));
```

If you're frequently using intersection types (for example, if you're using a framework such as Apache Spark where almost everything has to be serializable), you can create empty interfaces and use them in your code instead:

```
public interface SerializableComparator extends Comparator<Long>, Serializable {}

public class CustomTreeSet {
  public CustomTreeSet(SerializableComparator comparator) {}
}
```

This way you're guaranteed that the passed comparator will be serializable.

Section 73.5: Lambda - Listener Example

Anonymous class listener

Before Java 8, it's very common that an anonymous class is used to handle click event of a JButton, as shown in the following code. This example shows how to implement an anonymous listener within the scope of `btn.addActionListener`.

```
JButton btn = new JButton("My Button");
btn.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button was pressed");
    }
});
```

Lambda listener

Because the `ActionListener` interface defines only one method `actionPerformed()`, it is a functional interface which means there's a place to use Lambda expressions to replace the boilerplate code. The above example can be re-written using Lambda expressions as follows:

```
JButton btn = new JButton("My Button");
btn.addActionListener(e -> {
    System.out.println("Button was pressed");
});
```

Section 73.6: Java Closures with lambda expressions

A lambda closure is created when a lambda expression references the variables of an enclosing scope (global or local). The rules for doing this are the same as those for inline methods and anonymous classes.

Local variables from an enclosing scope that are used within a lambda have to be **`final`**. With Java 8 (the earliest version that supports lambdas), they don't need to be *declared* **`final`** in the outside context, but must be treated that way. For example:

```
int n = 0; // With Java 8 there is no need to explicit final
Runnable r = () -> { // Using lambda
    int i = n;
    // do something
};
```

This is legal as long as the value of the `n` variable is not changed. If you try to change the variable, inside or outside the lambda, you will get the following compilation error:

> "local variables referenced from a lambda expression must be *final* or *effectively final*".

For example:

```
int n = 0;
Runnable r = () -> { // Using lambda
    int i = n;
    // do something
};
n++; // Will generate an error.
```

If it is necessary to use a changing variable within a lambda, the normal approach is to declare a **`final`** copy of the variable and use the copy. For example

```
int n = 0;
final int k = n; // With Java 8 there is no need to explicit final
Runnable r = () -> { // Using lambda
    int i = k;
    // do something
};
n++;      // Now will not generate an error
r.run();  // Will run with i = 0 because k was 0 when the lambda was created
```

Naturally, the body of the lambda does not see the changes to the original variable.

Note that Java does not support true closures. A Java lambda cannot be created in a way that allows it to see

changes in the environment in which it was instantiated. If you want to implement a closure that observes or makes changes to its environment, you should simulate it using a regular class. For example:

```
// Does not compile ...
public IntUnaryOperator createAccumulator() {
    int value = 0;
    IntUnaryOperator accumulate = (x) -> { value += x; return value; };
    return accumulate;
}
```

The above example will not compile for reasons discussed previously. We can work around the compilation error as follows:

```
// Compiles, but is incorrect ...
public class AccumulatorGenerator {
    private int value = 0;

    public IntUnaryOperator createAccumulator() {
        IntUnaryOperator accumulate = (x) -> { value += x; return value; };
        return accumulate;
    }
}
```

The problem is that this breaks the design contract for the `IntUnaryOperator` interface which states that instances should be functional and stateless. If such a closure is passed to built-in functions that accept functional objects, it is liable to cause crashes or erroneous behavior. Closures that encapsulate mutable state should be implemented as regular classes. For example.

```
// Correct ...
public class Accumulator {
   private int value = 0;

   public int accumulate(int x) {
      value += x;
      return value;
   }
}
```

Section 73.7: Lambdas and memory utilization

Since Java lambdas are closures, they can "capture" the values of variables in the enclosing lexical scope. While not all lambdas capture anything -- simple lambdas like `s -> s.length()` capture nothing and are called *stateless* -- capturing lambdas require a temporary object to hold the captured variables. In this code snippet, the lambda `() -> j` is a capturing lambda, and may cause an object to be allocated when it is evaluated:

```
public static void main(String[] args) throws Exception {
    for (int i = 0; i < 1000000000; i++) {
        int j = i;
        doSomethingWithLambda(() -> j);
    }
}
```

Although it might not be immediately obvious since the **`new`** keyword doesn't appear anywhere in the snippet, this code is liable to create 1,000,000,000 separate objects to represent the instances of the `() -> j` lambda expression. However, it should also be noted that future versions of Java1 may be able to optimize this so that *at runtime* the lambda instances were reused, or were represented in some other way.

1 - For instance, Java 9 introduces an optional "link" phase to the Java build sequence which will provide the opportunity for doing global optimizations like this.

Section 73.8: Using lambda expression with your own functional interface

Lambdas are meant to provide inline implementation code for single method interfaces and the ability to pass them around as we have been doing with normal variables. We call them Functional Interface.

For example, writing a Runnable in anonymous class and starting a Thread looks like:

```
//Old way
new Thread(
        new Runnable(){
            public void run(){
                System.out.println("run logic...");
            }
        }
).start();

//lambdas, from Java 8
new Thread(
        ()-> System.out.println("run logic...")
).start();
```

Now, in line with above, lets say you have some custom interface:

```
interface TwoArgInterface {
    int operate(int a, int b);
}
```

How do you use lambda to give implementation of this interface in your code? Same as Runnable example shown above. See the driver program below:

```
public class CustomLambda {
    public static void main(String[] args) {

        TwoArgInterface plusOperation = (a, b) -> a + b;
        TwoArgInterface divideOperation = (a,b)->{
            if (b==0) throw new IllegalArgumentException("Divisor can not be 0");
            return a/b;
        };

        System.out.println("Plus operation of 3 and 5 is: " + plusOperation.operate(3, 5));
        System.out.println("Divide operation 50 by 25 is: " + divideOperation.operate(50, 25));

    }
}
```

Section 73.9: Traditional style to Lambda style

Traditional way

```
interface MathOperation{
    boolean unaryOperation(int num);
}
```

```
public class LambdaTry {
    public static void main(String[] args) {
        MathOperation isEven = new MathOperation() {
            @Override
            public boolean unaryOperation(int num) {
                return num%2 == 0;
            }
        };

        System.out.println(isEven.unaryOperation(25));
        System.out.println(isEven.unaryOperation(20));
    }
}
```

Lambda style

1. Remove class name and functional interface body.

```
public class LambdaTry {
    public static void main(String[] args) {
        MathOperation isEven = (int num) -> {
            return num%2 == 0;
        };

        System.out.println(isEven.unaryOperation(25));
        System.out.println(isEven.unaryOperation(20));
    }
}
```

2. Optional type declaration

```
MathOperation isEven = (num) -> {
    return num%2 == 0;
};
```

3. Optional parenthesis around parameter, if it is single parameter

```
MathOperation isEven = num -> {
    return num%2 == 0;
};
```

4. Optional curly braces, if there is only one line in function body
5. Optional return keyword, if there is only one line in function body

```
MathOperation isEven = num -> num%2 == 0;
```

Section 73.10: `return` only returns from the lambda, not the outer method

The `return` method only returns from the lambda, not the outer method.

Beware that this is *different* from Scala and Kotlin!

```
void threeTimes(IntConsumer r) {
  for (int i = 0; i < 3; i++) {
    r.accept(i);
  }
}
```

```
void demo() {
  threeTimes(i -> {
    System.out.println(i);
    return; // Return from lambda to threeTimes only!
  });
}
```

This can lead to unexpected behavior when attempting to write own language constructs, as in builtin constructs such as **`for`** loops **`return`** behaves differently:

```
void demo2() {
  for (int i = 0; i < 3; i++) {
    System.out.println(i);
    return; // Return from 'demo2' entirely
  }
}
```

In Scala and Kotlin, `demo` and `demo2` would both only print `0`. But this is *not* more consistent. The Java approach is consistent with refactoring and the use of classes - the **`return`** in the code at the top, and the code below behaves the same:

```
void demo3() {
  threeTimes(new MyIntConsumer());
}

class MyIntConsumer implements IntConsumer {
  public void accept(int i) {
    System.out.println(i);
    return;
  }
}
```

Therefore, the Java **`return`** is more consistent with class methods and refactoring, but less with the **`for`** and **`while`** builtins, these remain special.

Because of this, the following two are equivalent in Java:

```
IntStream.range(1, 4)
    .map(x -> x * x)
    .forEach(System.out::println);
IntStream.range(1, 4)
    .map(x -> { return x * x; })
    .forEach(System.out::println);
```

Furthermore, the use of try-with-resources is safe in Java:

```
class Resource implements AutoCloseable {
  public void close() { System.out.println("close()"); }
}

void executeAround(Consumer<Resource> f) {
  try (Resource r = new Resource()) {
    System.out.print("before ");
    f.accept(r);
    System.out.print("after ");
  }
}

void demo4() {
```

```
  executeAround(r -> {
    System.out.print("accept() ");
    return; // Does not return from demo4, but frees the resource.
  });
}
```

will print `before accept() after close()`. In the Scala and Kotlin semantics, the try-with-resources would not be closed, but it would print `before accept()` only.

Section 73.11: Lambdas and Execute-around Pattern

There are several good examples of using lambdas as a FunctionalInterface in simple scenarios. A fairly common use case that can be improved by lambdas is what is called the Execute-Around pattern. In this pattern, you have a set of standard setup/teardown code that is needed for multiple scenarios surrounding use case specific code. A few common example of this are file io, database io, try/catch blocks.

```
interface DataProcessor {
    void process( Connection connection ) throws SQLException;;
}

public void doProcessing( DataProcessor processor ) throws SQLException{
    try (Connection connection = DBUtil.getDatabaseConnection();) {
        processor.process(connection);
        connection.commit();
    }
}
```

Then to call this method with a lambda it might look like:

```
public static void updateMyDAO(MyVO vo) throws DatabaseException {
    doProcessing((Connection conn) -> MyDAO.update(conn, ObjectMapper.map(vo)));
}
```

This is not limited to I/O operations. It can apply to any scenario where similar setup/tear down tasks are applicable with minor variations. The main benefit of this Pattern is code re-use and enforcing DRY (Don't Repeat Yourself).

Section 73.12: Using lambda expressions & predicates to get a certain value(s) from a list

Starting with Java 8, you can use lambda expressions & predicates.

Example: Use a lambda expressions & a predicate to get a certain value from a list. In this example every person will be printed out with the fact if they are 18 and older or not.

Person Class:

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public int getAge() { return age; }
    public String getName() { return name; }
```

```
}
```

The built-in interface Predicate from the java.util.function.Predicate packages is a functional interface with a `boolean test(T t)` method.

Example Usage:

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class LambdaExample {
    public static void main(String[] args) {
        List<Person> personList = new ArrayList<Person>();
        personList.add(new Person("Jeroen", 20));
        personList.add(new Person("Jack", 5));
        personList.add(new Person("Lisa", 19));

        print(personList, p -> p.getAge() >= 18);
    }

    private static void print(List<Person> personList, Predicate<Person> checker) {
        for (Person person : personList) {
            if (checker.test(person)) {
                System.out.print(person + " matches your expression.");
            } else {
                System.out.println(person  + " doesn't match your expression.");
            }
        }
    }
}
```

The `print(personList, p -> p.getAge() >= 18);` method takes a lambda expression (because the Predicate is used a parameter) where you can define the expression that is needed. The checker's test method checks if this expression is correct or not: `checker.test(person)`.

You can easily change this to something else, for example to `print(personList, p -> p.getName().startsWith("J"));`. This will check if the person's name starts with a "J".

Chapter 74: Basic Control Structures

Section 74.1: Switch statement

The `switch` statement is Java's multi-way branch statement. It is used to take the place of long `if-else if-else` chains, and make them more readable. However, unlike `if` statements, one may not use inequalities; each value must be concretely defined.

There are three critical components to the `switch` statement:

- `case`: This is the value that is evaluated for equivalence with the argument to the `switch` statement.
- `default`: This is an optional, catch-all expression, should none of the `case` statements evaluate to `true`.
- Abrupt completion of the `case` statement; usually `break`: This is required to prevent the undesired evaluation of further `case` statements.

With the exception of `continue`, it is possible to use any statement which would cause the abrupt completion of a statement. This includes:

- `break`
- `return`
- `throw`

In the example below, a typical `switch` statement is written with four possible cases, including `default`.

```
Scanner scan = new Scanner(System.in);
int i = scan.nextInt();
switch (i) {
    case 0:
        System.out.println("i is zero");
        break;
    case 1:
        System.out.println("i is one");
        break;
    case 2:
        System.out.println("i is two");
        break;
    default:
        System.out.println("i is less than zero or greater than two");
}
```

By omitting `break` or any statement which would an abrupt completion, we can leverage what are known as "fall-through" cases, which evaluate against several values. This can be used to create ranges for a value to be successful against, but is still not as flexible as inequalities.

```
Scanner scan = new Scanner(System.in);
int foo = scan.nextInt();
switch(foo) {
    case 1:
        System.out.println("I'm equal or greater than one");
    case 2:
    case 3:
        System.out.println("I'm one, two, or three");
        break;
    default:
        System.out.println("I'm not either one, two, or three");
}
```

In case of `foo == 1` the output will be:

```
I'm equal or greater than one
I'm one, two, or three
```

In case of `foo == 3` the output will be:

```
I'm one, two, or three
```

Version ≥ Java SE 5

The switch statement can also be used with **`enum`**s.

```
enum Option {
    BLUE_PILL,
    RED_PILL
}

public void takeOne(Option option) {
    switch(option) {
        case BLUE_PILL:
            System.out.println("Story ends, wake up, believe whatever you want.");
            break;
        case RED_PILL:
            System.out.println("I show you how deep the rabbit hole goes.");
            break;

    }
}
```

Version ≥ Java SE 7

The **`switch`** statement can also be used with `String`s.

```
public void rhymingGame(String phrase) {
    switch (phrase) {
        case "apples and pears":
            System.out.println("Stairs");
            break;
        case "lorry":
            System.out.println("truck");
            break;
        default:
            System.out.println("Don't know any more");
    }
}
```

Section 74.2: do...while Loop

The **`do...while`** loop differs from other loops in that it is guaranteed to execute **at least once**. It is also called the "post-test loop" structure because the conditional statement is performed after the main loop body.

```
int i = 0;
do {
    i++;
    System.out.println(i);
} while (i < 100); // Condition gets checked AFTER the content of the loop executes.
```

In this example, the loop will run until the number `100` is printed (even though the condition is `i < 100` and not `i <= 100`), because the loop condition is evaluated *after* the loop executes.

With the guarantee of at least one execution, it is possible to declare variables outside of the loop and initialize them inside.

```
String theWord;
Scanner scan = new Scanner(System.in);
do {
    theWord = scan.nextLine();
} while (!theWord.equals("Bird"));

System.out.println(theWord);
```

In this context, `theWord` is defined outside of the loop, but since it's guaranteed to have a value based on its natural flow, `theWord` will be initialized.

Section 74.3: For Each

`Version ≥ Java SE 5`

With Java 5 and up, one can use for-each loops, also known as enhanced for-loops:

```
List strings = new ArrayList();

strings.add("This");
strings.add("is");
strings.add("a for-each loop");
```

(String string : strings) { System.out.println(string); }

For each loops can be used to iterate over Arrays and implementations of the `Iterable` interface, the later includes Collections classes, such as `List` or `Set`.

The loop variable can be of any type that is assignable from the source type.

The loop variable for a enhanced for loop for `Iterable<T>` or `T[]` can be of type `S`, if

- `T extends S`
- both `T` and `S` are primitive types and assignable without a cast
- `S` is a primitive type and `T` can be converted to a type assignable to `S` after unboxing conversion.
- `T` is a primitive type and can be converted to `S` by autoboxing conversion.

Examples:

```
T elements = ...
for (S s : elements) {
}
```

T	S	Compiles
int[]	long	yes
long[]	int	no
`Iterable<Byte>`	long	yes
`Iterable<String>`	CharSequence	yes
`Iterable<CharSequence>`	String	no
int[]	Long	no

int[]	Integer	yes

Section 74.4: Continue Statement in Java

The continue statement is used to skip the remaining steps in the current iteration and start with the next loop iteration. The control goes from the **continue** statement to the step value (increment or decrement), if any.

```
String[] programmers = {"Adrian", "Paul", "John", "Harry"};

    //john is not printed out
    for (String name : programmers) {
        if (name.equals("John"))
            continue;
        System.out.println(name);
    }
```

The **continue** statement can also make the control of the program shift to the step value (if any) of a named loop:

```
Outer: // The name of the outermost loop is kept here as 'Outer'
for(int i = 0; i < 5; )
{
    for(int j = 0; j < 5; j++)
    {
        continue Outer;
    }
}
```

Section 74.5: If / Else If / Else Control

```
if (i < 2) {
  System.out.println("i is less than 2");
} else if (i > 2) {
  System.out.println("i is more than 2");
} else {
  System.out.println("i is not less than 2, and not more than 2");
}
```

The `if` block will only run when `i` is 1 or less.

The **`else if`** condition is checked only if all the conditions before it (in previous **`else if`** constructs, and the parent `if` constructs) have been tested to **`false`**. In this example, the **`else if`** condition will only be checked if `i` is greater than or equal to 2.

If its result is **`true`**, its block is run, and any **`else if`** and **`else`** constructs after it will be skipped.

If none of the `if` and **`else if`** conditions have been tested to **`true`**, the **`else`** block at the end will be run.

Section 74.6: For Loops

```
for (int i = 0; i < 100; i++) {
    System.out.println(i);
}
```

The three components of the **`for`** loop (separated by `;`) are variable declaration/initialization (here **`int`** `i = 0`), the condition (here `i < 100`), and the increment statement (here `i++`). The variable declaration is done once as if placed just inside the `{` on the first run. Then the condition is checked, if it is **`true`** the body of the loop will execute, if it is

`false` the loop will stop. Assuming the loop continues, the body will execute and finally when the } is reached the increment statement will execute just before the condition is checked again.

The curly braces are optional (you can one line with a semicolon) if the loop contains just one statement. But, it's always recommended to use braces to avoid misunderstandings and bugs.

The `for` loop components are optional. If your business logic contains one of these parts, you can omit the corresponding component from your `for` loop.

```
int i = obj.getLastestValue(); // i value is fetched from a method

for (; i < 100; i++) { // here initialization is not done
    System.out.println(i);
}
```

The `for (;;) { function-body }` structure is equal to a `while (true)` loop.

`Nested For Loops`

Any looping statement having another loop statement inside called nested loop. The same way for looping having more inner loop is called 'nested for loop'.

```
    for(;;){
        //Outer Loop Statements
        for(;;){
            //Inner Loop Statements
        }
        //Outer Loop Statements
    }
```

Nested for loop can be demonstrated to print triangle shaped numbers.

```
for(int i=9;i>0;i--){//Outer Loop
    System.out.println();
    for(int k=i;k>0;k--){//Inner Loop -1
        System.out.print(" ");
    }
    for(int j=i;j<=9;j++){//Inner Loop -2
        System.out.print(" "+j);
    }
 }
```

Section 74.7: Ternary Operator

Sometimes you have to check for a condition and set the value of a variable.

For ex.

```
String name;

if (A > B) {
    name = "Billy";
} else {
    name = "Jimmy";
}
```

This can be easily written in one line as

```
String name = A > B ? "Billy" : "Jimmy";
```

The value of the variable is set to the value immediately after the condition, if the condition is true. If the condition is false, the second value will be given to the variable.

Section 74.8: Try ... Catch ... Finally

The `try { ... } catch ( ... ) { ... }` control structure is used for handling Exceptions.

```
String age_input = "abc";
try {
    int age = Integer.parseInt(age_input);
    if (age >= 18) {
        System.out.println("You can vote!");
    } else {
        System.out.println("Sorry, you can't vote yet.");
    }
} catch (NumberFormatException ex) {
    System.err.println("Invalid input.  '" + age_input + "' is not a valid integer.");
}
```

This would print:

> Invalid input. 'abc' is not a valid integer.

A **`finally`** clause can be added after the **`catch`**. The **`finally`** clause would always be executed, regardless of whether an exception was thrown.

`try { ... } catch ( ... ) { ... } finally { ... }`

```
String age_input = "abc";
try {
    int age = Integer.parseInt(age_input);
    if (age >= 18) {
        System.out.println("You can vote!");
    } else {
        System.out.println("Sorry, you can't vote yet.");
    }
} catch (NumberFormatException ex) {
    System.err.println("Invalid input.  '" + age_input + "' is not a valid integer.");
} finally {
    System.out.println("This code will always be run, even if an exception is thrown");
}
```

This would print:

> Invalid input. 'abc' is not a valid integer.
> This code will always be run, even if an exception is thrown

Section 74.9: Break

The **`break`** statement ends a loop (like **`for`**, **`while`**) or the evaluation of a switch statement.

Loop:

```
while(true) {
   if(someCondition == 5) {
       break;
   }
}
```

The loop in the example would run forever. But when `someCondition` equals `5` at some point of execution, then the loop ends.

If multiple loops are cascaded, only the most inner loop ends using **`break`**.

Section 74.10: While Loops

```
int i = 0;
while (i < 100) { // condition gets checked BEFORE the loop body executes
    System.out.println(i);
    i++;
}
```

A **`while`** loop runs as long as the condition inside the parentheses is **`true`**. This is also called the "pre-test loop" structure because the conditional statement must be met before the main loop body is performed every time.

The curly braces are optional if the loop contains just one statement, but some coding style conventions prefers having the braces regardless.

Section 74.11: If / Else

```
int i = 2;
if (i < 2) {
  System.out.println("i is less than 2");
} else {
  System.out.println("i is greater than 2");
}
```

An `if` statement executes code conditionally depending on the result of the condition in parentheses. When condition in parentheses is true it will enter to the block of if statement which is defined by curly braces like { and }. opening bracket till the closing bracket is the scope of the if statement.

The **`else`** block is optional and can be omitted. It runs if the `if` statement is **`false`** and does not run if the `if` statement is true Because in that case `if` statement executes.

See also: Ternary If

Section 74.12: Nested break / continue

It's possible to **`break`** / **`continue`** to an outer loop by using label statements:

```
outerloop:
for(...) {
    innerloop:
    for(...) {
        if(condition1)
            break outerloop;

        if(condition2)
            continue innerloop; // equivalent to: continue;
```

```
    }
}
```

There is no other use for labels in Java.

Chapter 75: BufferedWriter

Section 75.1: Write a line of text to File

This code writes the string to a file. It is important to close the writer, so this is done in a **`finally`** block.

```
  public void writeLineToFile(String str) throws IOException {
    File file = new File("file.txt");
    BufferedWriter bw = null;
    try {
      bw = new BufferedWriter(new FileWriter(file));
      bw.write(str);
    } finally {
      if (bw != null) {
        bw.close();
      }
    }
  }
```

Also note that `write(String s)` does not place newline character after string has been written. To put it use `newLine()` method.

Version ≥ Java SE 7

Java 7 adds the `java.nio.file` package, and try-with-resources:

```
public void writeLineToFile(String str) throws IOException {
    Path path = Paths.get("file.txt");
    try (BufferedWriter bw = Files.newBufferedWriter(path)) {
        bw.write(str);
    }
}
```

Chapter 76: New File I/O

Section 76.1: Creating paths

The `Path` class is used to programmaticaly represent a path in the file system (and can therefore point to files as well as directories, even to non-existent ones)

A path can be obtained using the helper class `Paths`:

```
Path p1 = Paths.get("/var/www");
Path p2 = Paths.get(URI.create("file:///home/testuser/File.txt"));
Path p3 = Paths.get("C:\\Users\\DentAr\\Documents\\HHGTDG.odt");
Path p4 = Paths.get("/home", "arthur", "files", "diary.tex");
```

Section 76.2: Manipulating paths

Joining Two Paths

Paths can be joined using the `resolve()` method. The path passed has to be a partial path, which is a path that doesn't include the root element.

```
Path p5 = Paths.get("/home/");
Path p6 = Paths.get("arthur/files");
Path joined = p5.resolve(p6);
Path otherJoined = p5.resolve("ford/files");
```

```
joined.toString() == "/home/arthur/files"
otherJoined.toString() == "/home/ford/files"
```

Normalizing a path

Paths may contain the elements `.` (which points to the directory you're currently in) and `..`(which points to the parent directory).

When used in a path, `.` can be removed at any time without changing the path's destination, and `..` can be removed together with the preceding element.

With the Paths API, this is done using the `.normalize()` method:

```
Path p7 = Paths.get("/home/./arthur/../ford/files");
Path p8 = Paths.get("C:\\Users\\.\\..\\Program Files");
```

```
p7.normalize().toString() == "/home/ford/files"
p8.normalize().toString() == "C:\\Program Files"
```

Section 76.3: Retrieving information about a path

Information about a path can be get using the methods of a `Path` object:

- `toString()` returns the string representation of the path

  ```
  Path p1 = Paths.get("/var/www"); // p1.toString() returns "/var/www"
  ```

- `getFileName()` returns the file name (or, more specifically, the last element of the path

  ```
  Path p1 = Paths.get("/var/www"); // p1.getFileName() returns "www"
  ```

```
Path p3 = Paths.get("C:\\Users\\DentAr\\Documents\\HHGTDG.odt"); // p3.getFileName() returns
"HHGTDG.odt"
```

- `getNameCount()` returns the number of elements that form the path

```
Path p1 = Paths.get("/var/www"); // p1.getNameCount() returns 2
```

- `getName(int index)` returns the element at the given index

```
Path p1 = Paths.get("/var/www"); // p1.getName(0) returns "var", p1.getName(1) returns "www"
```

- `getParent()` returns the path of the parent directory

```
Path p1 = Paths.get("/var/www"); // p1.getParent().toString() returns "/var"
```

- `getRoot()` returns the root of the path

```
Path p1 = Paths.get("/var/www"); // p1.getRoot().toString() returns "/"
Path p3 = Paths.get("C:\\Users\\DentAr\\Documents\\HHGTDG.odt"); // p3.getRoot().toString()
returns "C:\\"
```

Section 76.4: Retrieving information using the filesystem

To interact with the filesystem you use the methods of the class `Files`.

Checking existence

To check the existence of the file or directory a path points to, you use the following methods:

```
Files.exists(Path path)
```

and

```
Files.notExists(Path path)
```

`!Files.exists(path)` does not neccesarily have to be equal to `Files.notExists(path)`, because there are three possible scenarios:

- A file's or directory's existence is verified (`exists` returns **`true`** and `notExists` returns **`false`** in this case)
- A file's or directory's nonexistence is verfied (`exists` returns **`false`** and `notExists` returns **`true`**)
- Neither the existence nor the nonexistence of a file or a directory can be verified (for example due to access restrictions): Both `exists` and `nonExists` return false.

Checking whether a path points to a file or a directory

This is done using `Files.isDirectory(Path path)` and `Files.isRegularFile(Path path)`

```
Path p1 = Paths.get("/var/www");
Path p2 = Paths.get("/home/testuser/File.txt");
```

```
Files.isDirectory(p1) == true
Files.isRegularFile(p1) == false

Files.isDirectory(p2) == false
```

```
Files.isRegularFile(p2) == true
```

Getting properties

This can be done using the following methods:

```
Files.isReadable(Path path)
Files.isWritable(Path path)
Files.isExecutable(Path path)

Files.isHidden(Path path)
Files.isSymbolicLink(Path path)
```

Getting MIME type

```
Files.probeContentType(Path path)
```

This tries to get the MIME type of a file. It returns a MIME type String, like this:

- `text/plain` for text files
- `text/html` for HTML pages
- `application/pdf` for PDF files
- `image/png` for PNG files

Section 76.5: Reading files

Files can be read byte- and line-wise using the `Files` class.

```
Path p2 = Paths.get(URI.create("file:///home/testuser/File.txt"));
byte[] content = Files.readAllBytes(p2);
List<String> linesOfContent = Files.readAllLines(p2);
```

`Files.readAllLines()` optionally takes a charset as parameter (default is `StandardCharsets.UTF_8`):

```
List<String> linesOfContent = Files.readAllLines(p2, StandardCharsets.ISO_8859_1);
```

Section 76.6: Writing files

Files can be written bite- and line-wise using the `Files` class

```
Path p2 = Paths.get("/home/testuser/File.txt");
List<String> lines = Arrays.asList(
    new String[]{"First line", "Second line", "Third line"});

Files.write(p2, lines);
```

```
Files.write(Path path, byte[] bytes)
```

Existing files wile be overridden, non-existing files will be created.

Chapter 77: File I/O

Java I/O (Input and Output) is used to process the input and produce the output. Java uses the concept of stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations. Handling files is also done in java by Java I/O API.

Section 77.1: Migrating from java.io.File to Java 7 NIO (java.nio.file.Path)

These examples assume that you already know what Java 7's NIO is in general, and you are used to writing code using `java.io.File`. Use these examples as a means to quickly find more NIO-centric documentation for migrating.

There is much more to Java 7's NIO such as memory-mapped files or opening a ZIP or JAR file using FileSystem. These examples will only cover a limited number of basic use cases.

As a basic rule, if you are used to perform a file system read/write operation using a `java.io.File` instance method, you will find it as a static method within `java.nio.file.Files`.

Point to a path

```
// -> IO
File file = new File("io.txt");

// -> NIO
Path path = Paths.get("nio.txt");
```

Paths relative to another path

```
// Forward slashes can be used in place of backslashes even on a Windows operating system
// -> IO
File folder = new File("C:/");
File fileInFolder = new File(folder, "io.txt");

// -> NIO
Path directory = Paths.get("C:/");
Path pathInDirectory = directory.resolve("nio.txt");
```

Converting File from/to Path for use with libraries

```
// -> IO to NIO
Path pathFromFile = new File("io.txt").toPath();

// -> NIO to IO
File fileFromPath = Paths.get("nio.txt").toFile();
```

Check if the file exists and delete it if it does

```
// -> IO
if (file.exists()) {
    boolean deleted = file.delete();
    if (!deleted) {
        throw new IOException("Unable to delete file");
    }
}

// -> NIO
Files.deleteIfExists(path);
```

Write to a file via an OutputStream

There are several ways to write and read from a file using NIO for different performance and memory constraints, readability and use cases, such as `FileChannel`, `Files.write(Path path, byte\[\] bytes, OpenOption...`

`options)`... In this example, only `OutputStream` is covered, but you are strongly encouraged to learn about memory-mapped files and the various static methods available in `java.nio.file.Files`.

```
List<String> lines = Arrays.asList(
        String.valueOf(Calendar.getInstance().getTimeInMillis()),
        "line one",
        "line two");

// -> IO
if (file.exists()) {
    // Note: Not atomic
    throw new IOException("File already exists");
}
try (FileOutputStream outputStream = new FileOutputStream(file)) {
    for (String line : lines) {
        outputStream.write((line + System.lineSeparator()).getBytes(StandardCharsets.UTF_8));
    }
}

// -> NIO
try (OutputStream outputStream = Files.newOutputStream(path, StandardOpenOption.CREATE_NEW)) {
    for (String line : lines) {
        outputStream.write((line + System.lineSeparator()).getBytes(StandardCharsets.UTF_8));
    }
}
```

Iterating on each file within a folder

```
// -> IO
for (File selectedFile : folder.listFiles()) {
    // Note: Depending on the number of files in the directory folder.listFiles() may take a long
time to return
    System.out.println((selectedFile.isDirectory() ? "d" : "f") + " " +
selectedFile.getAbsolutePath());
}

// -> NIO
Files.walkFileTree(directory, EnumSet.noneOf(FileVisitOption.class), 1, new
SimpleFileVisitor<Path>() {
    @Override
    public FileVisitResult preVisitDirectory(Path selectedPath, BasicFileAttributes attrs) throws
IOException {
        System.out.println("d " + selectedPath.toAbsolutePath());
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult visitFile(Path selectedPath, BasicFileAttributes attrs) throws
IOException {
        System.out.println("f " + selectedPath.toAbsolutePath());
        return FileVisitResult.CONTINUE;
    }
});
```

Recursive folder iteration

```
// -> IO
recurseFolder(folder);

// -> NIO
// Note: Symbolic links are NOT followed unless explicitly passed as an argument to
Files.walkFileTree
Files.walkFileTree(directory, new SimpleFileVisitor<Path>() {
    @Override
```

```
    public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs) throws
IOException {
        System.out.println("d " + selectedPath.toAbsolutePath());
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult visitFile(Path selectedPath, BasicFileAttributes attrs) throws
IOException {
        System.out.println("f " + selectedPath.toAbsolutePath());
        return FileVisitResult.CONTINUE;
    }
});


private static void recurseFolder(File folder) {
    for (File selectedFile : folder.listFiles()) {
        System.out.println((selectedFile.isDirectory() ? "d" : "f") + " " +
selectedFile.getAbsolutePath());
        if (selectedFile.isDirectory()) {
            // Note: Symbolic links are followed
            recurseFolder(selectedFile);
        }
    }
}
```

Section 77.2: Reading an image from a file

```
import java.awt.Image;
import javax.imageio.ImageIO;

...

try {
    Image img = ImageIO.read(new File("~/Desktop/cat.png"));
} catch (IOException e) {
    e.printStackTrace();
}
```

Section 77.3: File Read/Write Using FileInputStream/FileOutputStream

Write to a file test.txt:

```
String filepath ="C:\\test.txt";
FileOutputStream fos = null;
try {
      fos = new FileOutputStream(filepath);
      byte[] buffer = "This will be written in test.txt".getBytes();
      fos.write(buffer, 0, buffer.length);
      fos.close();
} catch (FileNotFoundException e) {
      e.printStackTrace();
} catch (IOException e) {
      e.printStackTrace();
} finally{
      if(fos != null)
        fos.close();
}
```

Read from file test.txt:

```
String filepath ="C:\\test.txt";
FileInputStream fis = null;
try {
   fis = new FileInputStream(filepath);
   int length = (int) new File(filepath).length();
   byte[] buffer = new byte[length];
   fis.read(buffer, 0, length);
} catch (FileNotFoundException e) {
     e.printStackTrace();
} catch (IOException e) {
     e.printStackTrace();
} finally{
   if(fis != null)
     fis.close();
}
```

Note, that since Java 1.7 the try-with-resources statement was introduced what made implementation of reading\writing operation much simpler:

Write to a file test.txt:

```
String filepath ="C:\\test.txt";
try (FileOutputStream fos = new FileOutputStream(filepath)){
    byte[] buffer = "This will be written in test.txt".getBytes();
    fos.write(buffer, 0, buffer.length);
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

Read from file test.txt:

```
String filepath ="C:\\test.txt";
try (FileInputStream fis = new FileInputStream(filepath)){
    int length = (int) new File(filepath).length();
    byte[] buffer = new byte[length];
    fis.read(buffer, 0, length);
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

Section 77.4: Reading all bytes to a byte[]

Java 7 introduced the very useful Files class

Version ≥ Java SE 7

```
import java.nio.file.Files;
import java.nio.file.Paths;
import java.nio.file.Path;

Path path = Paths.get("path/to/file");

try {
    byte[] data = Files.readAllBytes(path);
```

```
} catch(IOException e) {
    e.printStackTrace();
}
```

Section 77.5: Copying a file using Channel

We can use `Channel` to copy file content faster. To do so, we can use `transferTo()` method of `FileChannel` .

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.channels.FileChannel;

public class FileCopier {

    public static void main(String[] args) {
        File sourceFile = new File("hello.txt");
        File sinkFile = new File("hello2.txt");
        copy(sourceFile, sinkFile);
    }

    public static void copy(File sourceFile, File destFile) {
        if (!sourceFile.exists() || !destFile.exists()) {
            System.out.println("Source or destination file doesn't exist");
            return;
        }


        try (FileChannel srcChannel = new FileInputStream(sourceFile).getChannel();
             FileChannel sinkChanel = new FileOutputStream(destFile).getChannel()) {

            srcChannel.transferTo(0, srcChannel.size(), sinkChanel);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Section 77.6: Writing a byte[] to a file

Version ≥ Java SE 7

```
byte[] bytes = { 0x48, 0x65, 0x6c, 0x6c, 0x6f };

try(FileOutputStream stream = new FileOutputStream("Hello world.txt")) {
    stream.write(bytes);
} catch (IOException ioe) {
    // Handle I/O Exception
    ioe.printStackTrace();
}
```

Version < Java SE 7

```
byte[] bytes = { 0x48, 0x65, 0x6c, 0x6c, 0x6f };

FileOutputStream stream = null;
try {
    stream = new FileOutputStream("Hello world.txt");
    stream.write(bytes);
} catch (IOException ioe) {
    // Handle I/O Exception
```

```
    ioe.printStackTrace();
} finally {
    if (stream != null) {
        try {
            stream.close();
        } catch (IOException ignored) {}
    }
}
```

Most java.io file APIs accept both `String`s and `File`s as arguments, so you could as well use

```
File file = new File("Hello world.txt");
FileOutputStream stream = new FileOutputStream(file);
```

Section 77.7: Stream vs Writer/Reader API

Streams provide the most direct access to the binary content, so any `InputStream` / `OutputStream` implementations always operate on **`int`**s and **`byte`**s.

```
// Read a single byte from the stream
int b = inputStream.read();
if (b >= 0) { // A negative value represents the end of the stream, normal values are in the range 0
- 255
    // Write the byte to another stream
    outputStream.write(b);
}

// Read a chunk
byte[] data = new byte[1024];
int nBytesRead = inputStream.read(data);
if (nBytesRead >= 0) { // A negative value represents end of stream
    // Write the chunk to another stream
    outputStream.write(data, 0, nBytesRead);
}
```

There are some exceptions, probably most notably the `PrintStream` which adds the "ability to print representations of various data values conveniently". This allows to use `System.out` both as a binary `InputStream` and as a textual output using methods such as `System.out.println()`.

Also, some stream implementations work as an interface to higher-level contents such as Java objects (see Serialization) or native types, e.g. `DataOutputStream` / `DataInputStream`.

With the `Writer` and `Reader` classes, Java also provides an API for explicit character streams. Although most applications will base these implementations on streams, the character stream API does not expose any methods for binary content.

```
// This example uses the platform's default charset, see below
// for a better implementation.

Writer writer = new OutputStreamWriter(System.out);
writer.write("Hello world!");

Reader reader = new InputStreamReader(System.in);
char singleCharacter = reader.read();
```

Whenever it is necessary to encode characters into binary data (e.g. when using the `InputStreamWriter` / `OutputStreamWriter` classes), you should specify a charset if you do not want to depend on the platform's default

charset. When in doubt, use a Unicode-compatible encoding, e.g. UTF-8 which is supported on all Java platforms. Therefore, you should probably stay away from classes like `FileWriter` and `FileReader` as those always use the default platform charset. A better way to access files using character streams is this:

```
Charset myCharset = StandardCharsets.UTF_8;

Writer writer = new OutputStreamWriter( new FileOutputStream("test.txt"), myCharset );
writer.write('Ä');
writer.flush();
writer.close();

Reader reader = new InputStreamReader( new FileInputStream("test.txt"), myCharset );
char someUnicodeCharacter = reader.read();
reader.close();
```

One of the most commonly used `Reader`s is `BufferedReader` which provides a method to read whole lines of text from another reader and is presumably the simplest way to read a character stream line by line:

```
// Read from baseReader, one line at a time
BufferedReader reader = new BufferedReader( baseReader );
String line;
while((line = reader.readLine()) != null) {
  // Remember: System.out is a stream, not a writer!
  System.out.println(line);
}
```

Section 77.8: Reading a file with a Scanner

Reading a file line by line

```
public class Main {

    public static void main(String[] args) {
        try {
            Scanner scanner = new Scanner(new File("example.txt"));
            while(scanner.hasNextLine())
            {
                String line = scanner.nextLine();
                //do stuff
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

word by word

```
public class Main {

    public static void main(String[] args) {
        try {
            Scanner scanner = new Scanner(new File("example.txt"));
            while(scanner.hasNext())
            {
                String line = scanner.next();
                //do stuff
            }
```

```
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

and you can also change the delimeter by using scanner.useDelimeter() method

Section 77.9: Copying a file using InputStream and OutputStream

We can directly copy data from a source to a data sink using a loop. In this example, we are reading data from an InputStream and at the same time, writing to an OutputStream. Once we are done reading and writing, we have to close the resource.

```
public void copy(InputStream source, OutputStream destination) throws IOException {
    try {
        int c;
        while ((c = source.read()) != -1) {
            destination.write(c);
        }
    } finally {
        if (source != null) {
            source.close();
        }
        if (destination != null) {
            destination.close();
        }
    }
}
```

Section 77.10: Reading from a binary file

You can read an a binary file using this piece of code in all recent versions of Java:

Version ≥ Java SE 1.4

```
File file = new File("path_to_the_file");
byte[] data = new byte[(int) file.length()];
DataInputStream stream = new DataInputStream(new FileInputStream(file));
stream.readFully(data);
stream.close();
```

If you are using Java 7 or later, there is a simpler way using the `nio API`:

Version ≥ Java SE 7

```
Path path = Paths.get("path_to_the_file");
byte [] data = Files.readAllBytes(path);
```

Section 77.11: Reading a file using Channel and Buffer

`Channel` uses a `Buffer` to read/write data. A buffer is a fixed sized container where we can write a block of data at once. `Channel` is a quite faster than stream-based I/O.

To read data from a file using `Channel` we need to have the following steps-

1. We need an instance of `FileInputStream`. `FileInputStream`has a method named `getChannel()` which returns a Channel.

2. Call the `getChannel()` method of FileInputStream and acquire Channel.
3. Create a ByteBuffer. ByteBuffer is a fixed size container of bytes.
4. Channel has a read method and we have to provide a ByteBuffer as an argument to this read method. ByteBuffer has two modes - read-only mood and write-only mood. We can change the mode using `flip()` method call. Buffer has a position, limit, and capacity. Once a buffer is created with a fixed size, its limit and capacity are the same as the size and the position starts from zero. While a buffer is written with data, its position gradually increases. Changing mode means, changing the position. To read data from the beginning of a buffer, we have to set the position to zero. flip() method change the position
5. When we call the read method of the `Channel`, it fills up the buffer using data.
6. If we need to read the data from the `ByteBuffer`, we need to flip the buffer to change its mode to write-only to read-only mode and then keep reading data from the buffer.
7. When there is no longer data to read, the `read()` method of channel returns 0 or -1.

```
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;

public class FileChannelRead {

public static void main(String[] args) {

   File inputFile = new File("hello.txt");

   if (!inputFile.exists()) {
    System.out.println("The input file doesn't exit.");
    return;
   }

  try {
   FileInputStream fis = new FileInputStream(inputFile);
   FileChannel fileChannel = fis.getChannel();
   ByteBuffer buffer = ByteBuffer.allocate(1024);

   while (fileChannel.read(buffer) > 0) {
    buffer.flip();
    while (buffer.hasRemaining()) {
     byte b = buffer.get();
     System.out.print((char) b);
    }
    buffer.clear();
   }

   fileChannel.close();
  } catch (IOException e) {
   e.printStackTrace();
  }
 }
}
```

Section 77.12: Adding Directories

To make a new directory from a `File` instance you would need to use one of two methods: `mkdirs()` or `mkdir()`.

- `mkdir()` - Creates the directory named by this abstract pathname. (source)
- `mkdirs()` - Creates the directory named by this abstract pathname, including any necessary but nonexistent parent directories. Note that if this operation fails it may have succeeded in creating some of the necessary

parent directories. (source)

Note: `createNewFile()` will not create a new directory only a file.

```
File singleDir = new File("C:/Users/SomeUser/Desktop/A New Folder/");

File multiDir = new File("C:/Users/SomeUser/Desktop/A New Folder 2/Another Folder/");

// assume that neither "A New Folder" or "A New Folder 2" exist

singleDir.createNewFile(); // will make a new file called "A New Folder.file"
singleDir.mkdir(); // will make the directory
singleDir.mkdirs(); // will make the directory

multiDir.createNewFile(); // will throw a IOException
multiDir.mkdir(); // will not work
multiDir.mkdirs(); // will make the directory
```

Section 77.13: Blocking or redirecting standard output / error

Sometimes a poorly designed 3rd-party library will write unwanted diagnostics to `System.out` or `System.err` streams. The recommended solutions to this would be to either find a better library or (in the case of open source) fix the problem and contribute a patch to the developers.

If the above solutions are not feasible, then you should consider redirecting the streams.

Redirection on the command line

On a UNIX, Linux or MacOSX system can be done from the shell using `>` redirection. For example:

```
$ java -jar app.jar arg1 arg2 > /dev/null 2>&1
$ java -jar app.jar arg1 arg2 > out.log 2> error.log
```

The first one redirects standard output and standard error to "/dev/null", which throws away anything written to those streams. The second of redirects standard output to "out.log" and standard error to "error.log".

(For more information on redirection, refer to the documentation of the command shell you are using. Similar advice applies to Windows.)

Alternatively, you could implement the redirection in a wrapper script or batch file that launches the Java application.

Redirection within a Java application

It is also possible to redired the streams *within* a Java application using `System.setOut()` and `System.setErr()`. For example, the following snippet redirects standard output and standard error to 2 log files:

```
System.setOut(new PrintStream(new FileOutputStream(new File("out.log"))));
System.setErr(new PrintStream(new FileOutputStream(new File("err.log"))));
```

If you want to throw away the output entirely, you can create an output stream that "writes" to an invalid file descriptor. This is functionally equivalent to writing to "/dev/null" on UNIX.

```
System.setOut(new PrintStream(new FileOutputStream(new FileDescriptor())));
System.setErr(new PrintStream(new FileOutputStream(new FileDescriptor())));
```

Caution: be careful how you use `setOut` and `setErr`:

1. The redirection will affect the entire JVM.
2. By doing this, you are taking away the user's ability to redirect the streams from the command line.

Section 77.14: Reading a whole file at once

```
File f = new File(path);
String content = new Scanner(f).useDelimiter("\\Z").next();
```

\Z is the EOF (End of File) Symbol. When set as delimiter the Scanner will read the fill until the EOF Flag is reached.

Section 77.15: Locking

A file can be locked using the `FileChannel` API that can be acquired from Input Output `streams` and `readers`

Example with `streams`

```
    // Open a file stream
    FileInputStream ios = new FileInputStream(filename);
    // get underlying channel
    FileChannel channel = ios.getChannel();

    /*
     * try to lock the file. true means whether the lock is shared or not i.e. multiple processes can
acquire a
     * shared lock (for reading only) Using false with readable channel only will generate an
exception. You should
     * use a writable channel (taken from FileOutputStream) when using false. tryLock will always
return immediately
     */
    FileLock lock = channel.tryLock(0, Long.MAX_VALUE, true);

    if (lock == null) {
        System.out.println("Unable to acquire lock");
    } else {
        System.out.println("Lock acquired successfully");
    }

    // you can also use blocking call which will block until a lock is acquired.
    channel.lock();

    // Once you have completed desired operations of file. release the lock
    if (lock != null) {
        lock.release();
    }

    // close the file stream afterwards
    // Example with reader
    RandomAccessFile randomAccessFile = new RandomAccessFile(filename,  "rw");
    FileChannel channel = randomAccessFile.getChannel();
    //repeat the same steps as above but now you can use shared as true or false as the channel is in
read write mode
```

Section 77.16: Reading a file using BufferedInputStream

Reading file using a `BufferedInputStream` generally faster than `FileInputStream` because it maintains an internal

buffer to store bytes read from the underlying input stream.

```
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.IOException;


public class FileReadingDemo {

    public static void main(String[] args) {
        String source = "hello.txt";

        try (BufferedInputStream bis = new BufferedInputStream(new FileInputStream(source))) {
            byte data;
            while ((data = (byte) bis.read()) != -1) {
                System.out.println((char) data);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

    }
}
```

Section 77.17: Iterate over a directory printing subdirectories in it

```
  public void iterate(final String dirPath) throws IOException {
    final DirectoryStream<Path> paths = Files.newDirectoryStream(Paths.get(dirPath));
    for (final Path path : paths) {
      if (Files.isDirectory(path)) {
        System.out.println(path.getFileName());
      }
    }
  }
```

Section 77.18: Writing a file using Channel and Buffer

To write data to a file using `Channel` we need to have the following steps:

1. First, we need to get an object of `FileOutputStream`
2. Acquire `FileChannel` calling the `getChannel()` method from the `FileOutputStream`
3. Create a `ByteBuffer` and then fill it with data
4. Then we have to call the `flip()` method of the `ByteBuffer` and pass it as an argument of the `write()` method of the `FileChannel`
5. Once we are done writing, we have to close the resource

```
import java.io.*;
import java.nio.*;
public class FileChannelWrite {

 public static void main(String[] args) {

  File outputFile = new File("hello.txt");
  String text = "I love Bangladesh.";

  try {
   FileOutputStream fos = new FileOutputStream(outputFile);
```

```
    FileChannel fileChannel = fos.getChannel();
    byte[] bytes = text.getBytes();
    ByteBuffer buffer = ByteBuffer.wrap(bytes);
    fileChannel.write(buffer);
    fileChannel.close();
   } catch (java.io.IOException e) {
    e.printStackTrace();
   }
  }
}
```

Section 77.19: Writing a file using PrintStream

We can use `PrintStream` class to write a file. It has several methods that let you print any data type values. `println()` method appends a new line. Once we are done printing, we have to flush the `PrintStream`.

```
import java.io.FileNotFoundException;
import java.io.PrintStream;
import java.time.LocalDate;

public class FileWritingDemo {
    public static void main(String[] args) {
        String destination = "file1.txt";

        try(PrintStream ps = new PrintStream(destination)){
            ps.println("Stackoverflow documentation seems fun.");
            ps.println();
            ps.println("I love Java!");
            ps.printf("Today is: %1$tm/%1$td/%1$tY", LocalDate.now());

            ps.flush();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }

    }
}
```

Section 77.20: Iterating over a directory and filter by file extension

```
    public void iterateAndFilter() throws IOException {
        Path dir = Paths.get("C:/foo/bar");
        PathMatcher imageFileMatcher =
            FileSystems.getDefault().getPathMatcher(
                "regex:.*(?i:jpg|jpeg|png|gif|bmp|jpe|jfif)");

        try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir,
                entry -> imageFileMatcher.matches(entry.getFileName()))) {

            for (Path path : stream) {
                System.out.println(path.getFileName());
            }
        }
    }
```

Section 77.21: Accessing the contents of a ZIP file

The FileSystem API of Java 7 allows to read and add entries from or to a Zip file using the Java NIO file API in the same way as operating on any other filesystem.

The FileSystem is a resource that should be properly closed after use, therefore the try-with-resources block should be used.

Reading from an existing file

```
Path pathToZip = Paths.get("path/to/file.zip");
try(FileSystem zipFs = FileSystems.newFileSystem(pathToZip, null)) {
  Path root = zipFs.getPath("/");
  ... //access the content of the zip file same as ordinary files
} catch(IOException ex) {
  ex.printStackTrace();
}
```

Creating a new file

```
Map<String, String> env = new HashMap<>();
env.put("create", "true"); //required for creating a new zip file
env.put("encoding", "UTF-8"); //optional: default is UTF-8
URI uri = URI.create("jar:file:/path/to/file.zip");
try (FileSystem zipfs = FileSystems.newFileSystem(uri, env)) {
  Path newFile = zipFs.getPath("/newFile.txt");
  //writing to file
  Files.write(newFile, "Hello world".getBytes());
} catch(IOException ex) {
  ex.printStackTrace();
}
```

Chapter 78: Scanner

Parameter	Details
Source	Source could be either one of String, File or any kind of InputStream

Section 78.1: General Pattern that does most commonly asked about tasks

The following is how to properly use the `java.util.Scanner` class to interactively read user input from `System.in` correctly(sometimes referred to as `stdin`, especially in C, C++ and other languages as well as in Unix and Linux). It idiomatically demonstrates the most common things that are requested to be done.

```
package com.stackoverflow.scanner;

import javax.annotation.Nonnull;
import java.math.BigInteger;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.*;
import java.util.regex.Pattern;

import static java.lang.String.format;

public class ScannerExample
{
    private static final Set<String> EXIT_COMMANDS;
    private static final Set<String> HELP_COMMANDS;
    private static final Pattern DATE_PATTERN;
    private static final String HELP_MESSAGE;

    static
    {
        final SortedSet<String> ecmds = new TreeSet<String>(String.CASE_INSENSITIVE_ORDER);
        ecmds.addAll(Arrays.asList("exit", "done", "quit", "end", "fino"));
        EXIT_COMMANDS = Collections.unmodifiableSortedSet(ecmds);
        final SortedSet<String> hcmds = new TreeSet<String>(String.CASE_INSENSITIVE_ORDER);
        hcmds.addAll(Arrays.asList("help", "helpi", "?"));
        HELP_COMMANDS = Collections.unmodifiableSet(hcmds);
        DATE_PATTERN = Pattern.compile("\\d{4}([-\\/])\\d{2}\\1\\d{2}"); //
http://regex101.com/r/xB8dR3/1
        HELP_MESSAGE = format("Please enter some data or enter one of the following commands to
exit %s", EXIT_COMMANDS);
    }

    /**
     * Using exceptions to control execution flow is always bad.
     * That is why this is encapsulated in a method, this is done this
     * way specifically so as not to introduce any external libraries
     * so that this is a completely self contained example.
     * @param s possible url
     * @return true if s represents a valid url, false otherwise
     */
    private static boolean isValidURL(@Nonnull final String s)
    {
        try { new URL(s); return true; }
        catch (final MalformedURLException e) { return false; }
    }

    private static void output(@Nonnull final String format, @Nonnull final Object... args)
```

```
    {
        System.out.println(format(format, args));
    }

    public static void main(final String[] args)
    {
        final Scanner sis = new Scanner(System.in);
        output(HELP_MESSAGE);
        while (sis.hasNext())
        {
            if (sis.hasNextInt())
            {
                final int next = sis.nextInt();
                output("You entered an Integer = %d", next);
            }
            else if (sis.hasNextLong())
            {
                final long next = sis.nextLong();
                output("You entered a Long = %d", next);
            }
            else if (sis.hasNextDouble())
            {
                final double next = sis.nextDouble();
                output("You entered a Double = %f", next);
            }
            else if (sis.hasNext("\\d+"))
            {
                final BigInteger next = sis.nextBigInteger();
                output("You entered a BigInteger = %s", next);
            }
            else if (sis.hasNextBoolean())
            {
                final boolean next = sis.nextBoolean();
                output("You entered a Boolean representation = %s", next);
            }
            else if (sis.hasNext(DATE_PATTERN))
            {
                final String next = sis.next(DATE_PATTERN);
                output("You entered a Date representation = %s", next);
            }
            else // unclassified
            {
                final String next = sis.next();
                if (isValidURL(next))
                {
                    output("You entered a valid URL = %s", next);
                }
                else
                {
                    if (EXIT_COMMANDS.contains(next))
                    {
                        output("Exit command %s issued, exiting!", next);
                        break;
                    }
                    else if (HELP_COMMANDS.contains(next)) { output(HELP_MESSAGE); }
                    else { output("You entered an unclassified String = %s", next); }
                }
            }
        }
        /*
           This will close the underlying Readable, in this case System.in, and free those resources.
           You will not be to read from System.in anymore after this you call .close().
```

```
            If you wanted to use System.in for something else, then don't close the Scanner.
        */
        sis.close();
        System.exit(0);
    }
}
```

Section 78.2: Using custom delimiters

You can use custom delimiters (regular expressions) with Scanner, with `.useDelimiter(",")`, to determine how the input is read. This works similarly to `String.split(...)`. For example, you can use `Scanner` to read from a list of comma separated values in a String:

```
Scanner scanner = null;
try{
    scanner = new Scanner("i,like,unicorns").useDelimiter(",");;
    while(scanner.hasNext()){
        System.out.println(scanner.next());
    }
}catch(Exception e){
    e.printStackTrace();
}finally{
    if (scanner != null)
        scanner.close();
}
```

This will allow you to read every element in the input individually. Note that you should **not** use this to parse CSV data, instead, use a proper CSV parser library, see CSV parser for Java for other possibilities.

Section 78.3: Reading system input using Scanner

```
Scanner scanner = new Scanner(System.in); //Scanner obj to read System input
String inputTaken = new String();
while (true) {
    String input = scanner.nextLine(); // reading one line of input
    if (input.matches("\\s+"))         // if it matches spaces/tabs, stop reading
        break;
    inputTaken += input + " ";
}
System.out.println(inputTaken);
```

The scanner object is initialized to read input from keyboard. So for the below input from keyboar, it'll produce the output as `Reading from keyboard`

```
Reading
from
keyboard
//space
```

Section 78.4: Reading file input using Scanner

```
Scanner scanner = null;
try {
    scanner = new Scanner(new File("Names.txt"));
    while (scanner.hasNext()) {
        System.out.println(scanner.nextLine());
```

```
    }
} catch (Exception e) {
    System.err.println("Exception occurred!");
} finally {
    if (scanner != null)
        scanner.close();
}
```

Here a `Scanner` object is created by passing a `File` object containing the name of a text file as input. This text file will be opened by the File object and read in by the scanner object in the following lines. `scanner.hasNext()` will check to see if there is a next line of data in the text file. Combining that with a **`while`** loop will allow you to iterate through every line of data in the `Names.txt` file. To retrieve the data itself, we can use methods such as `nextLine()`,`nextInt()`,`nextBoolean()`, etc. In the example above, `scanner.nextLine()`is used. `nextLine()` refers to the following line in a text file, and combining it with a `scanner` object allows you to print the contents of the line. To close a scanner object, you would use `.close()`.

Using try with resources (from Java 7 onwards), the above mentioned code can be written elegantly as below.

```
try (Scanner scanner = new Scanner(new File("Names.txt"))) {
    while (scanner.hasNext()) {
        System.out.println(scanner.nextLine());
    }
} catch (Exception e) {
    System.err.println("Exception occurred!");
}
```

Section 78.5: Read the entire input as a String using Scanner

You can use `Scanner` to read all of the text in the input as a String, by using `\Z` (entire input) as the delimiter. For example, this can be used to read all text in a text file in one line:

```
String content = new Scanner(new File("filename")).useDelimiter("\\Z").next();
System.out.println(content);
```

Remember that you'll have to close the Scanner, as well as catch the `IoException` this may throw, as described in the example Reading file input using Scanner.

Section 78.6: Carefully Closing a Scanner

it can happen that you use a scanner with the System.in as parameter for the constructor, then you need to be aware that closing the scanner will close the InputStream too giving as next that every try to read the input on that (Or any other scanner object) will throw an `java.util.NoSuchElementException` or an `java.lang.IllegalStateException`

example:

```
    Scanner sc1 = new Scanner(System.in);
    Scanner sc2 = new Scanner(System.in);
    int x1 = sc1.nextInt();
    sc1.close();
    // java.util.NoSuchElementException
    int x2 = sc2.nextInt();
    // java.lang.IllegalStateException
    x2 = sc1.nextInt();
```

Section 78.7: Read an int from the command line

```
import java.util.Scanner;

Scanner s = new Scanner(System.in);
int number = s.nextInt();
```

If you want to read an int from the command line, just use this snippet. First of all, you have to create a Scanner object, that listens to System.in, which is by default the Command Line, when you start the program from the command line. After that, with the help of the Scanner object, you read the first int that the user passes into the command line and store it in the variable number. Now you can do whatever you want with that stored int.

Chapter 79: Interfaces

An *interface* is a reference type, similar to a class, which can be declared by using **`interface`** keyword. Interfaces can contain only constants, method signatures, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods. Like abstract classes, Interfaces cannot be instantiated—they can only be implemented by classes or extended by other interfaces. Interface is a common way to achieve full abstraction in Java.

Section 79.1: Implementing multiple interfaces

A Java class can implement multiple interfaces.

```
public interface NoiseMaker {
    String noise = "Making Noise"; // interface variables are public static final by default

    String makeNoise(); //interface methods are public abstract by default
}

public interface FoodEater {
    void eat(Food food);
}

public class Cat implements NoiseMaker, FoodEater {
    @Override
    public String makeNoise() {
        return "meow";
    }

    @Override
    public void eat(Food food) {
        System.out.println("meows appreciatively");
    }
}
```

Notice how the `Cat` class **must** implement the inherited **`abstract`** methods in both the interfaces. Furthermore, notice how a class can practically implement as many interfaces as needed (there is a limit of **65,535** due to JVM Limitation).

```
NoiseMaker noiseMaker = new Cat(); // Valid
FoodEater foodEater = new Cat(); // Valid
Cat cat = new Cat(); // valid

Cat invalid1 = new NoiseMaker(); // Invalid
Cat invalid2 = new FoodEater(); // Invalid
```

Note:

1. All variables declared in an interface are **`public static final`**
2. All methods declared in an interface methods are **`public abstract`** (This statement is valid only through Java 7. From Java 8, you are allowed to have methods in an interface, which need not be abstract; such methods are known as default methods)
3. Interfaces cannot be declared as **`final`**
4. If more than one interface declares a method that has identical signature, then effectively it is treated as only one method and you cannot distinguish from which interface method is implemented
5. A corresponding **InterfaceName.class** file would be generated for each interface, upon compilation

Section 79.2: Declaring and Implementing an Interface

Declaration of an interface using the **`interface`** keyword:

```
public interface Animal {
    String getSound(); // Interface methods are public by default
}
```

Override Annotation

```
@Override
public String getSound() {
    // Code goes here...
}
```

This forces the compiler to check that we are overriding and prevents the program from defining a new method or messing up the method signature.

Interfaces are implemented using the `implements` keyword.

```
public class Cat implements Animal {

    @Override
    public String getSound() {
        return "meow";
    }
}

public class Dog implements Animal {

    @Override
    public String getSound() {
        return "woof";
    }
}
```

In the example, classes `Cat` and `Dog` **must** define the `getSound()` method as methods of an interface are inherently abstract (with the exception of default methods).

Using the interfaces

```
Animal cat = new Cat();
Animal dog = new Dog();

System.out.println(cat.getSound()); // prints "meow"
System.out.println(dog.getSound()); // prints "woof"
```

Section 79.3: Extending an interface

An interface can extend another interface via the **`extends`** keyword.

```
public interface BasicResourceService {
    Resource getResource();
}

public interface ExtendedResourceService extends BasicResourceService {
    void updateResource(Resource resource);
```

```
}
```

Now a class implementing `ExtendedResourceService` will need to implement both `getResource()` and `updateResource()`.

Extending multiple interfaces

Unlike classes, the **`extends`** keyword can be used to extend multiple interfaces (Separated by commas) allowing for combinations of interfaces into a new interface

```
public interface BasicResourceService {
    Resource getResource();
}

public interface AlternateResourceService {
    Resource getAlternateResource();
}

public interface ExtendedResourceService extends BasicResourceService, AlternateResourceService {
    Resource updateResource(Resource resource);
}
```

In this case a class implementing `ExtendedResourceService` will need to implement `getResource()`, `getAlternateResource()`, and `updateResource()`.

Section 79.4: Usefulness of interfaces

Interfaces can be extremely helpful in many cases. For example, say you had a list of animals and you wanted to loop through the list, each printing the sound they make.

```
{cat, dog, bird}
```

One way to do this would be to use interfaces. This would allow for the same method to be called on all of the classes

```
public interface Animal {
    public String getSound();
}
```

Any class that **`implements`** `Animal` also must have a `getSound()` method in them, yet they can all have different implementations

```
public class Dog implements Animal {
    public String getSound() {
        return "Woof";
    }
}

public class Cat implements Animal {
    public String getSound() {
        return "Meow";
    }
}

public class Bird implements Animal{
    public String getSound() {
        return "Chirp";
```

```
    }
}
```

We now have three different classes, each of which has a `getSound()` method. Because all of these classes `implement` the `Animal` interface, which declares the `getSound()` method, any instance of an `Animal` can have `getSound()` called on it

```
Animal dog = new Dog();
Animal cat = new Cat();
Animal bird = new Bird();

dog.getSound(); // "Woof"
cat.getSound(); // "Meow"
bird.getSound(); // "Chirp"
```

Because each of these is an `Animal`, we could even put the animals in a list, loop through them, and print out their sounds

```
Animal[] animals = { new Dog(), new Cat(), new Bird() };
for (Animal animal : animals) {
    System.out.println(animal.getSound());
}
```

Because the order of the array is `Dog`, `Cat`, and then `Bird`, *"Woof Meow Chirp"* will be printed to the console.

Interfaces can also be used as the return value for functions. For example, returning a `Dog` if the input is *"dog"*, `Cat` if the input is *"cat"*, and `Bird` if it is *"bird"*, and then printing the sound of that animal could be done using

```
public Animal getAnimalByName(String name) {
    switch(name.toLowerCase()) {
        case "dog":
            return new Dog();
        case "cat":
            return new Cat();
        case "bird":
            return new Bird();
        default:
            return null;
    }
}

public String getAnimalSoundByName(String name){
    Animal animal = getAnimalByName(name);
    if (animal == null) {
        return null;
    } else {
        return animal.getSound();
    }
}

String dogSound = getAnimalSoundByName("dog"); // "Woof"
String catSound = getAnimalSoundByName("cat"); // "Meow"
String birdSound = getAnimalSoundByName("bird"); // "Chirp"
String lightbulbSound = getAnimalSoundByName("lightbulb"); // null
```

Interfaces are also useful for extensibility, because if you want to add a new type of `Animal`, you wouldn't need to change anything with the operations you perform on them.

Section 79.5: Default methods

Introduced in Java 8, default methods are a way of specifying an implementation inside an interface. This could be used to avoid the typical "Base" or "Abstract" class by providing a partial implementation of an interface, and restricting the subclasses hierarchy.

Observer pattern implementation

For example, it's possible to implement the Observer-Listener pattern directly into the interface, providing more flexibility to the implementing classes.

```
interface Observer {
    void onAction(String a);
}

interface Observable{
    public abstract List<Observer> getObservers();

    public default void addObserver(Observer o){
        getObservers().add(o);
    }

    public default void notify(String something ){
        for( Observer l : getObservers() ){
            l.onAction(something);
        }
    }
}
```

Now, any class can be made "Observable" just by implementing the Observable interface, while being free to be part of a different class hierarchy.

```
abstract class Worker{
    public abstract void work();
}

public class MyWorker extends Worker implements Observable {

    private List<Observer> myObservers = new ArrayList<Observer>();

    @Override
    public List<Observer> getObservers() {
        return myObservers;
    }

    @Override
    public void work(){
        notify("Started work");

        // Code goes here...

        notify("Completed work");
    }

    public static void main(String[] args) {
        MyWorker w = new MyWorker();

        w.addListener(new Observer() {
            @Override
```

```
            public void onAction(String a) {
                System.out.println(a + " (" + new Date() + ")");
            }
        });

        w.work();
    }
}
```

Diamond problem

The compiler in Java 8 is aware of the diamond problem which is caused when a class is implementing interfaces containing a method with the same signature.

In order to solve it, an implementing class must override the shared method and provide its own implementation.

```
interface InterfaceA {
    public default String getName(){
        return "a";
    }
}

interface InterfaceB {
    public default String getName(){
        return "b";
    }
}

public class ImpClass implements InterfaceA, InterfaceB {

    @Override
    public String getName() {
        //Must provide its own implementation
        return InterfaceA.super.getName() + InterfaceB.super.getName();
    }

    public static void main(String[] args) {
        ImpClass c = new ImpClass();

        System.out.println( c.getName() );                  // Prints "ab"
        System.out.println( ((InterfaceA)c).getName() );    // Prints "ab"
        System.out.println( ((InterfaceB)c).getName() );    // Prints "ab"
    }
}
```

There's still the issue of having methods with the same name and parameters with different return types, which will not compile.

Use default methods to resolve compatibility issues

The default method implementations come in very handy if a method is added to an interface in an existing system where the interfaces is used by several classes.

To avoid breaking up the entire system, you can provide a default method implementation when you add a method to an interface. This way, the system will still compile and the actual implementations can be done step by step.

For more information, see the Default Methods topic.

Section 79.6: Modifiers in Interfaces

The Oracle Java Style Guide states:

> Modifiers should not be written out when they are implicit.

(See Modifiers in Oracle Official Code Standard for the context and a link to the actual Oracle document.)

This style guidance applies particularly to interfaces. Let's consider the following code snippet:

```
interface I {
    public static final int VARIABLE = 0;

    public abstract void method();

    public static void staticMethod() { ... }
    public default void defaultMethod() { ... }
}
```

Variables

All interface variables are implicitly *constants* with implicit **`public`** (accessible for all), **`static`** (are accessible by interface name) and **`final`** (must be initialized during declaration) modifiers:

```
public static final int VARIABLE = 0;
```

Methods

1. All methods which *don't provide implementation* are implicitly **`public`** and **`abstract`**.

```
public abstract void method();
```

Version ≥ Java SE 8

2. All methods with **`static`** or **`default`** modifier *must provide implementation* and are implicitly **`public`**.

```
public static void staticMethod() { ... }
```

After all of the above changes have been applied, we will get the following:

```
interface I {
    int VARIABLE = 0;

    void method();

    static void staticMethod() { ... }
    default void defaultMethod() { ... }
}
```

Section 79.7: Using Interfaces with Generics

Let's say you want to define an interface that allows publishing / consuming data to and from different types of channels (e.g. AMQP, JMS, etc), but you want to be able to switch out the implementation details ...

Let's define a basic IO interface that can be re-used across multiple implementations:

```
public interface IO<IncomingType, OutgoingType> {

    void publish(OutgoingType data);
    IncomingType consume();
    IncomingType RPCSubmit(OutgoingType data);

}
```

Now I can instantiate that interface, but since we don't have default implementations for those methods, it'll need an implementation when we instantiate it:

```
    IO<String, String> mockIO = new IO<String, String>() {

        private String channel = "somechannel";

        @Override
        public void publish(String data) {
            System.out.println("Publishing " + data + " to " + channel);
        }

        @Override
        public String consume() {
            System.out.println("Consuming from " + channel);
            return "some useful data";
        }

        @Override
        public String RPCSubmit(String data) {
            return "received " + data + " just now ";
        }

    };

    mockIO.consume(); // prints: Consuming from somechannel
    mockIO.publish("TestData"); // Publishing TestData to somechannel
    System.out.println(mockIO.RPCSubmit("TestData")); // received TestData just now
```

We can also do something more useful with that interface, let's say we want to use it to wrap some basic RabbitMQ functions:

```
public class RabbitMQ implements IO<String, String> {

    private String exchange;
    private String queue;

    public RabbitMQ(String exchange, String queue){
        this.exchange = exchange;
        this.queue = queue;
    }

    @Override
    public void publish(String data) {
        rabbit.basicPublish(exchange, queue, data.getBytes());
    }

    @Override
    public String consume() {
        return rabbit.basicConsume(exchange, queue);
    }
```

```
    @Override
    public String RPCSubmit(String data) {
        return rabbit.rpcPublish(exchange, queue, data);
    }

}
```

Let's say I want to use this IO interface now as a way to count visits to my website since my last system restart and then be able to display the total number of visits - you can do something like this:

```
import java.util.concurrent.atomic.AtomicLong;

public class VisitCounter implements IO<Long, Integer> {

    private static AtomicLong websiteCounter = new AtomicLong(0);

    @Override
    public void publish(Integer count) {
        websiteCounter.addAndGet(count);
    }

    @Override
    public Long consume() {
        return websiteCounter.get();
    }

    @Override
    public Long RPCSubmit(Integer count) {
        return websiteCounter.addAndGet(count);
    }

}
```

Now let's use the VisitCounter:

```
    VisitCounter counter = new VisitCounter();

    // just had 4 visits, yay
    counter.publish(4);
    // just had another visit, yay
    counter.publish(1);

    // get data for stats counter
    System.out.println(counter.consume()); // prints 5

    // show data for stats counter page, but include that as a page view
    System.out.println(counter.RPCSubmit(1)); // prints 6
```

When implementing multiple interfaces, you can't implement the same interface twice. That also applies to generic interfaces. Thus, the following code is invalid, and will result in a compile error:

```
interface Printer<T> {
    void print(T value);
}

// Invalid!
class SystemPrinter implements Printer<Double>, Printer<Integer> {
    @Override public void print(Double d){ System.out.println("Decimal: " + d); }
    @Override public void print(Integer i){ System.out.println("Discrete: " + i); }
```

```
}
```

Section 79.8: Strengthen bounded type parameters

Bounded type parameters allow you to set restrictions on generic type arguments:

```
class SomeClass {

}

class Demo<T extends SomeClass> {

}
```

But a type parameter can only bind to a single class type.

An interface type can be bound to a type that already had a binding. This is achieved using the `&` symbol:

```
interface SomeInterface {

}

class GenericClass<T extends SomeClass & SomeInterface> {

}
```

This strengthens the bind, potentially requiring type arguments to derive from multiple types.

Multiple interface types can be bound to a type parameter:

```
class Demo<T extends SomeClass & FirstInterface & SecondInterface> {

}
```

But should be used with caution. Multiple interface bindings is usually a sign of a code smell, suggesting that a new type should be created which acts as an adapter for the other types:

```
interface NewInterface extends FirstInterface, SecondInterface {

}

class Demo<T extends SomeClass & NewInterface> {

}
```

Section 79.9: Implementing interfaces in an abstract class

A method defined in an **`interface`** is by default **`public abstract`**. When an **`abstract class`** implements an **`interface`**, any methods which are defined in the **`interface`** do not have to be implemented by the **`abstract class`**. This is because a **`class`** that is declared **`abstract`** can contain abstract method declarations. It is therefore the responsibility of the first concrete sub-class to implement any **`abstract`** methods inherited from any interfaces and/or the **`abstract class`**.

```
public interface NoiseMaker {
    void makeNoise();
}
```

```java
public abstract class Animal implements NoiseMaker {
    //Does not need to declare or implement makeNoise()
    public abstract void eat();
}

//Because Dog is concrete, it must define both makeNoise() and eat()
public class Dog extends Animal {
    @Override
    public void makeNoise() {
        System.out.println("Borf borf");
    }

    @Override
    public void eat() {
        System.out.println("Dog eats some kibble.");
    }
}
```

From Java 8 onward it is possible for an **`interface`** to declare **`default`** implementations of methods which means the method won't be **`abstract`**, therefore any concrete sub-classes will not be forced to implement the method but will inherit the **`default`** implementation unless overridden.

Chapter 80: Regular Expressions

A regular expression is a special sequence of characters that helps in matching or finding other strings or sets of strings, using a specialized syntax held in a pattern. Java has support for regular expression usage through the `java.util.regex` package. This topic is to introduce and help developers understand more with examples on how Regular Expressions must be used in Java.

Section 80.1: Using capture groups

If you need to extract a part of string from the input string, we can use **capture groups** of regex.

For this example, we'll start with a simple phone number regex:

```
\d{3}-\d{3}-\d{4}
```

If parentheses are added to the regex, each set of parentheses is considered a *capturing group*. In this case, we are using what are called numbered capture groups:

```
(\d{3})-(\d{3})-(\d{4})
^-----^ ^-----^ ^-----^
Group 1 Group 2 Group 3
```

Before we can use it in Java, we must not forget to follow the rules of Strings, escaping the backslashes, resulting in the following pattern:

```
"(\\d{3})-(\\d{3})-(\\d{4})"
```

We first need to compile the regex pattern to make a `Pattern` and then we need a `Matcher` to match our input string with the pattern:

```
Pattern phonePattern = Pattern.compile("(\\d{3})-(\\d{3})-(\\d{4})");
Matcher phoneMatcher = phonePattern.matcher("abcd800-555-1234wxyz");
```

Next, the Matcher needs to find the first subsequence that matches the regex:

```
phoneMatcher.find();
```

Now, using the group method, we can extract the data from the string:

```
String number = phoneMatcher.group(0); //"800-555-1234" (Group 0 is everything the regex matched)
String aCode = phoneMatcher.group(1); //"800"
String threeDigit = phoneMatcher.group(2); //"555"
String fourDigit = phoneMatcher.group(3); //"1234"
```

Note: `Matcher.group()` can be used in place of `Matcher.group(0)`.

`Version ≥ Java SE 7`

Java 7 introduced named capture groups. Named capture groups function the same as numbered capture groups (but with a name instead of a number), although there are slight syntax changes. Using named capture groups improves readability.

We can alter the above code to use named groups:

```
(?\d{3})-(\d{3})-(\d{4})
^----------------^ ^-----^ ^-----^
AreaCode           Group 2 Group 3
```

To get the contents of "AreaCode", we can instead use:

```
String aCode = phoneMatcher.group("AreaCode"); //"800"
```

Section 80.2: Using regex with custom behaviour by compiling the Pattern with flags

A `Pattern` can be compiled with flags, if the regex is used as a literal `String`, use inline modifiers:

```
Pattern pattern = Pattern.compile("foo.", Pattern.CASE_INSENSITIVE | Pattern.DOTALL);
pattern.matcher("FOO\n").matches(); // Is true.

/* Had the regex not been compiled case insensitively and singlelined,
 * it would fail because FOO does not match /foo/ and \n (newline)
 * does not match /./.
 */

Pattern anotherPattern = Pattern.compile("(?si)foo");
anotherPattern.matcher("FOO\n").matches(); // Is true.

"foOt".replaceAll("(?si)foo", "ca"); // Returns "cat".
```

Section 80.3: Escape Characters

Generally

To use regular expression specific characters (`?+|` etc.) in their literal meaning they need to be escaped. In common regular expression this is done by a backslash `\`. However, as it has a special meaning in Java Strings, you have to use a double backslash `\\`.

These two examples will not work:

```
"???".replaceAll ("?", "!"); //java.util.regex.PatternSyntaxException
"???".replaceAll ("\?", "!"); //Invalid escape sequence
```

This example works

```
"???".replaceAll ("\\?", "!"); //"!!!"
```

Splitting a Pipe Delimited String

This does not return the expected result:

```
"a|b".split ("|"); // [a, |, b]
```

This returns the expected result:

```
"a|b".split ("\\|"); // [a, b]
```

Escaping backslash `\`

This will give an error:

```
"\\".matches("\\"); // PatternSyntaxException
"\\".matches("\\\"); // Syntax Error
```

This works:

```
"\\".matches("\\\\"); // true
```

Section 80.4: Not matching a given string

To match something that does *not* contain a given string, one can use negative lookahead:

Regex syntax: `(?!string-to-not-match)`

Example:

```
//not matching "popcorn"
String regexString = "^(?!popcorn).*$";
System.out.println("[popcorn] " + ("popcorn".matches(regexString) ? "matched!" : "nope!"));
System.out.println("[unicorn] " + ("unicorn".matches(regexString) ? "matched!" : "nope!"));
```

Output:

```
[popcorn] nope!
[unicorn] matched!
```

Section 80.5: Matching with a regex literal

If you need to match characters that are a part of the regular expression syntax you can mark all or part of the pattern as a regex literal.

`\Q` marks the beginning of the regex literal. `\E` marks the end of the regex literal.

```
// the following throws a PatternSyntaxException because of the un-closed bracket
"[123".matches("[123");

// wrapping the bracket in \Q and \E allows the pattern to match as you would expect.
"[123".matches("\\Q[\\E123"); // returns true
```

An easier way of doing it without having to remember the `\Q` and `\E` escape sequences is to use `Pattern.quote()`

```
"[123".matches(Pattern.quote("[") + "123"); // returns true
```

Section 80.6: Matching a backslash

If you want to match a backslash in your regular expression, you'll have to escape it.

Backslash is an escape character in regular expressions. You can use '\\' to refer to a single backslash in a regular expression.

However, backslash is *also* an escape character in Java literal strings. To make a regular expression from a string literal, you have to escape each of *its* backslashes. In a string literal '\\\\' can be used to create a regular expression

with '\\', which in turn can match '\'.

For example, consider matching strings like "C:\dir\myfile.txt". A regular expression `([A-Za-z]):\\(.*)` will match, and provide the drive letter as a capturing group. Note the doubled backslash.

To express that pattern in a Java string literal, each of the backslashes in the regular expression needs to be escaped.

```
String path = "C:\\dir\\myfile.txt";
System.out.println( "Local path: " + path ); // "C:\dir\myfile.txt"

String regex = "([A-Za-z]):\\\\.*"; // Four to match one
System.out.println("Regex:      " + regex ); // "([A-Za-z]):\\(.*)"

Pattern pattern = Pattern.compile( regex );
Matcher matcher = pattern.matcher( path );
if ( matcher.matches()) {
    System.out.println( "This path is on drive " + matcher.group( 1 ) + ":.");
    // This path is on drive C:.
}
```

If you want to match *two* backslashes, you'll find yourself using eight in a literal string, to represent four in the regular expression, to match two.

```
String path = "\\\\myhost\\share\\myfile.txt";
System.out.println( "UNC path: " + path ); // \\myhost\share\myfile.txt"

String regex = "\\\\\\\\(.+?)\\\\(.*)"; // Eight to match two
System.out.println("Regex:    " + regex ); // \\\\(.+?)\\(.*)

Pattern pattern = Pattern.compile( regex );
Matcher matcher = pattern.matcher( path );

if ( matcher.matches()) {
    System.out.println( "This path is on host '" + matcher.group( 1 ) + "'.");
    // This path is on host 'myhost'.
}
```

Chapter 81: Comparable and Comparator

Section 81.1: Sorting a List using Comparable<T> or a Comparator<T>

Say we are working on a class representing a Person by their first and last names. We have created a basic class to do this and implemented proper `equals` and `hashCode` methods.

```
public class Person {

    private final String lastName; //invariant - nonnull
    private final String firstName; //invariant - nonnull

    public Person(String firstName, String lastName){
        this.firstName = firstName != null ? firstName : "";
        this.lastName = lastName != null ? lastName : "";
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public String toString() {
        return lastName + ", " + firstName;
    }

    @Override
    public boolean equals(Object o) {
        if (! (o instanceof Person)) return false;
        Person p = (Person)o;
        return firstName.equals(p.firstName) && lastName.equals(p.lastName);
    }

    @Override
    public int hashCode() {
        return Objects.hash(firstName, lastName);
    }
}
```

Now we would like to sort a list of `Person` objects by their name, such as in the following scenario:

```
public static void main(String[] args) {
    List<Person> people = Arrays.asList(new Person("John", "Doe"),
                                        new Person("Bob", "Dole"),
                                        new Person("Ronald", "McDonald"),
                                        new Person("Alice", "McDonald"),
                                        new Person("Jill", "Doe"));
    Collections.sort(people); //This currently won't work.
}
```

Unfortunately, as marked, the above currently won't compile. `Collections.sort(..)` only knows how to sort a list if the elements in that list are comparable, or a custom method of comparison is given.

If you were asked to sort the following list : `1,3,5,4,2`, you'd have no problem saying the answer is `1,2,3,4,5`. This

is because Integers (both in Java and mathematically) have a *natural ordering*, a standard, default comparison base ordering. To give our Person class a natural ordering, we implement `Comparable<Person>`, which requires implementing the method `compareTo(Person p)`:

```
public class Person implements Comparable<Person> {

    private final String lastName; //invariant - nonnull
    private final String firstName; //invariant - nonnull

    public Person(String firstName, String lastName) {
        this.firstName = firstName != null ? firstName : "";
        this.lastName = lastName != null ? lastName : "";
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public String toString() {
        return lastName + ", " + firstName;
    }

    @Override
    public boolean equals(Object o) {
        if (! (o instanceof Person)) return false;
        Person p = (Person)o;
        return firstName.equals(p.firstName) && lastName.equals(p.lastName);
    }

    @Override
    public int hashCode() {
        return Objects.hash(firstName, lastName);
    }

    @Override
    public int compareTo(Person other) {
        // If this' lastName and other's lastName are not comparably equivalent,
        // Compare this to other by comparing their last names.
        // Otherwise, compare this to other by comparing their first names
        int lastNameCompare = lastName.compareTo(other.lastName);
        if (lastNameCompare != 0) {
            return lastNameCompare;
        } else {
            return firstName.compareTo(other.firstName);
        }
    }
}
```

Now, the main method given will function correctly

```
public static void main(String[] args) {
    List<Person> people = Arrays.asList(new Person("John", "Doe"),
                                        new Person("Bob", "Dole"),
                                        new Person("Ronald", "McDonald"),
                                        new Person("Alice", "McDonald"),
                                        new Person("Jill", "Doe"));
    Collections.sort(people); //Now functions correctly
```

```
    //people is now sorted by last name, then first name:
    // --> Jill Doe, John Doe, Bob Dole, Alice McDonald, Ronald McDonald
}
```

If, however, you either do not want or are unable to modify class `Person`, you can provide a custom `Comparator<T>` that handles the comparison of any two `Person` objects. If you were asked to sort the following list: `circle, square, rectangle, triangle, hexagon` you could not, but if you were asked to sort that list *based on the number of corners*, you could. Just so, providing a comparator instructs Java how to compare two normally not comparable objects.

```
public class PersonComparator implements Comparator<Person> {

    public int compare(Person p1, Person p2) {
        // If p1's lastName and p2's lastName are not comparably equivalent,
        // Compare p1 to p2 by comparing their last names.
        // Otherwise, compare p1 to p2 by comparing their first names
        if (p1.getLastName().compareTo(p2.getLastName()) != 0) {
            return p1.getLastName().compareTo(p2.getLastName());
        } else {
            return p1.getFirstName().compareTo(p2.getFirstName());
        }
    }
}

//Assume the first version of Person (that does not implement Comparable) is used here
public static void main(String[] args) {
    List<Person> people = Arrays.asList(new Person("John", "Doe"),
                                        new Person("Bob", "Dole"),
                                        new Person("Ronald", "McDonald"),
                                        new Person("Alice", "McDonald"),
                                        new Person("Jill", "Doe"));
    Collections.sort(people); //Illegal, Person doesn't implement Comparable.
    Collections.sort(people, new PersonComparator()); //Legal

    //people is now sorted by last name, then first name:
    // --> Jill Doe, John Doe, Bob Dole, Alice McDonald, Ronald McDonald
}
```

Comparators can also be created/used as an anonymous inner class

```
//Assume the first version of Person (that does not implement Comparable) is used here
public static void main(String[] args) {
    List<Person> people = Arrays.asList(new Person("John", "Doe"),
                                        new Person("Bob", "Dole"),
                                        new Person("Ronald", "McDonald"),
                                        new Person("Alice", "McDonald"),
                                        new Person("Jill", "Doe"));
    Collections.sort(people); //Illegal, Person doesn't implement Comparable.

    Collections.sort(people, new PersonComparator()); //Legal

    //people is now sorted by last name, then first name:
    // --> Jill Doe, John Doe, Bob Dole, Alice McDonald, Ronald McDonald

    //Anonymous Class
    Collections.sort(people, new Comparator<Person>() { //Legal
        public int compare(Person p1, Person p2) {
            //Method code...
        }
    });
```

```
}
```

Version ≥ Java SE 8

Lambda expression based comparators

As of Java 8, comparators can also be expressed as lambda expressions

```
    //Lambda
    Collections.sort(people, (p1, p2) -> { //Legal
        //Method code....
    });
```

Comparator default methods

Furthermore, there are interesting default methods on the Comparator interface for building comparators : the following builds a comparator comparing by `lastName` and then `firstName`.

```
Collections.sort(people, Comparator.comparing(Person::getLastName)
                                .thenComparing(Person::getFirstName));
```

Inversing the order of a comparator

Any comparator can also easily be reversed using the `reversedMethod` which will change ascending order to descending.

Section 81.2: The compareTo and compare Methods

The `Comparable<T>` interface requires one method:

```
public interface Comparable<T> {

    public int compareTo(T other);

}
```

And the `Comparator<T>` interface requires one method:

```
public interface Comparator<T> {

    public int compare(T t1, T t2);

}
```

These two methods do essentially the same thing, with one minor difference: `compareTo` compares **`this`** to `other`, whereas `compare` compares `t1` to `t2`, not caring at all about **`this`**.

Aside from that difference, the two methods have similar requirements. Specifically (for compareTo), Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object. Thus, for the comparison of `a` and `b`:

- If `a < b`, `a.compareTo(b)` and `compare(a,b)` should return a negative integer, and `b.compareTo(a)` and `compare(b,a)` should return a positive integer
- If `a > b`, `a.compareTo(b)` and `compare(a,b)` should return a positive integer, and `b.compareTo(a)` and `compare(b,a)` should return a negative integer
- If `a` equals `b` for comparison, all comparisons should return `0`.

Section 81.3: Natural (comparable) vs explicit (comparator) sorting

There are two `Collections.sort()` methods:

- One that takes a `List<T>` as a parameter where `T` must implement Comparable and override the `compareTo()` method that determines sort order.
- One that takes a List and a Comparator as the arguments, where the Comparator determines the sort order.

First, here is a Person class that implements Comparable:

```
public class Person implements Comparable<Person> {
    private String name;
    private int age;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public int compareTo(Person o) {
        return this.getAge() - o.getAge();
    }
    @Override
    public String toString() {
        return this.getAge()+"-"+this.getName();
    }

}
```

Here is how you would use the above class to sort a List in the natural ordering of its elements, defined by the `compareTo()` method override:

```
//-- usage
List<Person> pList = new ArrayList<Person>();
            Person p = new Person();
            p.setName("A");
            p.setAge(10);
            pList.add(p);
            p = new Person();
            p.setName("Z");
            p.setAge(20);
            pList.add(p);
            p = new Person();
            p.setName("D");
            p.setAge(30);
            pList.add(p);

            //-- natural sorting i.e comes with object implementation, by age
            Collections.sort(pList);
```

```
System.out.println(pList);
```

Here is how you would use an anonymous inline Comparator to sort a List that does not implement Comparable, or in this case, to sort a List in an order other than the natural ordering:

```
//-- explicit sorting, define sort on another property here goes with name
Collections.sort(pList, new Comparator<Person>() {

    @Override
    public int compare(Person o1, Person o2) {
        return o1.getName().compareTo(o2.getName());
    }
});
System.out.println(pList);
```

Section 81.4: Creating a Comparator using comparing method

```
Comparator.comparing(Person::getName)
```

This creates a comparator for the class `Person` that uses this person name as the comparison source. Also it is possible to use method version to compare long, int and double. For example:

```
Comparator.comparingInt(Person::getAge)
```

Reversed order

To create a comparator that imposes the reverse ordering use `reversed()` method:

```
Comparator.comparing(Person::getName).reversed()
```

Chain of comparators

```
Comparator.comparing(Person::getLastName).thenComparing(Person::getFirstName)
```

This will create a comparator that firs compares with last name then compares with first name. You can chain as many comparators as you want.

Section 81.5: Sorting Map entries

As of Java 8, there are default methods on the `Map.Entry` interface to allow sorting of map iterations.

Version ≥ Java SE 8

```
Map<String, Integer> numberOfEmployees = new HashMap<>();
numberOfEmployees.put("executives", 10);
numberOfEmployees.put("human ressources", 32);
numberOfEmployees.put("accounting", 12);
numberOfEmployees.put("IT", 100);

// Output the smallest departement in terms of number of employees
numberOfEmployees.entrySet().stream()
    .sorted(Map.Entry.comparingByValue())
    .limit(1)
    .forEach(System.out::println);   // outputs : executives=10
```

Of course, these can also be used outside of the stream api :

Version ≥ Java SE 8

```
List<Map.Entry<String, Integer>> entries = new ArrayList<>(numberOfEmployees.entrySet());
Collections.sort(entries, Map.Entry.comparingByValue());
```

Chapter 82: Java Floating Point Operations

Floating-point numbers are numbers that have fractional parts (usually expressed with a decimal point). In Java, there is two primitive types for floating-point numbers which are `float` (uses 4 bytes), and `double` (uses 8 bytes). This documentation page is for detailing with examples operations that can be done on floating points in Java.

Section 82.1: Comparing floating point values

You should be careful when comparing floating-point values (`float` or `double`) using relational operators: `==`, `!=`, `<` and so on. These operators give results according to the binary representations of the floating point values. For example:

```
public class CompareTest {
    public static void main(String[] args) {
        double oneThird = 1.0 / 3.0;
        double one = oneThird * 3;
        System.out.println(one == 1.0);      // prints "false"
    }
}
```

The calculation `oneThird` has introduced a tiny rounding error, and when we multiply `oneThird` by 3 we get a result that is slightly different to `1.0`.

This problem of inexact representations is more stark when we attempt to mix `double` and `float` in calculations. For example:

```
public class CompareTest2 {
    public static void main(String[] args) {
        float floatVal = 0.1f;
        double doubleVal = 0.1;
        double doubleValCopy = floatVal;

        System.out.println(floatVal);      // 0.1
        System.out.println(doubleVal);     // 0.1
        System.out.println(doubleValCopy); // 0.10000000149011612

        System.out.println(floatVal == doubleVal); // false
        System.out.println(doubleVal == doubleValCopy); // false
    }
}
```

The floating point representations used in Java for the `float` and `double` types have limited number of digits of precision. For the `float` type, the precision is 23 binary digits or about 8 decimal digits. For the `double` type, it is 52 bits or about 15 decimal digits. On top of that, some arithmetical operations will introduce rounding errors. Therefore, when a program compares floating point values, it standard practice to define an **acceptable delta** for the comparison. If the difference between the two numbers is less than the delta, they are deemed to be equal. For example

```
if (Math.abs(v1 - v2) < delta)
```

Delta compare example:

```
public class DeltaCompareExample {
```

```
    private static boolean deltaCompare(double v1, double v2, double delta) {
        // return true iff the difference between v1 and v2 is less than delta
        return Math.abs(v1 - v2) < delta;
    }

    public static void main(String[] args) {
        double[] doubles = {1.0, 1.0001, 1.0000001, 1.000000001, 1.0000000000001};
        double[] deltas = {0.01, 0.00001, 0.0000001, 0.0000000001, 0};

        // loop through all of deltas initialized above
        for (int j = 0; j < deltas.length; j++) {
            double delta = deltas[j];
            System.out.println("delta: " + delta);

            // loop through all of the doubles initialized above
            for (int i = 0; i < doubles.length - 1; i++) {
                double d1 = doubles[i];
                double d2 = doubles[i + 1];
                boolean result = deltaCompare(d1, d2, delta);

                System.out.println("" + d1 + " == " + d2 + " ? " + result);

            }

            System.out.println();
        }
    }
}
```

Result:

```
delta: 0.01
1.0 == 1.0001 ? true
1.0001 == 1.0000001 ? true
1.0000001 == 1.000000001 ? true
1.000000001 == 1.0000000000001 ? true

delta: 1.0E-5
1.0 == 1.0001 ? false
1.0001 == 1.0000001 ? false
1.0000001 == 1.000000001 ? true
1.000000001 == 1.0000000000001 ? true

delta: 1.0E-7
1.0 == 1.0001 ? false
1.0001 == 1.0000001 ? false
1.0000001 == 1.000000001 ? true
1.000000001 == 1.0000000000001 ? true

delta: 1.0E-10
1.0 == 1.0001 ? false
1.0001 == 1.0000001 ? false
1.0000001 == 1.000000001 ? false
1.000000001 == 1.0000000000001 ? false

delta: 0.0
1.0 == 1.0001 ? false
1.0001 == 1.0000001 ? false
1.0000001 == 1.000000001 ? false
1.000000001 == 1.0000000000001 ? false
```

Also for comparison of **double** and **float** primitive types static `compare` method of corresponding boxing type can be used. For example:

```
double a = 1.0;
double b = 1.0001;

System.out.println(Double.compare(a, b));//-1
System.out.println(Double.compare(b, a));//1
```

Finally, determining what deltas are most appropriate for a comparison can be tricky. A commonly used approach is to pick delta values that are our intuition says are about right. However, if you know scale and (true) accuracy of the input values, and the calculations performed, it may be possible to come up with mathematically sound bounds on the accuracy of the results, and hence for the deltas. (There is a formal branch of Mathematics known as Numerical Analysis that used to be taught to computational scientists that covered this kind of analysis.)

Section 82.2: OverFlow and UnderFlow

Float data type

The float data type is a single-precision 32-bit IEEE 754 floating point.

`Float` overflow

Maximum possible value is `3.4028235e+38` , When it exceeds this value it produces `Infinity`

```
float f = 3.4e38f;
float result = f*2;
System.out.println(result); //Infinity
```

`Float` **UnderFlow**

Minimum value is 1.4e-45f, when is goes below this value it produces `0.0`

```
    float f = 1e-45f;
    float result = f/1000;
    System.out.println(result);
```

double data type

The double data type is a double-precision `64-bit` IEEE 754 floating point.

`Double` **OverFlow**

Maximum possible value is `1.7976931348623157e+308` , When it exceeds this value it produces `Infinity`

```
double d = 1e308;
double result=d*2;
System.out.println(result); //Infinity
```

`Double` **UnderFlow**

Minimum value is 4.9e-324, when is goes below this value it produces `0.0`

```
    double d = 4.8e-323;
    double result = d/1000;
```

```
System.out.println(result); //0.0
```

Section 82.3: Formatting the floating point values

Floating point Numbers can be formatted as a decimal number using `String.format` with `'f'` flag

```
//Two digits in fracttional part are rounded
String format1 = String.format("%.2f", 1.2399);
System.out.println(format1); // "1.24"

// three digits in fractional part are rounded
String format2 = String.format("%.3f", 1.2399);
System.out.println(format2); // "1.240"

//rounded to two digits, filled with zero
String format3 = String.format("%.2f", 1.2);
System.out.println(format3); // returns "1.20"

//rounder to two digits
String format4 = String.format("%.2f", 3.19999);
System.out.println(format4); // "3.20"
```

Floating point Numbers can be formatted as a decimal number using `DecimalFormat`

```
// rounded with one digit fractional part
 String format = new DecimalFormat("0.#").format(4.3200);
 System.out.println(format); // 4.3

// rounded with two digit fractional part
 String format = new DecimalFormat("0.##").format(1.2323000);
 System.out.println(format); //1.23

 // formatting floating numbers to decimal number
 double dv = 123456789;
 System.out.println(dv); // 1.23456789E8
 String format =  new DecimalFormat("0").format(dv);
 System.out.println(format); //123456789
```

Section 82.4: Strict Adherence to the IEEE Specification

By default, floating point operations on **`float`** and **`double`** *do not* strictly adhere to the rules of the IEEE 754 specification. An expression is allowed to use implementation-specific extensions to the range of these values; essentially allowing them to be *more* accurate than required.

`strictfp` disables this behavior. It is applied to a class, interface, or method, and applies to everything contained in it, such as classes, interfaces, methods, constructors, variable initializers, etc. With **`strictfp`**, the intermediate values of a floating-point expression *must* be within the float value set or the double value set. This causes the results of such expressions to be exactly those that the IEEE 754 specification predicts.

All constant expressions are implicitly strict, even if they aren't inside a **`strictfp`** scope.

Therefore, **`strictfp`** has the net effect of sometimes making certain corner case computations *less* accurate, and can also make floating point operations *slower* (as the CPU is now doing more work to ensure any native extra precision does not affect the result). However, it also causes the results to be exactly the same on all platforms. It is therefore useful in things like scientific programs, where reproducibility is more important than speed.

```
public class StrictFP { // No strictfp -> default lenient
```

```
    public strictfp float strict(float input) {
        return input * input / 3.4f; // Strictly adheres to the spec.
                                     // May be less accurate and may be slower.
    }

    public float lenient(float input) {
        return input * input / 3.4f; // Can sometimes be more accurate and faster,
                                     // but results may not be reproducable.
    }

    public static final strictfp class Ops { // strictfp affects all enclosed entities
        private StrictOps() {}

        public static div(double dividend, double divisor) { // implicitly strictfp
            return dividend / divisor;
        }
    }
}
```

Chapter 83: Currency and Money

Section 83.1: Add custom currency

Required JARs on classpath:

- javax.money:money-api:1.0 (JSR354 money and currency api)
- org.javamoney:moneta:1.0 (Reference implementation)
- javax:annotation-api:1.2. (Common annotations used by reference implementation)

```
// Let's create non-ISO currency, such as bitcoin

// At first, this will throw UnknownCurrencyException
MonetaryAmount moneys = Money.of(new BigDecimal("0.1"), "BTC");

// This happens because bitcoin is unknown to default currency
// providers
System.out.println(Monetary.isCurrencyAvailable("BTC")); // false

// We will build new currency using CurrencyUnitBuilder provided by org.javamoney.moneta
CurrencyUnit bitcoin = CurrencyUnitBuilder
    .of("BTC", "BtcCurrencyProvider") // Set currency code and currency provider name
    .setDefaultFractionDigits(2)      // Set default fraction digits
    .build(true);                     // Build new currency unit. Here 'true' means
                                      // currency unit is to be registered and
                                      // accessible within default monetary context

// Now BTC is available
System.out.println(Monetary.isCurrencyAvailable("BTC")); // True
```

Chapter 84: Object Cloning

Section 84.1: Cloning performing a deep copy

To copy nested objects, a deep copy must be performed, as shown in this example.

```
import java.util.ArrayList;
import java.util.List;

public class Sheep implements Cloneable {

    private String name;

    private int weight;

    private List<Sheep> children;

    public Sheep(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        Sheep clone = (Sheep) super.clone();
        if (children != null) {
            // make a deep copy of the children
            List<Sheep> cloneChildren = new ArrayList<>(children.size());
            for (Sheep child : children) {
                cloneChildren.add((Sheep) child.clone());
            }
            clone.setChildren(cloneChildren);
        }
        return clone;
    }

    public List<Sheep> getChildren() {
        return children;
    }

    public void setChildren(List<Sheep> children) {
        this.children = children;
    }

}

import java.util.Arrays;
import java.util.List;

// create a sheep
Sheep sheep = new Sheep("Dolly", 20);

// create children
Sheep child1 = new Sheep("Child1", 4);
Sheep child2 = new Sheep("Child2", 5);

sheep.setChildren(Arrays.asList(child1, child2));

// clone the sheep
Sheep dolly =  (Sheep) sheep.clone();
```

```
List<Sheep> sheepChildren = sheep.getChildren();
List<Sheep> dollysChildren = dolly.getChildren();
for (int i = 0; i < sheepChildren.size(); i++) {
    // prints false, both arrays contain copies of the objects inside
    System.out.println(sheepChildren.get(i) == dollysChildren.get(i));
}
```

Section 84.2: Cloning using a copy factory

```
public class Sheep {

    private String name;

    private int weight;

    public Sheep(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    public static Sheep newInstance(Sheep other);
        return new Sheep(other.name, other.weight)
    }

}
```

Section 84.3: Cloning using a copy constructor

An easy way to clone an object is by implementing a copy constructor.

```
public class Sheep {

    private String name;

    private int weight;

    public Sheep(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    // copy constructor
    // copies the fields of other into the new object
    public Sheep(Sheep other) {
        this.name = other.name;
        this.weight = other.weight;
    }

}

// create a sheep
Sheep sheep = new Sheep("Dolly", 20);
// clone the sheep
Sheep dolly = new Sheep(sheep); // dolly.name is "Dolly" and dolly.weight is 20
```

Section 84.4: Cloning by implementing Clonable interface

Cloning an object by implementing the Cloneable interface.

```
public class Sheep implements Cloneable {

    private String name;

    private int weight;

    public Sheep(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

}

// create a sheep
Sheep sheep = new Sheep("Dolly", 20);
// clone the sheep
Sheep dolly =  (Sheep) sheep.clone(); // dolly.name is "Dolly" and dolly.weight is 20
```

Section 84.5: Cloning performing a shallow copy

Default behavior when cloning an object is to perform a shallow copy of the object's fields. In that case, both the original object and the cloned object, hold references to the same objects.

This example shows that behavior.

```
import java.util.List;

public class Sheep implements Cloneable {

    private String name;

    private int weight;

    private List<Sheep> children;

    public Sheep(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    public List<Sheep> getChildren() {
        return children;
    }

    public void setChildren(List<Sheep> children) {
        this.children = children;
    }

}
```

```java
import java.util.Arrays;
import java.util.List;

// create a sheep
Sheep sheep = new Sheep("Dolly", 20);

// create children
Sheep child1 = new Sheep("Child1", 4);
Sheep child2 = new Sheep("Child2", 5);

sheep.setChildren(Arrays.asList(child1, child2));

// clone the sheep
Sheep dolly =  (Sheep) sheep.clone();
List<Sheep> sheepChildren = sheep.getChildren();
List<Sheep> dollysChildren = dolly.getChildren();
for (int i = 0; i < sheepChildren.size(); i++) {
    // prints true, both arrays contain the same objects
    System.out.println(sheepChildren.get(i) == dollysChildren.get(i));
}
```

Chapter 85: Recursion

Recursion occurs when a method calls itself. Such a method is called **recursive**. A recursive method may be more concise than an equivalent non-recursive approach. However, for deep recursion, sometimes an iterative solution can consume less of a thread's finite stack space.

This topic includes examples of recursion in Java.

Section 85.1: The basic idea of recursion

What is recursion:

In general, recursion is when a function invokes itself, either directly or indirectly. For example:

```
// This method calls itself "infinitely"
public void useless() {
    useless();  // method calls itself (directly)
}
```

Conditions for applying recursion to a problem:

There are two preconditions for using recursive functions to solving a specific problem:

1. There must be a base condition for the problem, which will be the endpoint for the recursion. When a recursive function reaches the base condition, it makes no further (deeper) recursive calls.
2. Each level of recursion should be attempting a smaller problem. The recursive function thus divides the problem into smaller and smaller parts. Assuming that the problem is finite, this will ensure that the recursion terminates.

In Java there is a third precondition: it should not be necessary to recurse too deeply to solve the problem; see Deep recursion is problematic in Java

Example

The following function calculates factorials using recursion. Notice how the method `factorial` calls itself within the function. Each time it calls itself, it reduces the parameter `n` by 1. When `n` reaches 1 (the base condition) the function will recurse no deeper.

```
public int factorial(int n) {
    if (n <= 1) { // the base condition
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

This is not a practical way of computing factorials in Java, since it does not take account of integer overflow, or call stack overflow (i.e. `StackOverflowError` exceptions) for large values of `n`.

Section 85.2: Deep recursion is problematic in Java

Consider the following naive method for adding two positive numbers using recursion:

```
public static int add(int a, int b) {
    if (a == 0) {
        return b;
    } else {
        return add(a - 1, b + 1);  // TAIL CALL
    }
}
```

This is algorithmically correct, but it has a major problem. If you call `add` with a large `a`, it will crash with a `StackOverflowError`, on any version of Java up to (at least) Java 9.

In a typical functional programming language (and many other languages) the compiler optimizes tail recursion. The compiler would notice that the call to `add` (at the tagged line) is a tail call, and would effectively rewrite the recursion as a loop. This transformation is called tail-call elimination.

However, current generation Java compilers do not perform tail call elimination. (This is not a simple oversight. There are substantial technical reasons for this; see below.) Instead, each recursive call of `add` causes a new frame to be allocated on the thread's stack. For example, if you call `add(1000, 1)`, it will take `1000` recursive calls to arrive at the answer `1001`.

The problem is that the size of Java thread stack is fixed when the thread is created. (This includes the "main" thread in a single-threaded program.) If too many stack frames are allocated the stack will overflow. The JVM will detect this and throw a `StackOverflowError`.

One approach to dealing with this is to simply use a bigger stack. There are JVM options that control the default size of a stack, and you can also specify the stack size as a `Thread` constructor parameter. Unfortunately, this only "puts off" the stack overflow. If you need to do a computation that requires an even larger stack, then the `StackOverflowError` comes back.

The real solution is to identify recursive algorithms where deep recursion is likely, and *manually* perform the tail-call optimization at the source code level. For example, our `add` method can be rewritten as follows:

```
public static int add(int a, int b) {
    while (a != 0) {
       a = a - 1;
       b = b + 1;
    }
    return b;
}
```

(Obviously, there are better ways to add two integers. The above is simply to illustrate the effect of manual tail-call elimination.)

Why tail-call elimination is not implemented in Java (yet)

There are a number of reasons why adding tail call elimination to Java is not easy. For example:

- Some code could rely on `StackOverflowError` to (for example) place a bound on the size of a computational problem.
- Sandbox security managers often rely on analyzing the call stack when deciding whether to allow non-privileged code to perform a privileged action.

As John Rose explains in "Tail calls in the VM":

> *"The effects of removing the caller's stack frame are visible to some APIs, notably access control checks and*

> *stack tracing. It is as if the caller's caller had directly called the callee. Any privileges possessed by the caller are discarded after control is transferred to the callee. However, the linkage and accessibility of the callee method are computed before the transfer of control, and take into account the tail-calling caller."*

In other words, tail-call elimination could cause an access control method to mistakenly think that a security sensitive API was was being called by trusted code.

Section 85.3: Types of Recursion

Recursion can be categorized as either **Head Recursion** or **Tail Recursion**, depending on where the recursive method call is placed.

In **head recursion**, the recursive call, when it happens, comes before other processing in the function (think of it happening at the top, or head, of the function).

In **tail recursion**, it's the opposite—the processing occurs before the recursive call. Choosing between the two recursive styles may seem arbitrary, but the choice can make all the difference.

A function with a path with a single recursive call at the beginning of the path uses what is called head recursion. The factorial function of a previous exhibit uses head recursion. The first thing it does once it determines that recursion is needed is to call itself with the decremented parameter. A function with a single recursive call at the end of a path is using tail recursion.

```
public void tail(int n)
{
   if(n == 1)
      return;
   else
      System.out.println(n);

   tail(n-1);
}
```

```
public void head(int n)
   {
     if(n == 0)
        return;
     else
        head(n-1);

      System.out.println(n);
    }
```

If the recursive call occurs at the end of a method, it is called a `tail recursion`. The tail recursion is `similar to a loop`. The `method executes all the statements before jumping into the next recursive call`.

If the recursive call occurs at the `beginning of a method, it is called a head recursion`. The `method saves the state before jumping into the next recursive call`.

Reference: The difference between head & tail recursion

Section 85.4: Computing the Nth Fibonacci Number

The following method computes the Nth Fibonacci number using recursion.

```
public int fib(final int n) {
    if (n > 2) {
        return fib(n - 2) + fib(n - 1);
    }
    return 1;
}
```

The method implements a base case (n <= 2) and a recursive case (n>2). This illustrates the use of recursion to compute a recursive relation.

However, while this example is illustrative, it is also inefficient: each single instance of the method will call the function itself twice, leading to an exponential growth in the number of times the function is called as N increases. The above function is O(2N), but an equivalent iterative solution has complexity O(N). In addition, there is a "closed form" expression that can be evaluated in O(N) floating-point multiplications.

Section 85.5: StackOverflowError & recursion to loop

If a recursive call goes "too deep", this results in a `StackOverflowError`. Java allocates a new frame for every method call on its thread's stack. However, the space of each thread's stack is limited. Too many frames on the stack leads to the Stack Overflow (SO).

Example

```
public static void recursion(int depth) {
    if (depth > 0) {
        recursion(depth-1);
    }
}
```

Calling this method with large parameters (e.g. `recursion(50000)` probably will result in a stack overflow. The exact value depends on the thread stack size, which in turn depends on the thread construction, command-line parameters such as `-Xss`, or the default size for the JVM.

Workaround

A recursion can be converted to a loop by storing the data for each recursive call in a data structure. This data structure can be stored on the heap rather than on the thread stack.

In general the data required to restore the state of a method invocation can be stored in a stack and a while loop can be used to "simulate" the recursive calls. Data that may be required include:

- the object the method was called for (instance methods only)
- the method parameters
- local variables
- the current position in the execution or the method

Example

The following class allows recursive of a tree structure printing up to a specified depth.

```
public class Node {

    public int data;
    public Node left;
    public Node right;

    public Node(int data) {
        this(data, null, null);
    }

    public Node(int data, Node left, Node right) {
        this.data = data;
        this.left = left;
        this.right = right;
    }

    public void print(final int maxDepth) {
        if (maxDepth <= 0) {
```

```
            System.out.print("(...)");
        } else {
            System.out.print("(");
            if (left != null) {
                left.print(maxDepth-1);
            }
            System.out.print(data);
            if (right != null) {
                right.print(maxDepth-1);
            }
            System.out.print(")");
        }
    }

}
```

e.g.

```
Node n = new Node(10, new Node(20, new Node(50), new Node(1)), new Node(30, new Node(42), null));
n.print(2);
System.out.println();
```

Prints

```
(((...)20(...))10((...)30))
```

This could be converted to the following loop:

```
public class Frame {

    public final Node node;

    // 0: before printing anything
    // 1: before printing data
    // 2: before printing ")"
    public int state = 0;
    public final int maxDepth;

    public Frame(Node node, int maxDepth) {
        this.node = node;
        this.maxDepth = maxDepth;
    }

}
```

```
List<Frame> stack = new ArrayList<>();
stack.add(new Frame(n, 2)); // first frame = initial call

while (!stack.isEmpty()) {
    // get topmost stack element
    int index = stack.size() - 1;
    Frame frame = stack.get(index); // get topmost frame
    if (frame.maxDepth <= 0) {
        // termial case (too deep)
        System.out.print("(...)");
        stack.remove(index); // drop frame
    } else {
        switch (frame.state) {
            case 0:
                frame.state++;
```

```
                // do everything done before the first recursive call
                System.out.print("(");
                if (frame.node.left != null) {
                    // add new frame (recursive call to left and stop)
                    stack.add(new Frame(frame.node.left, frame.maxDepth - 1));
                    break;
                }
            case 1:
                frame.state++;

                // do everything done before the second recursive call
                System.out.print(frame.node.data);
                if (frame.node.right != null) {
                    // add new frame (recursive call to right and stop)
                    stack.add(new Frame(frame.node.right, frame.maxDepth - 1));
                    break;
                }
            case 2:
                // do everything after the second recursive call & drop frame
                System.out.print(")");
                stack.remove(index);
        }
    }
}
System.out.println();
```

Note: This is just an example of the general approach. Often you can come up with a much better way to represent a frame and/or store the frame data.

Section 85.6: Computing the Nth power of a number

The following method computes the value of `num` raised to the power of `exp` using recursion:

```
public long power(final int num, final int exp) {
    if (exp == 0) {
        return 1;
    }
    if (exp == 1) {
        return num;
    }
    return num * power(num, exp - 1);
}
```

This illustrates the principles mentioned above: the recursive method implements a base case (two cases, n = 0 and n = 1) that terminates the recursion, and a recursive case that calls the method again. This method is O(N) and can be reduced to a simple loop using tail-call optimization.

Section 85.7: Traversing a Tree data structure with recursion

Consider the Node class having 3 members data, left child pointer and right child pointer like below.

```
public class Node {
    public int data;
    public Node left;
    public Node right;

    public Node(int data){
        this.data = data;
    }
```

```
}
```

We can traverse the tree constructed by connecting multiple Node class's object like below, the traversal is called in-order traversal of tree.

```
public static void inOrderTraversal(Node root) {
        if (root != null) {
            inOrderTraversal(root.left); // traverse left sub tree
            System.out.print(root.data + " "); // traverse current node
            inOrderTraversal(root.right); // traverse right sub tree
        }
    }
```

As demonstrated above, using **recursion** we can traverse the **tree data structure** without using any other data structure which is not possible with the **iterative** approach.

Section 85.8: Reverse a string using Recursion

Below is a recursive code to reverse a string

```
/**
 * Just a snippet to explain the idea of recursion
 *
 **/

public class Reverse {
    public static void main (String args[]) {
        String string = "hello world";
        System.out.println(reverse(string)); //prints dlrow olleh
    }

    public static String reverse(String s) {
        if (s.length() == 1) {
            return s;
        }

        return reverse(s.substring(1)) + s.charAt(0);
    }
}
```

Section 85.9: Computing the sum of integers from 1 to N

The following method computes the sum of integers from 0 to N using recursion.

```
public int sum(final int n) {
    if (n > 0) {
        return n + sum(n - 1);
    } else {
        return n;
    }
}
```

This method is O(N) and can be reduced to a simple loop using tail-call optimization. In fact there is a *closed form* expression that computes the sum in `O(1)` operations.

Chapter 86: Converting to and from Strings

Section 86.1: Converting String to other datatypes

You can convert a **numeric** string to various Java numeric types as follows:

String to int:

```
String number = "12";
int num = Integer.parseInt(number);
```

String to float:

```
String number = "12.0";
float num = Float.parseFloat(number);
```

String to double:

```
String double = "1.47";
double num = Double.parseDouble(double);
```

String to boolean:

```
String falseString = "False";
boolean falseBool = Boolean.parseBoolean(falseString);   // falseBool = false

String trueString = "True";
boolean trueBool = Boolean.parseBoolean(trueString);     // trueBool = true
```

String to long:

```
String number = "47";
long num = Long.parseLong(number);
```

String to BigInteger:

```
String bigNumber = "21";
BigInteger reallyBig = new BigInteger(bigNumber);
```

String to BigDecimal:

```
String bigFraction = "17.21455";
BigDecimal reallyBig = new BigDecimal(bigFraction);
```

Conversion Exceptions:

The numeric conversions above will all throw an (unchecked) `NumberFormatException` if you attempt to parse a string that is not a suitably formatted number, or is out of range for the target type. The Exceptions topic discusses how to deal with such exceptions.

If you wanted to test that you can parse a string, you could implement a `tryParse...` method like this:

```
boolean tryParseInt (String value) {
    try {
        String somechar = Integer.parseInt(value);
        return true;
     } catch (NumberFormatException e) {
        return false;
     }
}
```

However, calling this `tryParse...` method immediately before parsing is (arguably) poor practice. It would be better to just call the `parse...` method and deal with the exception.

Section 86.2: Conversion to / from bytes

To encode a string into a byte array, you can simply use the `String#getBytes()` method, with one of the standard character sets available on any Java runtime:

```
byte[] bytes = "test".getBytes(StandardCharsets.UTF_8);
```

and to decode:

```
String testString = new String(bytes, StandardCharsets.UTF_8);
```

you can further simplify the call by using a static import:

```
import static java.nio.charset.StandardCharsets.UTF_8;
...
byte[] bytes = "test".getBytes(UTF_8);
```

For less common character sets you can indicate the character set with a string:

```
byte[] bytes = "test".getBytes("UTF-8");
```

and the reverse:

```
String testString = new String (bytes, "UTF-8");
```

this does however mean that you have to handle the checked `UnsupportedCharsetException`.

The following call will use the default character set. The default character set is platform specific and generally differs between Windows, Mac and Linux platforms.

```
byte[] bytes = "test".getBytes();
```

and the reverse:

```
String testString = new String(bytes);
```

Note that invalid characters and bytes may be replaced or skipped by these methods. For more control - for instance for validating input - you're encouraged to use the `CharsetEncoder` and `CharsetDecoder` classes.

Section 86.3: Base64 Encoding / Decoding

Occasionally you will find the need to encode binary data as a *base64*-encoded string.

For this we can use the `DatatypeConverter` class from the `javax.xml.bind` package:

```
import javax.xml.bind.DatatypeConverter;
import java.util.Arrays;

// arbitrary binary data specified as a byte array
byte[] binaryData = "some arbitrary data".getBytes("UTF-8");

// convert the binary data to the base64-encoded string
String encodedData = DatatypeConverter.printBase64Binary(binaryData);
// encodedData is now "c29tZSBhcmJpdHJhcnkgZGF0YQ=="

// convert the base64-encoded string back to a byte array
byte[] decodedData = DatatypeConverter.parseBase64Binary(encodedData);

// assert that the original data and the decoded data are equal
assert Arrays.equals(binaryData, decodedData);
```

Apache commons-codec

Alternatively, we can use `Base64` from Apache commons-codec.

```
import org.apache.commons.codec.binary.Base64;

// your blob of binary as a byte array
byte[] blob = "someBinaryData".getBytes();

// use the Base64 class to encode
String binaryAsAString = Base64.encodeBase64String(blob);

// use the Base64 class to decode
byte[] blob2 = Base64.decodeBase64(binaryAsAString);

// assert that the two blobs are equal
System.out.println("Equal : " + Boolean.toString(Arrays.equals(blob, blob2)));
```

If you inspect this program wile running, you will see that `someBinaryData` encodes to `c29tZUJpbmFyeURhdGE=`, a very managable *UTF-8* String object.

`Version ≥ Java SE 8`

Details for the same can be found at Base64

```
// encode with padding
String encoded = Base64.getEncoder().encodeToString(someByteArray);

// encode without padding
String encoded = Base64.getEncoder().withoutPadding().encodeToString(someByteArray);

// decode a String
byte [] barr = Base64.getDecoder().decode(encoded);
```

Reference

Section 86.4: Converting other datatypes to String

- You can get the value of other primitive data types as a String using one the String class's `valueOf` methods.

  For example:

  ```
  int i = 42;
  String string = String.valueOf(i);
  //string now equals "42".
  ```

  This method is also overloaded for other datatypes, such as **`float`**, **`double`**, **`boolean`**, and even `Object`.

- You can also get any other Object (any instance of any class) as a String by calling `.toString` on it. For this to give useful output, the class must override `toString()`. Most of the standard Java library classes do, such as `Date` and others.

  For example:

  ```
  Foo foo = new Foo(); //Any class.
  String stringifiedFoo = foo.toString().
  ```

  Here `stringifiedFoo` contains a representation of `foo` as a String.

You can also convert any number type to String with short notation like below.

```
int i = 10;
String str = i + "";
```

Or just simple way is

```
String str = 10 + "";
```

Section 86.5: Getting a `String` from an `InputStream`

A `String` can be read from an `InputStream` using the byte array constructor.

```
import java.io.*;

public String readString(InputStream input) throws IOException {
    byte[] bytes = new byte[50]; // supply the length of the string in bytes here
    input.read(bytes);
    return new String(bytes);
}
```

This uses the system default charset, although an alternate charset may be specified:

```
return new String(bytes, Charset.forName("UTF-8"));
```

Chapter 87: Random Number Generation

Section 87.1: Pseudo Random Numbers

Java provides, as part of the `utils` package, a basic pseudo-random number generator, appropriately named `Random`. This object can be used to generate a pseudo-random value as any of the built-in numerical datatypes (`int`, `float`, etc). You can also use it to generate a random Boolean value, or a random array of bytes. An example usage is as follows:

```
import java.util.Random;

...

Random random = new Random();
int randInt = random.nextInt();
long randLong = random.nextLong();

double randDouble = random.nextDouble(); //This returns a value between 0.0 and 1.0
float randFloat = random.nextFloat(); //Same as nextDouble

byte[] randBytes = new byte[16];
random.nextBytes(randBytes); //nextBytes takes a user-supplied byte array, and fills it with random
bytes. It returns nothing.
```

NOTE: This class only produces fairly low-quality pseudo-random numbers, and should never be used to generate random numbers for cryptographic operations or other situations where higher-quality randomness is critical (For that, you would want to use the `SecureRandom` class, as noted below). An explanation for the distinction between "secure" and "insecure" randomness is beyond the scope of this example.

Section 87.2: Pseudo Random Numbers in Specific Range

The method `nextInt(int bound)` of `Random` accepts an upper exclusive boundary, i.e. a number that the returned random value must be less than. However, only the `nextInt` method accepts a bound; `nextLong`, `nextDouble` etc. do not.

```
Random random = new Random();
random.nextInt(1000); // 0 - 999

int number = 10 + random.nextInt(100); // number is in the range of 10 to 109
```

Starting in Java 1.7, you may also use `ThreadLocalRandom` (source). This class provides a thread-safe PRNG (pseudo-random number generator). Note that the `nextInt` method of this class accepts both an upper and lower bound.

```
import java.util.concurrent.ThreadLocalRandom;

// nextInt is normally exclusive of the top value,
// so add 1 to make it inclusive
ThreadLocalRandom.current().nextInt(min, max + 1);
```

Note that the official documentation states that `nextInt(int bound)` can do weird things when `bound` is near 230+1 (emphasis added):

> The algorithm is slightly tricky. **It rejects values that would result in an uneven distribution** (due to the fact that 2^31 is not divisible by n). The probability of a value being rejected depends on n. **The worst**

> **case is n=2^30+1, for which the probability of a reject is 1/2, and the expected number of iterations before the loop terminates is 2.**

In other words, specifying a bound will (slightly) decrease the performance of the `nextInt` method, and this performance decrease will become more pronounced as the `bound` approaches half the max int value.

Section 87.3: Generating cryptographically secure pseudorandom numbers

`Random` and `ThreadLocalRandom` are good enough for everyday use, but they have a big problem: They are based on a linear congruential generator, an algorithm whose output can be predicted rather easily. Thus, these two classes are **not** suitable for cryptographic uses (such as key generation).

One can use `java.security.SecureRandom` in situations where a PRNG with an output that is very hard to predict is required. Predicting the random numbers created by instances of this class is hard enough to label the class as **cryptographically secure**.

```
import java.security.SecureRandom;
import java.util.Arrays;

public class Foo {
    public static void main(String[] args) {
        SecureRandom rng = new SecureRandom();
        byte[] randomBytes = new byte[64];
        rng.nextBytes(randomBytes); // Fills randomBytes with random bytes (duh)
        System.out.println(Arrays.toString(randomBytes));
    }
}
```

Besides being cryptographically secure, `SecureRandom` has a gigantic period of 2160, compared to `Random`s period of 248. It has one drawback of being considerably slower than `Random` and other linear PRNGs such as Mersenne Twister and Xorshift, however.

Note that SecureRandom implementation is both platform and provider dependent. The default `SecureRandom` (given by `SUN` provider in `sun.security.provider.SecureRandom`):

- on Unix-like systems, seeded with data from `/dev/random` and/or `/dev/urandom`.
- on Windows, seeded with calls to `CryptGenRandom()` in CryptoAPI.

Section 87.4: Generating Random Numbers with a Specified Seed

```
//Creates a Random instance with a seed of 12345.
Random random = new Random(12345L);

//Gets a ThreadLocalRandom instance
ThreadLocalRandom tlr = ThreadLocalRandom.current();

//Set the instance's seed.
tlr.setSeed(12345L);
```

Using the same seed to generate random numbers will return the same numbers every time, so setting a different seed for every `Random` instance is a good idea if you don't want to end up with duplicate numbers.

A good method to get a Long that is different for every call is System.currentTimeMillis():

```
Random random = new Random(System.currentTimeMillis());
ThreadLocalRandom.current().setSeed(System.currentTimeMillis());
```

Section 87.5: Select random numbers without duplicates

```
/**
 * returns a array of random numbers with no duplicates
 * @param range the range of possible numbers for ex. if 100 then it can be anywhere from 1-100
 * @param length the length of the array of random numbers
 * @return array of random numbers with no duplicates.
 */
public static int[] getRandomNumbersWithNoDuplicates(int range, int length){
    if (length<range){
        // this is where all the random numbers
        int[] randomNumbers = new int[length];

        // loop through all the random numbers to set them
        for (int q = 0; q < randomNumbers.length; q++){

            // get the remaining possible numbers
            int remainingNumbers = range-q;

            // get a new random number from the remainingNumbers
            int newRandSpot = (int) (Math.random()*remainingNumbers);

            newRandSpot++;

            // loop through all the possible numbers
            for (int t = 1; t < range+1; t++){

                // check to see if this number has already been taken
                boolean taken = false;
                for (int number : randomNumbers){
                    if (t==number){
                        taken = true;
                        break;
                    }
                }

                // if it hasnt been taken then remove one from the spots
                if (!taken){
                    newRandSpot--;

                    // if we have gone though all the spots then set the value
                    if (newRandSpot==0){
                        randomNumbers[q] = t;
                    }
                }
            }
        }
        return randomNumbers;
    } else {
        // invalid can't have a length larger then the range of possible numbers
    }
    return null;
}
```

The method works by looping though an array that has the size of the requested length and finds the remaining

length of possible numbers. It sets a random number of those possible numbers `newRandSpot` and finds that number within the non taken number left. It does this by looping through the range and checking to see if that number has already been taken.

For example if the range is 5 and the length is 3 and we have already chosen the number 2. Then we have 4 remaining numbers so we get a random number between 1 and 4 and we loop through the range(5) skipping over any numbers that we have already used(2).

Now let's say the next number chosen between 1 & 4 is 3. On the first loop we get 1 which has not yet been taken so we can remove 1 from 3 making it 2. Now on the second loop we get 2 which has been taken so we do nothing. We follow this pattern until we get to 4 where once we remove 1 it becomes 0 so we set the new randomNumber to 4.

Section 87.6: Generating Random number using apache-common lang3

We can use `org.apache.commons.lang3.RandomUtils` to generate random numbers using a single line.

```
int x = RandomUtils.nextInt(1, 1000);
```

The method `nextInt(int startInclusive, int endExclusive)` takes a range.

Apart from int, we can generate random **`long`**, **`double`**, **`float`** and `bytes` using this class.

`RandomUtils` class contains the following methods-

```
static byte[] nextBytes(int count) //Creates an array of random bytes.
static double nextDouble() //Returns a random double within 0 - Double.MAX_VALUE
static double nextDouble(double startInclusive, double endInclusive) //Returns a random double
within the specified range.
static float nextFloat() //Returns a random float within 0 - Float.MAX_VALUE
static float nextFloat(float startInclusive, float endInclusive) //Returns a random float within
the specified range.
static int nextInt() //Returns a random int within 0 - Integer.MAX_VALUE
static int nextInt(int startInclusive, int endExclusive) //Returns a random integer within the
specified range.
static long nextLong() //Returns a random long within 0 - Long.MAX_VALUE
static long nextLong(long startInclusive, long endExclusive) //Returns a random long within the
specified range.
```

Chapter 88: Singletons

A singleton is a class that only ever has one single instance. For more information on the Singleton *design pattern*, please refer to the Singleton topic in the Design Patterns tag.

Section 88.1: Enum Singleton

Version ≥ Java SE 5

```
public enum Singleton {
    INSTANCE;

    public void execute (String arg) {
        // Perform operation here
    }
}
```

Enums have private constructors, are final and provide proper serialization machinery. They are also very concise and lazily initialized in a thread safe manner.

The JVM provides a guarantee that enum values will not be instantiated more than once each, which gives the enum singleton pattern a very strong defense against reflection attacks.

What the enum pattern *doesn't* protect against is other developers physically adding more elements to the source code. Consequently, if you choose this implementation style for your singletons it is imperative that you very clearly document that no new values should be added to those enums.

This is the recommended way of implementing the singleton pattern, as explained by Joshua Bloch in Effective Java.

Section 88.2: Singleton without use of Enum (eager initialization)

```
public class Singleton {

    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

It can be argued that this example is *effectively* lazy initialization. Section 12.4.1 of the Java Language Specification states:

> A class or interface type T will be initialized immediately before the first occurrence of any one of the following:
>
> - T is a class and an instance of T is created
> - T is a class and a static method declared by T is invoked
> - A static field declared by T is assigned
> - A static field declared by T is used and the field is not a constant variable
> - T is a top level class, and an assert statement lexically nested within T is executed.

Therefore, as long as there are no other static fields or static methods in the class, the `Singleton` instance will not be initialized until the method `getInstance()` is invoked the first time.

Section 88.3: Thread-safe lazy initialization using holder class | Bill Pugh Singleton implementation

```
public class Singleton {
    private static class InstanceHolder {
        static final Singleton INSTANCE = new Singleton();
    }

    public static Singleton getInstance() {
        return InstanceHolder.INSTANCE;
    }

    private Singleton() {}
}
```

This initializes the `INSTANCE` variable on the first call to `Singleton.getInstance()`, taking advantage of the language's thread safety guarantees for static initialization without requiring additional synchronization.

This implementation is also known as Bill Pugh singleton pattern. [Wiki]

Section 88.4: Thread safe Singleton with double checked locking

This type of Singleton is thread safe, and prevents unnecessary locking after the Singleton instance has been created.

Version ≥ Java SE 5

```
public class MySingleton {

    // instance of class
    private static volatile MySingleton instance = null;

    // Private constructor
    private MySingleton() {
        // Some code for constructing object
    }

    public static MySingleton getInstance() {
        MySingleton result = instance;

        //If the instance already exists, no locking is necessary
        if(result == null) {
            //The singleton instance doesn't exist, lock and check again
            synchronized(MySingleton.class) {
                result = instance;
                if(result == null) {
                    instance = result = new MySingleton();
                }
            }
        }
        return result;
    }
}
```

It must be emphasized -- in versions prior to Java SE 5, the implementation above is incorrect and should be

avoided. It is not possible to implement double-checked locking correctly in Java prior to Java 5.

Section 88.5: Extending singleton (singleton inheritance)

In this example, base class `Singleton` provides `getMessage()` method that returns `"Hello world!"` message.

It's subclasses `UppercaseSingleton` and `LowercaseSingleton` override getMessage() method to provide appropriate representation of the message.

```
//Yeah, we'll need reflection to pull this off.
import java.lang.reflect.*;

/*
Enumeration that represents possible classes of singleton instance.
If unknown, we'll go with base class - Singleton.
*/
enum SingletonKind {
    UNKNOWN,
    LOWERCASE,
    UPPERCASE
}

//Base class
class Singleton{

    /*
    Extended classes has to be private inner classes, to prevent extending them in
    uncontrolled manner.
     */
    private class UppercaseSingleton extends Singleton {

        private UppercaseSingleton(){
            super();
        }

        @Override
        public String getMessage() {
            return super.getMessage().toUpperCase();
        }
    }

    //Another extended class.
    private class LowercaseSingleton extends Singleton
    {
        private LowercaseSingleton(){
            super();
        }

        @Override
        public String getMessage() {
            return super.getMessage().toLowerCase();
        }
    }

    //Applying Singleton pattern
    private static SingletonKind kind = SingletonKind.UNKNOWN;

    private static Singleton instance;

    /*
    By using this method prior to getInstance() method, you effectively change the
```

```
    type of singleton instance to be created.
     */
    public static void setKind(SingletonKind kind) {
        Singleton.kind = kind;
    }

    /*
    If needed, getInstance() creates instance appropriate class, based on value of
    singletonKind field.
     */
    public static Singleton getInstance()
        throws  NoSuchMethodException,
                IllegalAccessException,
                InvocationTargetException,
                InstantiationException {

        if(instance==null){
            synchronized (Singleton.class){
                if(instance==null){
                    Singleton singleton = new Singleton();
                    switch (kind){
                        case UNKNOWN:

                            instance = singleton;
                            break;

                        case LOWERCASE:

                            /*
                            I can't use simple

                            instance = new LowercaseSingleton();

                            because java compiler won't allow me to use
                            constructor of inner class in static context,
                            so I use reflection API instead.

                            To be able to access inner class by reflection API,
                            I have to create instance of outer class first.
                            Therefore, in this implementation, Singleton cannot be
                            abstract class.
                            */

                            //Get the constructor of inner class.
                            Constructor<LowercaseSingleton> lcConstructor =

LowercaseSingleton.class.getDeclaredConstructor(Singleton.class);

                            //The constructor is private, so I have to make it accessible.
                            lcConstructor.setAccessible(true);

                            // Use the constructor to create instance.
                            instance = lcConstructor.newInstance(singleton);

                            break;

                        case UPPERCASE:

                            //Same goes here, just with different type
                            Constructor<UppercaseSingleton> ucConstructor =

UppercaseSingleton.class.getDeclaredConstructor(Singleton.class);
```

```
                            ucConstructor.setAccessible(true);
                            instance = ucConstructor.newInstance(singleton);
                        }
                    }
                }
            }
            return instance;
        }

        //Singletons state that is to be used by subclasses
        protected String message;

        //Private constructor prevents external instantiation.
        private Singleton()
        {
            message = "Hello world!";
        }

        //Singleton's API. Implementation can be overwritten by subclasses.
        public String getMessage() {
            return message;
        }
}

//Just a small test program
public class ExtendingSingletonExample {

    public static void main(String args[]){

        //just uncomment one of following lines to change singleton class

        //Singleton.setKind(SingletonKind.UPPERCASE);
        //Singleton.setKind(SingletonKind.LOWERCASE);

        Singleton singleton = null;
        try {
            singleton = Singleton.getInstance();
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        } catch (InstantiationException e) {
            e.printStackTrace();
        }
        System.out.println(singleton.getMessage());
    }
}
```

Chapter 89: Autoboxing

Autoboxing is the automatic conversion that Java compiler makes between primitive types and their corresponding object wrapper classes. Example, converting int -> Integer, double -> Double... If the conversion goes the other way, this is called unboxing. Typically, this is used in Collections that cannot hold other than Objects, where boxing primitive types is needed before setting them in the collection.

Section 89.1: Using int and Integer interchangeably

As you use generic types with utility classes, you may often find that number types aren't very helpful when specified as the object types, as they aren't equal to their primitive counterparts.

```
List<Integer> ints = new ArrayList<Integer>();
```

Version ≥ Java SE 7

```
List<Integer> ints = new ArrayList<>();
```

Fortunately, expressions that evaluate to `int` can be used in place of an `Integer` when it is needed.

```
for (int i = 0; i < 10; i++)
    ints.add(i);
```

The `ints.add(i);` statement is equivalent to:

```
ints.add(Integer.valueOf(i));
```

And retains properties from `Integer#valueOf` such as having the same `Integer` objects cached by the JVM when it is within the number caching range.

This also applies to:

- `byte` and `Byte`
- `short` and `Short`
- `float` and `Float`
- `double` and `Double`
- `long` and `Long`
- `char` and `Character`
- `boolean` and `Boolean`

Care must be taken, however, in ambiguous situations. Consider the following code:

```
List<Integer> ints = new ArrayList<Integer>();
ints.add(1);
ints.add(2);
ints.add(3);
ints.remove(1); // ints is now [1, 3]
```

The `java.util.List` interface contains both a `remove(int index)` (`List` interface method) and a `remove(Object o)` (method inherited from `java.util.Collection`). In this case no boxing takes place and `remove(int index)` is called.

One more example of strange Java code behavior caused by autoboxing Integers with values in range from `-128` to `127`:

```
Integer a = 127;
Integer b = 127;
Integer c = 128;
Integer d = 128;
System.out.println(a == b); // true
System.out.println(c <= d); // true
System.out.println(c >= d); // true
System.out.println(c == d); // false
```

This happens because `>=` operator implicitly calls `intValue()` which returns **`int`** while `==` compares **references**, not the **`int`** values.

By default, Java caches values in range `[-128, 127]`, so the operator `==` works because the `Integers` in this range reference to the same objects if their values are same. Maximal value of the cacheable range can be defined with `-XX:AutoBoxCacheMax` JVM option. So, if you run the program with `-XX:AutoBoxCacheMax=1000`, the following code will print **`true`**:

```
Integer a = 1000;
Integer b = 1000;
System.out.println(a == b); // true
```

Section 89.2: Auto-unboxing may lead to NullPointerException

This code compiles:

```
Integer arg = null;
int x = arg;
```

But it will crash at runtime with a `java.lang.NullPointerException` on the second line.

The problem is that a primitive **`int`** cannot have a **`null`** value.

This is a minimalistic example, but in practice it often manifests in more sophisticated forms. The `NullPointerException` is not very intuitive and is often little help in locating such bugs.

Rely on autoboxing and auto-unboxing with care, make sure that unboxed values will not have **`null`** values at runtime.

Section 89.3: Using Boolean in if statement

Due to auto unboxing, one can use a `Boolean` in an `if` statement:

```
Boolean a = Boolean.TRUE;
if (a) { // a gets converted to boolean
    System.out.println("It works!");
}
```

That works for **`while`**, **`do while`** and the condition in the **`for`** statements as well.

Note that, if the `Boolean` is **`null`**, a `NullPointerException` will be thrown in the conversion.

Section 89.4: Different Cases When Integer and int can be used interchangeably

Case 1: While using in the place of method arguments.

If a method requires an object of wrapper class as argument.Then interchangeably the argument can be passed a variable of the respective primitive type and vice versa.

Example:

```
int i;
Integer j;
void ex_method(Integer i)//Is a valid statement
void ex_method1(int j)//Is a valid statement
```

Case 2: While passing return values:

When a method returns a primitive type variable then an object of corresponding wrapper class can be passed as the return value interchangeably and vice versa.

Example:

```
int i;
Integer j;
int ex_method()
{...
return j;}//Is a valid statement
Integer ex_method1()
{...
return i;//Is a valid statement
}
```

Case 3: While performing operations.

Whenever performing operations on numbers the primitive type variable and object of respective wrapper class can be used interchangeably.

```
int i=5;
Integer j=new Integer(7);
int k=i+j;//Is a valid statement
Integer m=i+j;//Is also a valid statement
```

Pitfall:Remember to initialize or assign a value to an object of the wrapper class.

While using wrapper class object and primitive variable interchangeably never forget or miss to initialize or assign a value to the wrapper class object else it may lead to null pointer exception at runtime.

Example:

```
public class Test{
    Integer i;
    int j;
    public void met()
    {j=i;//Null pointer exception
    SOP(j);
    SOP(i);}
    public static void main(String[] args)
    {Test t=new Test();
    t.go();//Null pointer exception
    }
```

In the above example, the value of the object is unassigned and uninitialized and thus at runtime the program will

run into null pointer exception.So as clear from the above example the value of object should never be left uninitialized and unassigned.

Section 89.5: Memory and Computational Overhead of Autoboxing

Autoboxing can come at a substantial memory overhead. For example:

```
Map<Integer, Integer> square = new HashMap<Integer, Integer>();
for(int i = 256; i < 1024; i++) {
    square.put(i, i * i); // Autoboxing of large integers
}
```

will typically consume substantial amount of memory (about 60kb for 6k of actual data).

Furthermore, boxed integers usually require additional round-trips in the memory, and thus make CPU caches less effective. In above example, the memory accessed is spread out to five different locations that may be in entirely different regions of the memory: 1. the `HashMap` object, 2. the map's `Entry[] table` object, 3. the `Entry` object, 4. the entrys `key` object (boxing the primitive key), 5. the entrys `value` object (boxing the primitive value).

```
class Example {
  int primitive; // Stored directly in the class `Example`
  Integer boxed; // Reference to another memory location
}
```

Reading `boxed` requires two memory accesses, accessing `primitive` only one.

When getting data from this map, the seemingly innocent code

```
int sumOfSquares = 0;
for(int i = 256; i < 1024; i++) {
    sumOfSquares += square.get(i);
}
```

is equivalent to:

```
int sumOfSquares = 0;
for(int i = 256; i < 1024; i++) {
    sumOfSquares += square.get(Integer.valueOf(i)).intValue();
}
```

Typically, the above code causes the *creation and garbage collection* of an `Integer` object for every `Map#get(Integer)` operation. (See Note below for more details.)

To reduce this overhead, several libraries offer optimized collections for primitive types that do *not* require boxing. In addition to avoiding the boxing overhead, these collection will require about 4x less memory per entry. While Java Hotspot *may* be able to optimize the autoboxing by working with objects on the stack instead of the heap, it is not possible to optimize the memory overhead and resulting memory indirection.

Java 8 streams also have optimized interfaces for primitive data types, such as `IntStream` that do not require boxing.

Note: a typical Java runtime maintains a simple cache of `Integer` and other primitive wrapper object that is used by the `valueOf` factory methods, and by autoboxing. For `Integer`, the default range of this cache is -128 to +127. Some JVMs provide a JVM command-line option for changing the cache size / range.

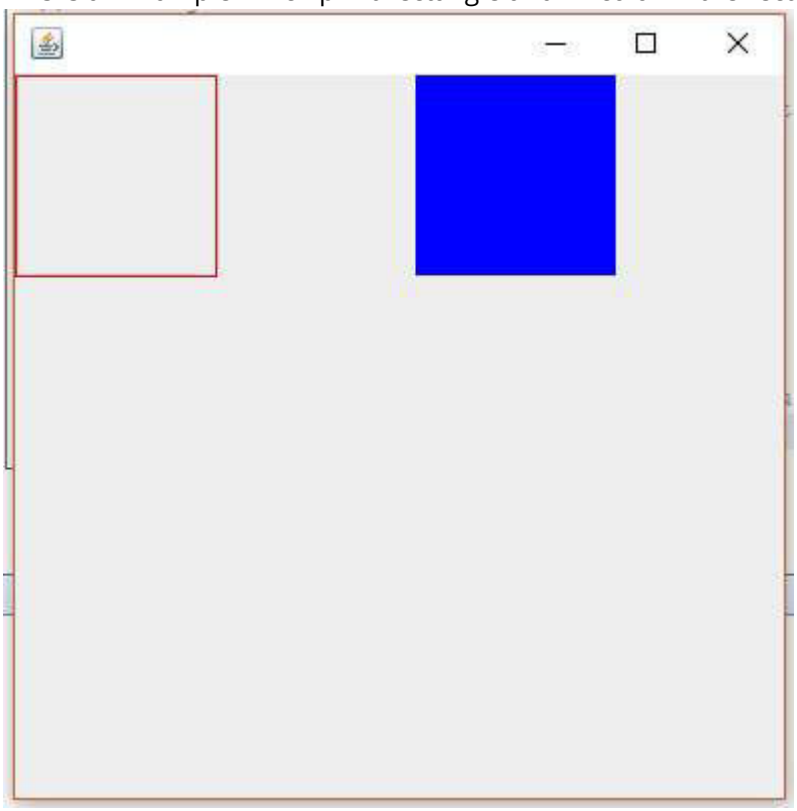
Chapter 90: 2D Graphics in Java

Graphics are visual images or designs on some surface, such as a wall, canvas, screen, paper, or stone to inform, illustrate, or entertain. It includes: pictorial representation of data, as in computer-aided design and manufacture, in typesetting and the graphic arts, and in educational and recreational software. Images that are generated by a computer are called computer graphics.

The Java 2D API is powerful and complex. There are multiple ways to do 2D graphics in Java.

Section 90.1: Example 1: Draw and Fill a Rectangle Using Java

This is an Example which print rectangle and fill color in the rectangle. https://i.stack.imgur.com/dlC5v.jpg



Most methods of the Graphics class can be divided into two basic groups:

1. Draw and fill methods, enabling you to render basic shapes, text, and images
2. Attributes setting methods, which affect how that drawing and filling appears

Code Example: Let us start this with a little example of drawing a rectangle and filling color in it. There we declare two classes, one class is MyPanel and other Class is Test. In class MyPanel we use drawRect( ) & fillRect( ) mathods to draw rectangle and fill Color in it. We set the color by setColor(Color.blue) method. In Second Class we Test our graphic which is Test Class we make a Frame and put MyPanel with p=new MyPanel() object in it.By running Test Class we see a Rectangle and a Blue Color Filled Rectangle.

First Class: MyPanel

```
import javax.swing.*;
import java.awt.*;
// MyPanel extends JPanel, which will eventually be placed in a JFrame
public class MyPanel extends JPanel {
    // custom painting is performed by the paintComponent method
    @Override
    public void paintComponent(Graphics g){
```

```
        // clear the previous painting
        super.paintComponent(g);
        // cast Graphics to Graphics2D
        Graphics2D g2 = (Graphics2D) g;
        g2.setColor(Color.red); // sets Graphics2D color
        // draw the rectangle
        g2.drawRect(0,0,100,100); // drawRect(x-position, y-position, width, height)
        g2.setColor(Color.blue);
        g2.fillRect(200,0,100,100); // fill new rectangle with color blue
    }
}
```

Second Class: Test

```
import javax.swing.*;
import java.awt.*;
public class Test { //the Class by which we display our rectangle
    JFrame f;
    MyPanel p;
    public Test(){
        f = new JFrame();
        // get the content area of Panel.
        Container c = f.getContentPane();
        // set the LayoutManager
        c.setLayout(new BorderLayout());
        p = new MyPanel();
        // add MyPanel object into container
        c.add(p);
        // set the size of the JFrame
        f.setSize(400,400);
        // make the JFrame visible
        f.setVisible(true);
        // sets close behavior; EXIT_ON_CLOSE invokes System.exit(0) on closing the JFrame
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    public static void main(String args[ ]){
        Test t = new Test();
    }
}
```

For More Explanation about Border Layout: https://docs.oracle.com/javase/tutorial/uiswing/layout/border.html

paintComponent( )

- It is a main method for painting
- By default, it first paints the background
- After that, it performs custom painting (drawing circle, rectangles etc.)

Graphic2D refers Graphic2D Class

Note: The Java 2D API enables you to easily perform the following tasks:

- Draw lines, rectangles and any other geometric shape.
- Fill those shapes with solid colors or gradients and textures.
- Draw text with options for fine control over the font and rendering process.
- Draw images, optionally applying filtering operations.
- Apply operations such as compositing and transforming during any of the above rendering operations.

Section 90.2: Example 2: Drawing and Filling Oval

```
import javax.swing.*;
import java.awt.*;

public class MyPanel extends JPanel {
    @Override
    public void paintComponent(Graphics g){
        // clear the previous painting
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D)g;
        g2.setColor(Color.blue);
        g2.drawOval(0, 0, 20,20);
        g2.fillOval(50,50,20,20);
    }
}
```

g2.drawOval(int x,int y,int height, int width);
This method will draw an oval at specified x and y position with given height and width.

g2.fillOval(int x,int y,int height, int width); This method will fill an oval at specified x and y position with given height and width.

Chapter 91: JAXB

Parameter	Details
fileObjOfXML	`File` object of an XML file
className	Name of a class with `.class` extension

JAXB or Java Architecture for XML Binding (JAXB) is a software framework that allows Java developers to map Java classes to XML representations. This Page will introduce readers to JAXB using detailed examples about its functions provided mainly for marshaling and un-marshaling Java Objects into xml format and vice-versa.

Section 91.1: Reading an XML file (unmarshalling)

To read an XML file named `UserDetails.xml` with the below content

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<user>
    <name>Jon Skeet</name>
    <userID>8884321</userID>
</user>
```

We need a POJO class named `User.java` as below

```
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class User {

    private long userID;
    private String name;

    // getters and setters
}
```

Here we have created the variables and class name according to the XML nodes. To map them, we use the annotation `XmlRootElement` on the class.

```
public class XMLReader {
    public static void main(String[] args) {
        try {
            User user = JAXB.unmarshal(new File("UserDetails.xml"), User.class);
            System.out.println(user.getName());   // prints Jon Skeet
            System.out.println(user.getUserID()); // prints 8884321
        } catch (Exception e) {
            System.err.println("Exception occurred while reading the XML!");
        }
    }
}
```

Here `unmarshal()` method is used to parse the XML file. It takes the XML file name and the class type as two arguments. Then we can use the getter methods of the object to print the data.

Section 91.2: Writing an XML file (marshalling an object)

```
import javax.xml.bind.annotation.XmlRootElement;
```

```
@XmlRootElement
public class User {

    private long userID;
    private String name;

    // getters and setters
}
```

By using the annotation `XMLRootElement`, we can mark a class as a root element of an XML file.

```
import java.io.File;
import javax.xml.bind.JAXB;

public class XMLCreator {
    public static void main(String[] args) {
        User user = new User();
        user.setName("Jon Skeet");
        user.setUserID(8884321);

        try {
            JAXB.marshal(user, new File("UserDetails.xml"));
        } catch (Exception e) {
            System.err.println("Exception occurred while writing in XML!");
        } finally {
            System.out.println("XML created");
        }
    }
}
```

`marshal()` is used to write the object's content into an XML file. Here `user`object and a new `File` object are passed as arguments to the `marshal()`.

On successful execution, this creates an XML file named `UserDetails.xml` in the class-path with the below content.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<user>
    <name>Jon Skeet</name>
    <userID>8884321</userID>
</user>
```

Section 91.3: Manual field/property XML mapping configuration

Annotations `@XmlElement`, `@XmlAttribute` or `@XmlTransient` and other in package `javax.xml.bind.annotation` allow the programmer to specify which and how marked fields or properties should be serialized.

```
@XmlAccessorType(XmlAccessType.NONE) // we want no automatic field/property marshalling
public class ManualXmlElementsExample {

    @XmlElement
    private String field="field value";

    @XmlAttribute
    private String attribute="attr value";

    @XmlAttribute(name="differentAttribute")
    private String oneAttribute="other attr value";
```

```
    @XmlElement(name="different name")
    private String oneName="different name value";

    @XmlTransient
    private String transientField = "will not get serialized ever";


    @XmlElement
    public String getModifiedTransientValue() {
        return transientField.replace(" ever", ", unless in a getter");
    }

    public void setModifiedTransientValue(String val) {} // empty on purpose


    public static void main(String[] args) {
        try {
            JAXB.marshal(new ManualXmlElementsExample(), System.out);
        } catch (Exception e) {
            System.err.println("Exception occurred while writing in XML!");
        }
    }
}
```

Section 91.4: Binding an XML namespace to a serializable Java class

This is an example of a `package-info.java` file that binds an XML namespace to a serializable Java class. This should be placed in the same package as the Java classes that should be serialized using the namespace.

```
/**
 * A package containing serializable classes.
 */
@XmlSchema
(
    xmlns =
    {
        @XmlNs(prefix = MySerializableClass.NAMESPACE_PREFIX, namespaceURI =
MySerializableClass.NAMESPACE)
    },
    namespace = MySerializableClass.NAMESPACE,
    elementFormDefault = XmlNsForm.QUALIFIED
)
package com.test.jaxb;

import javax.xml.bind.annotation.XmlNs;
import javax.xml.bind.annotation.XmlNsForm;
import javax.xml.bind.annotation.XmlSchema;
```

Section 91.5: Using XmlAdapter to generate desired xml format

When desired XML format differs from Java object model, an XmlAdapter implementation can be used to transform model object into xml-format object and vice versa. This example demonstrates how to put a field's value into an attribute of an element with field's name.

```
public class XmlAdapterExample {
```

```
    @XmlAccessorType(XmlAccessType.FIELD)
    public static class NodeValueElement {

        @XmlAttribute(name="attrValue")
        String value;

        public NodeValueElement() {
        }

        public NodeValueElement(String value) {
            super();
            this.value = value;
        }

        public String getValue() {
            return value;
        }

        public void setValue(String value) {
            this.value = value;
        }
    }

    public static class ValueAsAttrXmlAdapter extends XmlAdapter<NodeValueElement, String> {

        @Override
        public NodeValueElement marshal(String v) throws Exception {
            return new NodeValueElement(v);
        }

        @Override
        public String unmarshal(NodeValueElement v) throws Exception {
            if (v==null) return "";
            return v.getValue();
        }
    }

    @XmlRootElement(name="DataObject")
    @XmlAccessorType(XmlAccessType.FIELD)
    public static class DataObject {

        String elementWithValue;

        @XmlJavaTypeAdapter(value=ValueAsAttrXmlAdapter.class)
        String elementWithAttribute;
    }

    public static void main(String[] args) {
        DataObject data = new DataObject();
        data.elementWithValue="value1";
        data.elementWithAttribute ="value2";

        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        JAXB.marshal(data, baos);

        String xmlString = new String(baos.toByteArray(), StandardCharsets.UTF_8);

        System.out.println(xmlString);
    }
}
```

Section 91.6: Using XmlAdapter to trim string

```
package com.example.xml.adapters;

import javax.xml.bind.annotation.adapters.XmlAdapter;

public class StringTrimAdapter extends XmlAdapter<String, String> {
    @Override
    public String unmarshal(String v) throws Exception {
        if (v == null)
            return null;
        return v.trim();
    }

    @Override
    public String marshal(String v) throws Exception {
        if (v == null)
            return null;
        return v.trim();
    }
}
```

And in package-info.java add following declaration.

```
@XmlJavaTypeAdapter(value = com.example.xml.adapters.StringTrimAdapter.class, type = String.class)
package com.example.xml.jaxb.bindings;// Packge where you intend to apply trimming filter

import javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter;
```

Section 91.7: Automatic field/property XML mapping configuration (@XmlAccessorType)

Annotation `@XmlAccessorType` determines whether fields/properties will be automatically serialized to XML. Note, that field and method annotations `@XmlElement`, `@XmlAttribute` or `@XmlTransient` take precedence over the default settings.

```
public class XmlAccessTypeExample {

@XmlAccessorType(XmlAccessType.FIELD)
static class AccessorExampleField {
    public String field="value1";

    public String getGetter() {
        return "getter";
    }

    public void setGetter(String value) {}
}

@XmlAccessorType(XmlAccessType.NONE)
static class AccessorExampleNone {
    public String field="value1";

    public String getGetter() {
        return "getter";
    }

    public void setGetter(String value) {}
}
```

```
@XmlAccessorType(XmlAccessType.PROPERTY)
static class AccessorExampleProperty {
    public String field="value1";

    public String getGetter() {
        return "getter";
    }

    public void setGetter(String value) {}
}

@XmlAccessorType(XmlAccessType.PUBLIC_MEMBER)
static class AccessorExamplePublic {
    public String field="value1";

    public String getGetter() {
        return "getter";
    }

    public void setGetter(String value) {}
}

public static void main(String[] args) {
    try {
        System.out.println("\nField:");
        JAXB.marshal(new AccessorExampleField(), System.out);
        System.out.println("\nNone:");
        JAXB.marshal(new AccessorExampleNone(), System.out);
        System.out.println("\nProperty:");
        JAXB.marshal(new AccessorExampleProperty(), System.out);
        System.out.println("\nPublic:");
        JAXB.marshal(new AccessorExamplePublic(), System.out);
    } catch (Exception e) {
        System.err.println("Exception occurred while writing in XML!");
    }
}

} // outer class end
```

Output

```
Field:
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

value1


None:
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>


Property:
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

getter


Public:
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

value1
```

```
getter
```

Section 91.8: Specifying a XmlAdapter instance to (re)use existing data

Sometimes specific instances of data should be used. Recreation is not desired and referencing **`static`** data would have a code smell.

It is possible to specify a `XmlAdapter` instance the `Unmarshaller` should use, which allows the user to use `XmlAdapters` with no zero-arg constructor and/or pass data to the adapter.

Example
User class

The following class contains a name and a user's image.

```
import java.awt.image.BufferedImage;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter;

@XmlRootElement
public class User {

    private String name;
    private BufferedImage image;

    @XmlAttribute
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @XmlJavaTypeAdapter(value=ImageCacheAdapter.class)
    @XmlAttribute
    public BufferedImage getImage() {
        return image;
    }

    public void setImage(BufferedImage image) {
        this.image = image;
    }

    public User(String name, BufferedImage image) {
        this.name = name;
        this.image = image;
    }

    public User() {
        this("", null);
    }

}
```

Adapter

To avoid creating the same image in memory twice (as well as downloading the data again), the adapter stores the images in a map.

Version ≤ Java SE 7

For valid Java 7 code replace the `getImage` method with

```
public BufferedImage getImage(URL url) {
    BufferedImage image = imageCache.get(url);
    if (image == null) {
        try {
            image = ImageIO.read(url);
        } catch (IOException ex) {
            Logger.getLogger(ImageCacheAdapter.class.getName()).log(Level.SEVERE, null, ex);
            return null;
        }
        imageCache.put(url, image);
        reverseIndex.put(image, url);
    }
    return image;
}
```

```
import java.awt.image.BufferedImage;
import java.io.IOException;
import java.net.URL;
import java.util.HashMap;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.imageio.ImageIO;
import javax.xml.bind.annotation.adapters.XmlAdapter;

public class ImageCacheAdapter extends XmlAdapter<String, BufferedImage> {

    private final Map<URL, BufferedImage> imageCache = new HashMap<>();
    private final Map<BufferedImage, URL> reverseIndex = new HashMap<>();

    public BufferedImage getImage(URL url) {
        // using a single lookup using Java 8 methods
        return imageCache.computeIfAbsent(url, s -> {
            try {
                BufferedImage img = ImageIO.read(s);
                reverseIndex.put(img, s);
                return img;
            } catch (IOException ex) {
                Logger.getLogger(ImageCacheAdapter.class.getName()).log(Level.SEVERE, null, ex);
                return null;
            }
        });
    }

    @Override
    public BufferedImage unmarshal(String v) throws Exception {
        return getImage(new URL(v));
    }

    @Override
    public String marshal(BufferedImage v) throws Exception {
        return reverseIndex.get(v).toExternalForm();
    }
```

```
}
```

Example XMLs

The following 2 xmls are for *Jon Skeet* and his earth 2 counterpart, which both look exactly the same and therefore use the same avatar.

```
<?xml version="1.0" encoding="UTF-8"?>

<user name="Jon Skeet"
image="https://www.gravatar.com/avatar/6d8ebb117e8d83d74ea95fbdd0f87e13?s=328&amp;d=identicon&amp;r
=PG"/>
```

```
<?xml version="1.0" encoding="UTF-8"?>

<user name="Jon Skeet (Earth 2)"
image="https://www.gravatar.com/avatar/6d8ebb117e8d83d74ea95fbdd0f87e13?s=328&amp;d=identicon&amp;r
=PG"/>
```

Using the adapter

```
ImageCacheAdapter adapter = new ImageCacheAdapter();

JAXBContext context = JAXBContext.newInstance(User.class);

Unmarshaller unmarshaller = context.createUnmarshaller();

// specifiy the adapter instance to use for every
// @XmlJavaTypeAdapter(value=ImageCacheAdapter.class)
unmarshaller.setAdapter(ImageCacheAdapter.class, adapter);

User result1 = (User) unmarshaller.unmarshal(Main.class.getResource("user.xml"));

// unmarshal second xml using the same adapter instance
Unmarshaller unmarshaller2 = context.createUnmarshaller();
unmarshaller2.setAdapter(ImageCacheAdapter.class, adapter);
User result2 = (User) unmarshaller2.unmarshal(Main.class.getResource("user2.xml"));

System.out.println(result1.getName());
System.out.println(result2.getName());

// yields true, since image is reused
System.out.println(result1.getImage() == result2.getImage());
```

Chapter 92: Class - Java Reflection

The java.lang.Class class provides many methods that can be used to get metadata, examine and change the run time behavior of a class.

The java.lang and java.lang.reflect packages provide classes for java reflection.

Where it is used

The Reflection API is mainly used in:

IDE (Integrated Development Environment) e.g. Eclipse, MyEclipse, NetBeans etc. Debugger Test Tools etc.

Section 92.1: getClass() method of Object class

```
class Simple { }

class Test {
     void printName(Object obj){
         Class c = obj.getClass();
         System.out.println(c.getName());
     }
     public static void main(String args[]){
         Simple s = new Simple();

         Test t = new Test();
         t.printName(s);
    }
}
```

Chapter 93: Networking

Section 93.1: Basic Client and Server Communication using a Socket

Server: Start, and wait for incoming connections

```
//Open a listening "ServerSocket" on port 1234.
ServerSocket serverSocket = new ServerSocket(1234);

while (true) {
    // Wait for a client connection.
    // Once a client connected, we get a "Socket" object
    // that can be used to send and receive messages to/from the newly
    // connected client
    Socket clientSocket = serverSocket.accept();

    // Here we'll add the code to handle one specific client.
}
```

Server: Handling clients

We'll handle each client in a separate thread so multiple clients could interact with the server at the same time. This technique works fine as long as the number of clients is low (<< 1000 clients, depending on the OS architecture and the expected load of each thread).

```
new Thread(() -> {
    // Get the socket's InputStream, to read bytes from the socket
    InputStream in = clientSocket.getInputStream();
    // wrap the InputStream in a reader so you can read a String instead of bytes
    BufferedReader reader = new BufferedReader(
            new InputStreamReader(in, StandardCharsets.UTF_8));
    // Read text from the socket and print line by line
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
    }).start();
```

Client: Connect to the server and send a message

```
// 127.0.0.1 is the address of the server (this is the localhost address; i.e.
// the address of our own machine)
// 1234 is the port that the server will be listening on
Socket socket = new Socket("127.0.0.1", 1234);

// Write a string into the socket, and flush the buffer
OutputStream outStream = socket.getOutputStream();
PrintWriter writer = new PrintWriter(
        new OutputStreamWriter(outStream, StandardCharsets.UTF_8));
writer.println("Hello world!");
writer.flush();
```

Closing Sockets and Handling Exceptions

The above examples left out some things to make them easier to read.

1. Just like files and other external resources, it's important we tell the OS when we're done with them. When

we're done with a socket, call `socket.close()` to properly close it.

2. Sockets handle I/O (Input/Output) operations that depend on a variety of external factors. For example what if the other side suddenly disconnects? What if there are network error? These things are beyond our control. This is why many socket operations might throw exceptions, especially `IOException`.

A more complete code for the client would therefore be something like this:

```
// "try-with-resources" will close the socket once we leave its scope
try (Socket socket = new Socket("127.0.0.1", 1234)) {
    OutputStream outStream = socket.getOutputStream();
    PrintWriter writer = new PrintWriter(
            new OutputStreamWriter(outStream, StandardCharsets.UTF_8));
    writer.println("Hello world!");
    writer.flush();
} catch (IOException e) {
    //Handle the error
}
```

Basic Server and Client - complete examples

Server:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.ServerSocket;
import java.net.Socket;
import java.nio.charset.StandardCharsets;

public class Server {
    public static void main(String args[]) {
        try (ServerSocket serverSocket = new ServerSocket(1234)) {
            while (true) {
                // Wait for a client connection.
                Socket clientSocket = serverSocket.accept();

                // Create and start a thread to handle the new client
                new Thread(() -> {
                    try {
                        // Get the socket's InputStream, to read bytes
                        // from the socket
                        InputStream in = clientSocket.getInputStream();
                        // wrap the InputStream in a reader so you can
                        // read a String instead of bytes
                        BufferedReader reader = new BufferedReader(
                             new InputStreamReader(in, StandardCharsets.UTF_8));
                        // Read from the socket and print line by line
                        String line;
                        while ((line = reader.readLine()) != null) {
                            System.out.println(line);
                        }
                    }
                    catch (IOException e) {
                        e.printStackTrace();
                    } finally {
                        // This finally block ensures the socket is closed.
                        // A try-with-resources block cannot be used because
                        // the socket is passed into a thread, so it isn't
```

```
                    // created and closed in the same block
                    try {
                        clientSocket.close();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            }).start();
        }
    }
    catch (IOException e) {
        e.printStackTrace();
    }

  }
}
```

Client:

```
import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.Socket;
import java.nio.charset.StandardCharsets;

public class Client {
    public static void main(String args[]) {
        try (Socket socket = new Socket("127.0.0.1", 1234)) {
            // We'll reach this code once we've connected to the server

            // Write a string into the socket, and flush the buffer
            OutputStream outStream = socket.getOutputStream();
            PrintWriter writer = new PrintWriter(
                    new OutputStreamWriter(outStream, StandardCharsets.UTF_8));
            writer.println("Hello world!");
            writer.flush();
        } catch (IOException e) {
            // Exception should be handled.
            e.printStackTrace();
        }
    }
}
```

Section 93.2: Basic Client/Server Communication using UDP (Datagram)

Client.java

```
import java.io.*;
import java.net.*;

public class Client{
    public static void main(String [] args) throws IOException{
        DatagramSocket clientSocket = new DatagramSocket();
        InetAddress address = InetAddress.getByName(args[0]);

        String ex = "Hello, World!";
        byte[] buf = ex.getBytes();
```

```
        DatagramPacket packet = new DatagramPacket(buf,buf.length, address, 4160);
        clientSocket.send(packet);
    }
}
```

In this case, we pass in the address of the server, via an argument (`args[0]`). The port we are using is 4160.

Server.java

```
import java.io.*;
import java.net.*;

public class Server{
    public static void main(String [] args) throws IOException{
        DatagramSocket serverSocket = new DatagramSocket(4160);

        byte[] rbuf = new byte[256];
        DatagramPacket packet = new DatagramPacket(rbuf, rbuf.length);
        serverSocket.receive(packet);
        String response = new String(packet.getData());
        System.out.println("Response: " + response);
    }
}
```

On the server-side, declare a DatagramSocket on the same port which we sent our message to (4160) and wait for a response.

Section 93.3: Loading TrustStore and KeyStore from InputStream

```
public class TrustLoader {

    public static void main(String args[]) {
        try {
                //Gets the inputstream of a a trust store file under ssl/rpgrenadesClient.jks
                //This path refers to the ssl folder in the jar file, in a jar file in the same
directory
                //as this jar file, or a different directory in the same directory as the jar file
                InputStream stream =
TrustLoader.class.getResourceAsStream("/ssl/rpgrenadesClient.jks");
                //Both trustStores and keyStores are represented by the KeyStore object
                KeyStore trustStore = KeyStore.getInstance(KeyStore.getDefaultType());
                //The password for the trustStore
                char[] trustStorePassword = "password".toCharArray();
                //This loads the trust store into the object
                trustStore.load(stream, trustStorePassword);

                //This is defining the SSLContext so the trust store will be used
                //Getting default SSLContext to edit.
                SSLContext context = SSLContext.getInstance("SSL");
                //TrustMangers hold trust stores, more than one can be added
                TrustManagerFactory factory =
TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
                //Adds the truststore to the factory
                factory.init(trustStore);
                //This is passed to the SSLContext init method
                TrustManager[] managers = factory.getTrustManagers();
                context.init(null, managers, null);
                //Sets our new SSLContext to be used.
                SSLContext.setDefault(context);
```

```
            } catch (KeyStoreException | IOException | NoSuchAlgorithmException
                    | CertificateException | KeyManagementException ex) {
                //Handle error
                ex.printStackTrace();
            }

    }
}
```

Intiating a KeyStore works the same, except replace any word `Trust` in a object name with `Key`. Additionally, the `KeyManager[]` array must be passed to the the first argument of `SSLContext.init`. That is `SSLContext.init(keyMangers, trustMangers, null)`

Section 93.4: Socket example - reading a web page using a simple socket

```
import java.io.*;
import java.net.Socket;

public class Main {

    public static void main(String[] args) throws IOException {//We don't handle Exceptions in this
example
        //Open a socket to stackoverflow.com, port 80
        Socket socket = new Socket("stackoverflow.com",80);

        //Prepare input, output stream before sending request
        OutputStream outStream = socket.getOutputStream();
        InputStream inStream = socket.getInputStream();
        BufferedReader reader = new BufferedReader(new InputStreamReader(inStream));
        PrintWriter writer = new PrintWriter(new BufferedOutputStream(outStream));

        //Send a basic HTTP header
        writer.print("GET / HTTP/1.1\nHost:stackoverflow.com\n\n");
        writer.flush();

        //Read the response
        System.out.println(readFully(reader));

        //Close the socket
        socket.close();
    }

    private static String readFully(Reader in) {
        StringBuilder sb = new StringBuilder();
        int BUFFER_SIZE=1024;
        char[] buffer = new char[BUFFER_SIZE]; // or some other size,
        int charsRead = 0;
        while ( (charsRead  = rd.read(buffer, 0, BUFFER_SIZE)) != -1) {
          sb.append(buffer, 0, charsRead);
        }
    }
}
```

You should get a response that starts with `HTTP/1.1 200 OK`, which indicates a normal HTTP response, followed by the rest of the HTTP header, followed by the raw web page in HTML form.

Note the `readFully()` method is important to prevent a premature EOF exception. The last line of the web page may be missing a return, to signal the end of line, then `readLine()` will complain, so one must read it by hand or

use utility methods from Apache commons-io IOUtils

This example is meant as a simple demonstration of connecting to an existing resource using a socket, it's not a practical way of accessing web pages. If you need to access a web page using Java, it's best to use an existing HTTP client library such as Apache's HTTP Client or Google's HTTP Client

Section 93.5: Temporarily disable SSL verification (for testing purposes)

Sometimes in a development or testing environment, the SSL certificate chain might not have been fully established (yet).

To continue developing and testing, you can turn off SSL verification programmatically by installing an "all-trusting" trust manager:

```
try {
   // Create a trust manager that does not validate certificate chains
   TrustManager[] trustAllCerts = new TrustManager[] {
      new X509TrustManager() {
       public X509Certificate[] getAcceptedIssuers() {
           return null;
       }
       public void checkClientTrusted(X509Certificate[] certs, String authType) {
       }
       public void checkServerTrusted(X509Certificate[] certs, String authType) {
       }
      }
   };

   // Install the all-trusting trust manager
   SSLContext sc = SSLContext.getInstance("SSL");
   sc.init(null, trustAllCerts, new java.security.SecureRandom());
   HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());

   // Create all-trusting host name verifier
   HostnameVerifier allHostsValid = new HostnameVerifier() {
       public boolean verify(String hostname, SSLSession session) {
           return true;
       }
   };

   // Install the all-trusting host verifier
   HttpsURLConnection.setDefaultHostnameVerifier(allHostsValid);
} catch (NoSuchAlgorithmException | KeyManagementException e) {
    e.printStackTrace();
}
```

Section 93.6: Downloading a file using Channel

If the file already exists, it will be overwritten!

```
String fileName     = "file.zip";                  // name of the file
String urlToGetFrom = "http://www.mywebsite.com/"; // URL to get it from
String pathToSaveTo = "C:\\Users\\user\\";         // where to put it

//If the file already exists, it will be overwritten!

//Opening OutputStream to the destination file
```

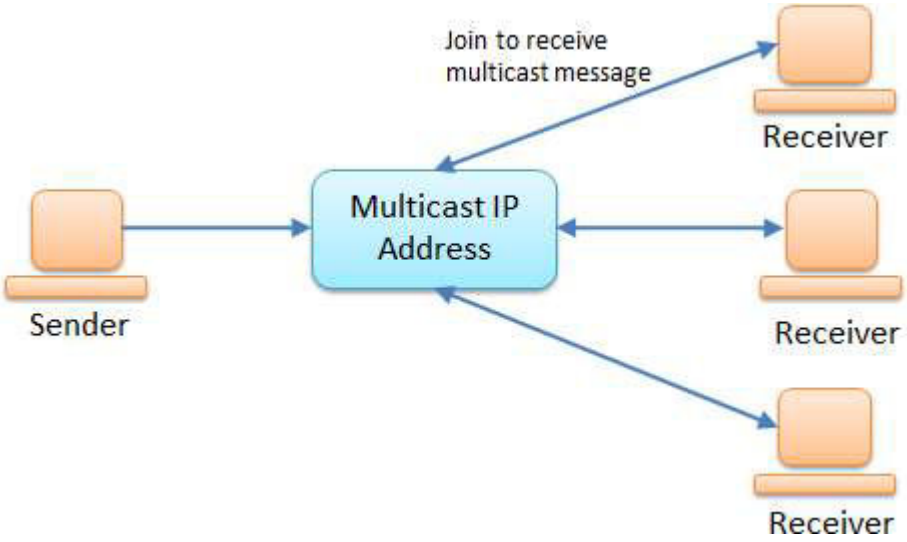
```
try (ReadableByteChannel rbc =
      Channels.newChannel(new URL(urlToGetFrom + fileName).openStream()) ) {
    try ( FileChannel channel =
        new FileOutputStream(pathToSaveTo + fileName).getChannel(); ) {
      channel.transferFrom(rbc, 0, Long.MAX_VALUE);
    }
    catch (FileNotFoundException e) { /* Output directory not found */ }
    catch (IOException e)           { /* File IO error */ }
}
catch (MalformedURLException e)     { /* URL is malformed */ }
catch (IOException e)               { /* IO error connecting to website */ }
```

Notes

- Don't leave the catch blocks empty!
- In case of error, check if the remote file exists
- This is a blocking operation, can take long time with large files

Section 93.7: Multicasting

Multicasting is a type of Datagram Socket. Unlike regular Datagrams, Multicasting doesn't handle each client individually instead it sends it out to one IP Address and all subscribed clients will get the message.



Example code for a server side:

```
public class Server {

    private DatagramSocket serverSocket;

    private String ip;

    private int port;

    public Server(String ip, int port) throws SocketException, IOException{
        this.ip = ip;
        this.port = port;
        // socket used to send
        serverSocket = new DatagramSocket();
    }

    public void send() throws IOException{
        // make datagram packet
        byte[] message = ("Multicasting...").getBytes();
```

```
        DatagramPacket packet = new DatagramPacket(message, message.length,
            InetAddress.getByName(ip), port);
        // send packet
        serverSocket.send(packet);
    }

    public void close(){
        serverSocket.close();
    }
}
```

Example code for a client side:

```
public class Client {

    private MulticastSocket socket;

    public Client(String ip, int port) throws IOException {

        // important that this is a multicast socket
        socket = new MulticastSocket(port);

        // join by ip
        socket.joinGroup(InetAddress.getByName(ip));
    }

    public void printMessage() throws IOException{
        // make datagram packet to recieve
        byte[] message = new byte[256];
        DatagramPacket packet = new DatagramPacket(message, message.length);

        // recieve the packet
        socket.receive(packet);
        System.out.println(new String(packet.getData()));
    }

    public void close(){
        socket.close();
    }
}
```

Code for running the Server:

```
public static void main(String[] args) {
    try {
        final String ip = args[0];
        final int port = Integer.parseInt(args[1]);
        Server server = new Server(ip, port);
        server.send();
        server.close();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}
```

Code for running a Client:

```
public static void main(String[] args) {
    try {
        final String ip = args[0];
```

```
        final int port = Integer.parseInt(args[1]);
        Client client = new Client(ip, port);
        client.printMessage();
        client.close();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}
```

Run the Client First: The Client must subscribe to the IP before it can start receiving any packets. If you start the server and call the `send()` method, and then make a client (& call `printMessage()`). Nothing will happen because the client connected after the message was sent.

Chapter 94: NIO - Networking

Section 94.1: Using Selector to wait for events (example with OP_CONNECT)

NIO appeared in Java 1.4 and introduced the concept of "Channels", which are supposed to be faster than regular I/O. Network-wise, the `SelectableChannel` is the most interesting as it allows to monitor different states of the Channel. It works in a similar manner as the C `SELECT()` system call: we get woken-up when certain types of events occur:

- connection received (`OP_ACCEPT`)
- connection realized (`OP_CONNECT`)
- data available in read FIFO (`OP_READ`)
- data can be pushed to write FIFO (`OP_WRITE`)

It allows for separation between *detecting* socket I/O (something can be read/written/...) and *performing* the I/O (read/write/...). Especially, all I/O detection can be done in a single thread for multiple sockets (clients), while performing I/O can be handled in a thread pool or anywhere else. That allows for an application to scale easily to the number of connected clients.

The following example shows the basics:

1. Create a `Selector`
2. Create a `SocketChannel`
3. Register the `SocketChannel`to the `Selector`
4. Loop with the `Selector` to detect events

```
Selector sel = Selector.open(); // CREATE the Selector
SocketChannel sc = SocketChannel.open(); // CREATE a SocketChannel
sc.configureBlocking(FALSE); // ... non blocking
sc.setOption(StandardSocketOptions.SO_KEEPALIVE, TRUE); // ... SET SOME options

// Register the Channel TO the Selector FOR wake-up ON CONNECT event AND USE SOME description AS an
attachement
sc.register(sel, SelectionKey.OP_CONNECT, "Connection to google.com"); // RETURNS a SelectionKey:
the association BETWEEN the SocketChannel AND the Selector
System.out.println("Initiating connection");
IF (sc.connect(NEW InetSocketAddress("www.google.com", 80)))
    System.out.println("Connected"); // Connected right-away: nothing ELSE TO do
ELSE {
    BOOLEAN exit = FALSE;
    while (!exit) {
        IF (sel.select(100) == 0) // Did something happen ON SOME registered Channels during the
LAST 100ms?
            continue; // No, wait SOME more

        // Something happened...
        Set<SelectionKey> KEYS = sel.selectedKeys(); // List OF SelectionKeys ON which SOME
registered operation was triggered
        FOR (SelectionKey k : KEYS) {
            System.out.println("Checking "+k.attachment());
            IF (k.isConnectable()) { // CONNECT event
                System.out.print("Connected through select() on "+k.channel()+" -> ");
                IF (sc.finishConnect()) { // Finish connection process
                    System.out.println("done!");
                    k.interestOps(k.interestOps() & ~SelectionKey.OP_CONNECT); // We are already
connected: remove interest IN CONNECT event
```

```
                    exit = TRUE;
                } ELSE
                    System.out.println("unfinished...");
            }
            // TODO: ELSE IF (k.isReadable()) { ...
        }
        KEYS.clear(); // Have TO clear the selected KEYS SET once processed!
    }
}
System.out.print("Disconnecting ... ");
sc.shutdownOutput(); // Initiate graceful disconnection
// TODO: empty receive buffer
sc.close();
System.out.println("done");
```

Would give the following output:

```
Initiating connection
Checking Connection to google.com
Connected through 'select()' on java.nio.channels.SocketChannel[connection-pending
remote=www.google.com/216.58.208.228:80] -> done!
Disconnecting ... done
```

Chapter 95: HttpURLConnection

Section 95.1: Get response body from a URL as a String

```
String getText(String url) throws IOException {
    HttpURLConnection connection = (HttpURLConnection) new URL(url).openConnection();
    //add headers to the connection, or check the status if desired..

    // handle error response code it occurs
    int responseCode = conn.getResponseCode();
    InputStream inputStream;
    if (200 <= responseCode && responseCode <= 299) {
        inputStream = connection.getInputStream();
    } else {
        inputStream = connection.getErrorStream();
    }

    BufferedReader in = new BufferedReader(
        new InputStreamReader(
            inputStream));

    StringBuilder response = new StringBuilder();
    String currentLine;

    while ((currentLine = in.readLine()) != null)
        response.append(currentLine);

    in.close();

    return response.toString();
}
```

This will download text data from the specified URL, and return it as a String.

How this works:

- First, we create a `HttpUrlConnection` from our URL, with `new URL(url).openConnection()`. We cast the `UrlConnection` this returns to a `HttpUrlConnection`, so we have access to things like adding headers (such as User Agent), or checking the response code. (This example does not do that, but it's easy to add.)
- Then, create `InputStream` basing on the response code (for error handling)
- Then, create a `BufferedReader` which allows us to read text from `InputStream` we get from the connection.
- Now, we append the text to a `StringBuilder`, line by line.
- Close the `InputStream`, and return the String we now have.

Notes:

- This method will throw an `IoException` in case of failure (such as a network error, or no internet connection), and it will also throw an *unchecked* `MalformedUrlException` if the given URL is not valid.
- It can be used for reading from any URL which returns text, such as webpages (HTML), REST APIs which return JSON or XML, etc.
- See also: Read URL to String in few lines of Java code.

Usage:

Is very simple:

```
String text = getText("http://example.com");
//Do something with the text from example.com, in this case the HTML.
```

Section 95.2: POST data

```
public static void post(String url, byte [] data, String contentType) throws IOException {
    HttpURLConnection connection = null;
    OutputStream out = null;
    InputStream in = null;

    try {
        connection = (HttpURLConnection) new URL(url).openConnection();
        connection.setRequestProperty("Content-Type", contentType);
        connection.setDoOutput(true);

        out = connection.getOutputStream();
        out.write(data);
        out.close();

        in = connection.getInputStream();
        BufferedReader reader = new BufferedReader(new InputStreamReader(in));
        String line = null;
        while ((line = reader.readLine()) != null) {
            System.out.println(line);
        }
        in.close();

    } finally {
        if (connection != null) connection.disconnect();
        if (out != null) out.close();
        if (in != null) in.close();
    }
}
```

This will POST data to the specified URL, then read the response line-by-line.

How it works

- As usual we obtain the `HttpURLConnection` from a `URL`.
- Set the content type using `setRequestProperty`, by default it's `application/x-www-form-urlencoded`
- `setDoOutput(true)` tells the connection that we will send data.
- Then we obtain the `OutputStream` by calling `getOutputStream()` and write data to it. Don't forget to close it after you are done.
- At last we read the server response.

Section 95.3: Delete resource

```
   public static void delete (String urlString, String contentType) throws IOException {
        HttpURLConnection connection = null;

        try {
            URL url = new URL(urlString);
            connection = (HttpURLConnection) url.openConnection();
            connection.setDoInput(true);
```

```
            connection.setRequestMethod("DELETE");
            connection.setRequestProperty("Content-Type", contentType);

            Map<String, List<String>> map = connection.getHeaderFields();
            StringBuilder sb = new StringBuilder();
            Iterator<Map.Entry<String, String>> iterator = responseHeader.entrySet().iterator();
            while(iterator.hasNext())
            {
                Map.Entry<String, String> entry = iterator.next();
                sb.append(entry.getKey());
                sb.append('=').append('"');
                sb.append(entry.getValue());
                sb.append('"');
                if(iterator.hasNext())
                {
                    sb.append(',').append(' ');
                }
            }
            System.out.println(sb.toString());

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (connection != null) connection.disconnect();
        }
    }
```

This will DELETE the resource in the specified URL, then print the response header.

How it works

- we obtain the `HttpURLConnection` from a `URL`.
- Set the content type using `setRequestProperty`, by default it's `application/x-www-form-urlencoded`
- `setDoInput(true)` tells the connection that we intend to use the URL connection for input.
- `setRequestMethod("DELETE")` to perform HTTP DELETE

At last we print the server response header.

Section 95.4: Check if resource exists

```
/**
 * Checks if a resource exists by sending a HEAD-Request.
 * @param url The url of a resource which has to be checked.
 * @return true if the response code is 200 OK.
 */
public static final boolean checkIfResourceExists(URL url) throws IOException {
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    conn.setRequestMethod("HEAD");
    int code = conn.getResponseCode();
    conn.disconnect();
    return code == 200;
}
```

Explanation:

If you are just checking if a resource exists, it better to use a HEAD request than a GET. This avoids the overhead of transferring the resource.

Note that the method only returns `true` if the response code is `200`. If you anticipate redirect (i.e. 3XX) responses, then the method may need to be enhanced to honor them.

Example:

```
checkIfResourceExists(new URL("http://images.google.com/")); // true
checkIfResourceExists(new URL("http://pictures.google.com/")); // false
```

Chapter 96: JAX-WS

Section 96.1: Basic Authentication

The way to do a JAX-WS call with basic authentication is a little unobvious.

Here is an example where `Service` is the service class representation and `Port` is the service port you want to access.

```
Service s = new Service();
Port port = s.getPort();

BindingProvider prov = (BindingProvider)port;
prov.getRequestContext().put(BindingProvider.USERNAME_PROPERTY, "myusername");
prov.getRequestContext().put(BindingProvider.PASSWORD_PROPERTY, "mypassword");

port.call();
```

Chapter 97: Nashorn JavaScript engine

Nashorn is a JavaScript engine developed in Java by Oracle, and has been released with Java 8. Nashorn allows embedding Javascript in Java applications via JSR-223 and allows to develop standalone Javascript applications, and it provides better runtime performance and better compliance with the ECMA normalized Javascript specification.

Section 97.1: Execute JavaScript file

```
// Required imports
import javax.script.ScriptEngineManager;
import javax.script.ScriptEngine;
import javax.script.ScriptException;
import java.io.FileReader;
import java.io.FileNotFoundException;

// Obtain an instance of the JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

// Load and execute a script from the file 'demo.js'
try {
    engine.eval(new FileReader("demo.js"));
} catch (FileNotFoundException ex) {
    ex.printStackTrace();
} catch (ScriptException ex) {
    // This is the generic Exception subclass for the Scripting API
    ex.printStackTrace();
}

// Outcome:
// 'Script from file!' printed on standard output
```

demo.js:

```
print('Script from file!');
```

Section 97.2: Intercept script output

```
// Obtain an instance of JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

// Setup a custom writer
StringWriter stringWriter = new StringWriter();
// Modify the engine context so that the custom writer is now the default
// output writer of the engine
engine.getContext().setWriter(stringWriter);

// Execute some script
try {
    engine.eval("print('Redirected text!');");
} catch (ScriptException ex) {
    ex.printStackTrace();
}

// Outcome:
// Nothing printed on standard output, but
```

```
// stringWriter.toString() contains 'Redirected text!'
```

Section 97.3: Hello Nashorn

```
// Obtain an instance of JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

// Execute an hardcoded script
try {
    engine.eval("print('Hello Nashorn!');");
} catch (ScriptException ex) {
    // This is the generic Exception subclass for the Scripting API
    ex.printStackTrace();
}

// Outcome:
// 'Hello Nashorn!' printed on standard output
```

Section 97.4: Evaluate Arithmetic Strings

```
// Obtain an instance of JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("JavaScript");

//String to be evaluated
String str = "3+2*4+5";
//Value after doing Arithmetic operation with operator precedence will be 16

//Printing the value
try {
    System.out.println(engine.eval(str));
} catch (ScriptException ex) {
    ex.printStackTrace();
}

//Outcome:
//Value of the string after arithmetic evaluation is printed on standard output.
//In this case '16.0' will be printed on standard output.
```

Section 97.5: Set global variables

```
// Obtain an instance of JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

// Define a global variable
engine.put("textToPrint", "Data defined in Java.");

// Print the global variable
try {
    engine.eval("print(textToPrint);");
} catch (ScriptException ex) {
    ex.printStackTrace();
}

// Outcome:
// 'Data defined in Java.' printed on standard output
```

Section 97.6: Set and get global variables

```
// Obtain an instance of JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

try {
    // Set value in the global name space of the engine
    engine.put("name","Nashorn");
    // Execute an hardcoded script
    engine.eval("var value='Hello '+name+'!';");
    // Get value
    String value=(String)engine.get("value");
    System.out.println(value);
} catch (ScriptException ex) {
    // This is the generic Exception subclass for the Scripting API
    ex.printStackTrace();
}

// Outcome:
// 'Hello Nashorn!' printed on standard output
```

Section 97.7: Usage of Java objects in JavaScript in Nashorn

It's possible to pass Java objects to Nashorn engine to be processed in Java code. At the same time, there are some JavaScript (and Nashorn) specific constructions, and it's not always clear how they work with java objects.

Below there is a table which describes behaviour of native Java objects inside JavaScript constructions.

Tested constructions:

1. Expression in if clause. In JS expression in if clause doesn't have to be boolean unlike Java. It's evaluated as false for so called falsy values (null, undefined, 0, empty strings etc)
2. for each statement Nashorn has a special kind of loop - for each - which can iterate over different JS and Java object.
3. Getting object size. In JS objects have a property length, which returns size of an array or a string.

Results:

Type	If	for each	.length
Java null	false	No iterations	**Exception**
Java empty string	false	No iterations	0
Java string	true	Iterates over string characters	Length of the string
Java Integer/Long	value != 0	No iterations	undefined
Java ArrayList	true	Iterates over elements	Length of the list
Java HashMap	true	Iterates over values	null
Java HashSet	true	Iterates over items	undefined

Recommendatons:

- It's advisable to use `if (some_string)` to check if a string is not null and not empty
- `for each` can be safely used to iterate over any collection, and it doesn't raise exceptions if the collection is not iterable, null or undefined
- Before getting length of an object it must be checked for null or undefined (the same is true for any attempt of calling a method or getting a property of Java object)

Section 97.8: Implementing an interface from script

```
import java.io.FileReader;
import java.io.IOException;

import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class InterfaceImplementationExample {
    public static interface Pet {
        public void eat();
    }

    public static void main(String[] args) throws IOException {
        // Obtain an instance of JavaScript engine
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("nashorn");

        try {
            //evaluate a script
            /* pet.js */
            /*
                var Pet = Java.type("InterfaceImplementationExample.Pet");

                new Pet() {
                    eat: function() { print("eat"); }
                }
            */

            Pet pet = (Pet) engine.eval(new FileReader("pet.js"));

            pet.eat();
        } catch (ScriptException ex) {
            ex.printStackTrace();
        }

        // Outcome:
        // 'eat' printed on standard output
    }
}
```

Chapter 98: Java Native Interface

Parameter	Details
JNIEnv	Pointer to the JNI environment
jobject	The object which invoked the non-`static` `native` method
jclass	The class which invoked the `static` `native` method

Section 98.1: Calling C++ methods from Java

Static and member methods in Java can be marked as *native* to indicate that their implementation is to be found in a shared library file. Upon execution of a native method, the JVM looks for a corresponding function in loaded libraries (see Loading native libraries), using a simple name mangling scheme, performs argument conversion and stack setup, then hands over control to native code.

Java code

```
/*** com/example/jni/JNIJava.java **/

package com.example.jni;

public class JNIJava {
    static {
        System.loadLibrary("libJNI_CPP");
    }

    // Obviously, native methods may not have a body defined in Java
    public native void printString(String name);
    public static native double average(int[] nums);

    public static void main(final String[] args) {
        JNIJava jniJava = new JNIJava();
        jniJava.printString("Invoked C++ 'printString' from Java");

        double d = average(new int[]{1, 2, 3, 4, 7});
        System.out.println("Got result from C++ 'average': " + d);
    }
}
```

C++ code

Header files containing native function declarations should be generated using the `javah` tool on target classes. Running the following command at the build directory :

```
javah -o com_example_jni_JNIJava.hpp com.example.jni.JNIJava
```

... produces the following header file (*comments stripped for brevity*) :

```
// com_example_jni_JNIJava.hpp

/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h> // The JNI API declarations

#ifndef _Included_com_example_jni_JNIJava
#define _Included_com_example_jni_JNIJava
#ifdef __cplusplus
extern "C" { // This is absolutely required if using a C++ compiler
#endif
```

```
JNIEXPORT void JNICALL Java_com_example_jni_JNIJava_printString
  (JNIEnv *, jobject, jstring);

JNIEXPORT jdouble JNICALL Java_com_example_jni_JNIJava_average
  (JNIEnv *, jclass, jintArray);

#ifdef __cplusplus
}
#endif
#endif
```

Here is an example implementation :

```
// com_example_jni_JNIJava.cpp

#include <iostream>
#include "com_example_jni_JNIJava.hpp"

using namespace std;

JNIEXPORT void JNICALL Java_com_example_jni_JNIJava_printString(JNIEnv *env, jobject jthis, jstring
string) {
    const char *stringInC = env->GetStringUTFChars(string, NULL);
    if (NULL == stringInC)
        return;
    cout << stringInC << endl;
    env->ReleaseStringUTFChars(string, stringInC);
}

JNIEXPORT jdouble JNICALL Java_com_example_jni_JNIJava_average(JNIEnv *env, jclass jthis, jintArray
intArray) {
    jint *intArrayInC = env->GetIntArrayElements(intArray, NULL);
    if (NULL == intArrayInC)
        return -1;
    jsize length = env->GetArrayLength(intArray);
    int sum = 0;
    for (int i = 0; i < length; i++) {
        sum += intArrayInC[i];
    }
    env->ReleaseIntArrayElements(intArray, intArrayInC, 0);
    return (double) sum / length;
}
```

Output

Running the example class above yields the following output :

> Invoked C++ 'printString' from Java
> Got result from C++ 'average': 3.4

Section 98.2: Calling Java methods from C++ (callback)

Calling a Java method from native code is a two-step process :

1. obtain a method pointer with the `GetMethodID` JNI function, using the method name and descriptor ;
2. call one of the `Call*Method` functions listed here.

Java code

```java
/*** com.example.jni.JNIJavaCallback.java ***/

package com.example.jni;

public class JNIJavaCallback {
    static {
        System.loadLibrary("libJNI_CPP");
    }

    public static void main(String[] args) {
        new JNIJavaCallback().callback();
    }

    public native void callback();

    public static void printNum(int i) {
        System.out.println("Got int from C++: " + i);
    }

    public void printFloat(float i) {
        System.out.println("Got float from C++: " + i);
    }
}
```

C++ code

```cpp
// com_example_jni_JNICppCallback.cpp

#include <iostream>
#include "com_example_jni_JNIJavaCallback.h"

using namespace std;

JNIEXPORT void JNICALL Java_com_example_jni_JNIJavaCallback_callback(JNIEnv *env, jobject jthis) {
    jclass thisClass = env->GetObjectClass(jthis);

    jmethodID printFloat = env->GetMethodID(thisClass, "printFloat", "(F)V");
    if (NULL == printFloat)
        return;
    env->CallVoidMethod(jthis, printFloat, 5.221);

    jmethodID staticPrintInt = env->GetStaticMethodID(thisClass, "printNum", "(I)V");
    if (NULL == staticPrintInt)
        return;
    env->CallVoidMethod(jthis, staticPrintInt, 17);
}
```

Output

> Got float from C++: 5.221
> Got int from C++: 17

Getting the descriptor

Descriptors (or *internal type signatures*) are obtained using the **javap** program on the compiled `.class` file. Here is the output of `javap -p -s com.example.jni.JNIJavaCallback` :

```
Compiled from "JNIJavaCallback.java"
public class com.example.jni.JNIJavaCallback {
  static {};
    descriptor: ()V

  public com.example.jni.JNIJavaCallback();
    descriptor: ()V

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V

  public native void callback();
    descriptor: ()V

  public static void printNum(int);
    descriptor: (I)V  // <---- Needed

  public void printFloat(float);
    descriptor: (F)V  // <---- Needed
}
```

Section 98.3: Loading native libraries

The common idiom for loading shared library files in Java is the following :

```
public class ClassWithNativeMethods {
    static {
        System.loadLibrary("Example");
    }

    public native void someNativeMethod(String arg);
    ...
```

Calls to `System.loadLibrary` are almost always static so as to occur during class loading, ensuring that no native method can execute before the shared library has been loaded. However the following is possible :

```
public class ClassWithNativeMethods {
    // Call this before using any native method
    public static void prepareNativeMethods() {
        System.loadLibrary("Example");
    }

    ...
```

This allows to defer shared library loading until necessary, but requires extra care to avoid `java.lang.UnsatisfiedLinkError`s.

Target file lookup

Shared library files are searched for in the paths defined by the `java.library.path` system property, which can be overridden using the `-Djava.library.path=` JVM argument at runtime :

```
java -Djava.library.path=path/to/lib/:path/to/other/lib MainClassWithNativeMethods
```

Watch out for system path separators : for example, Windows uses ; instead of :.

Note that `System.loadLibrary` resolves library filenames in a platform-dependent manner : the code snippet above expects a file named `libExample.so` on Linux, and `Example.dll` on Windows.

An alternative to `System.loadLibrary` is `System.load(String)`, which takes the full path to a shared library file, circumventing the `java.library.path` lookup :

```
public class ClassWithNativeMethods {
    static {
        System.load("/path/to/lib/libExample.so");
    }

    ...
```

Chapter 99: Functional Interfaces

In Java 8+, a *functional interface* is an interface that has just one abstract method (aside from the methods of Object). See JLS §9.8. Functional Interfaces.

Section 99.1: List of standard Java Runtime Library functional interfaces by signature

Parameter Types	Return Type	Interface
()	void	Runnable
()	T	Supplier
()	boolean	BooleanSupplier
()	int	IntSupplier
()	long	LongSupplier
()	double	DoubleSupplier
(T)	void	Consumer<T>
(T)	T	UnaryOperator<T>
(T)	R	Function<T,R>
(T)	boolean	Predicate<T>
(T)	int	ToIntFunction<T>
(T)	long	ToLongFunction<T>
(T)	double	ToDoubleFunction<T>
(T, T)	T	BinaryOperator<T>
(T, U)	void	BiConsumer<T,U>
(T, U)	R	BiFunction<T,U,R>
(T, U)	boolean	BiPredicate<T,U>
(T, U)	int	ToIntBiFunction<T,U>
(T, U)	long	ToLongBiFunction<T,U>
(T, U)	double	ToDoubleBiFunction<T,U>
(T, int)	void	ObjIntConsumer<T>
(T, long)	void	ObjLongConsumer<T>
(T, double)	void	ObjDoubleConsumer<T>
(int)	void	IntConsumer
(int)	R	IntFunction<R>
(int)	boolean	IntPredicate
(int)	int	IntUnaryOperator
(int)	long	IntToLongFunction
(int)	double	IntToDoubleFunction
(int, int)	int	IntBinaryOperator
(long)	void	LongConsumer
(long)	R	LongFunction<R>
(long)	boolean	LongPredicate
(long)	int	LongToIntFunction
(long)	long	LongUnaryOperator
(long)	double	LongToDoubleFunction
(long, long)	long	LongBinaryOperator

(double)	void	DoubleConsumer
(double)	R	DoubleFunction<R>
(double)	boolean	DoublePredicate
(double)	int	DoubleToIntFunction
(double)	long	DoubleToLongFunction
(double)	double	DoubleUnaryOperator
(double, double)	double	DoubleBinaryOperator

Chapter 100: Fluent Interface

Section 100.1: Fluent programming style

In fluent programming style you return `this` from fluent (setter) methods that would return nothing in non-fluent programming style.

This allows you to chain the different method calls which makes your code shorter and easier to handle for the developers.

Consider this non-fluent code:

```
public class Person {
  private  String firstName;
  private String lastName;

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getLastName() {
    return lastName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

  public String whoAreYou() {
    return "I am " + firstName + " " + lastName;
  }

  public static void main(String[] args) {
    Person person = new Person();
    person.setFirstName("John");
    person.setLastName("Doe");
    System.out.println(person.whoAreYou());
  }
}
```

As the setter methods don't return anything, we need 4 instructions in the `main`method to instantiate a `Person` with some data and print it. With a fluent style this code can be changed to:

```
public class Person {
  private  String firstName;
  private String lastName;

  public String getFirstName() {
    return firstName;
  }

  public Person withFirstName(String firstName) {
    this.firstName = firstName;
    return this;
```

```
  }

  public String getLastName() {
    return lastName;
  }

  public Person withLastName(String lastName) {
    this.lastName = lastName;
    return this;
  }

  public String whoAreYou() {
    return "I am " + firstName + " " + lastName;
  }

  public static void main(String[] args) {
    System.out.println(new Person().withFirstName("John")
      .withLastName("Doe").whoAreYou());
  }
}
```

The idea is to always return some object to enable building of a method call chain and to use method names which reflect natural speaking. This fluent style makes the code more readable.

Section 100.2: Truth - Fluent Testing Framework

From "How to use Truth" http://google.github.io/truth/

```
String string = "awesome";
assertThat(string).startsWith("awe");
assertWithMessage("Without me, it's just aweso").that(string).contains("me");

Iterable<Color> googleColors = googleLogo.getColors();
assertThat(googleColors)
    .containsExactly(BLUE, RED, YELLOW, BLUE, GREEN, RED)
    .inOrder();
```

Chapter 101: Remote Method Invocation (RMI)

Section 101.1: Callback: invoking methods on a "client"

Overview

In this example 2 clients send information to each other through a server. One client sends the server a number which is relayed to the second client. The second client halves the number and sends it back to the first client through the server. The first client does the same. The server stops the communication when the number returned to it by any of the clients is less than 10. The return value from the server to the clients (the number it got converted to string representation) then backtracks the process.

1. A login server binds itself to a registry.
2. A client looks up the login server and calls the `login` method with its information. Then:
    - The login server stores the client information. It includes the client's stub with the callback methods.
    - The login server creates and returns a server stub ("connection" or "session") to the client to store. It includes the server's stub with its methods including a `logout` method (unused in this example).
3. A client calls the server's `passInt` with the name of the recipient client and an **`int`**.
4. The server calls the `half` on the recipient client with that **`int`**. This initiates a back-and-forth (calls and callbacks) communication until stopped by the server.

The shared remote interfaces

The login server:

```
package callbackRemote;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteLogin extends Remote {

    RemoteConnection login(String name, RemoteClient client) throws RemoteException;
}
```

The server:

```
package callbackRemote;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteConnection extends Remote {

    void logout() throws RemoteException;

    String passInt(String name, int i) throws RemoteException;
}
```

The client:

```
package callbackRemote;

import java.rmi.Remote;
```

```
import java.rmi.RemoteException;

public interface RemoteClient extends Remote {

    void half(int i) throws RemoteException;
}
```

The implementations

The login server:

```
package callbackServer;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.util.HashMap;
import java.util.Map;

import callbackRemote.RemoteClient;
import callbackRemote.RemoteConnection;
import callbackRemote.RemoteLogin;

public class LoginServer implements RemoteLogin {

    static Map<String, RemoteClient> clients = new HashMap<>();

    @Override
    public RemoteConnection login(String name, RemoteClient client) {

        Connection connection = new Connection(name, client);
        clients.put(name, client);
        System.out.println(name + " logged in");
        return connection;
    }

    public static void main(String[] args) {

        try {
            Registry reg = LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
            LoginServer server = new LoginServer();
            UnicastRemoteObject.exportObject(server, Registry.REGISTRY_PORT);
            reg.rebind("LoginServerName", server);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

The server:

```
package callbackServer;

import java.rmi.NoSuchObjectException;
import java.rmi.RemoteException;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.server.Unreferenced;

import callbackRemote.RemoteClient;
import callbackRemote.RemoteConnection;
```

```
public class Connection implements RemoteConnection, Unreferenced {

    RemoteClient client;
    String name;

    public Connection(String name, RemoteClient client) {

        this.client = client;
        this.name = name;
        try {
            UnicastRemoteObject.exportObject(this, Registry.REGISTRY_PORT);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void unreferenced() {

        try {
            UnicastRemoteObject.unexportObject(this, true);
        } catch (NoSuchObjectException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void logout() {

        try {
            UnicastRemoteObject.unexportObject(this, true);
        } catch (NoSuchObjectException e) {
            e.printStackTrace();
        }
    }

    @Override
    public String passInt(String recipient, int i) {

        System.out.println("Server received from " + name + ":" + i);
        if (i < 10)
            return String.valueOf(i);
        RemoteClient client = LoginServer.clients.get(recipient);
        try {
            client.half(i);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
        return String.valueOf(i);
    }
}
```

The client:

```
package callbackClient;

import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
```

```
import callbackRemote.RemoteClient;
import callbackRemote.RemoteConnection;
import callbackRemote.RemoteLogin;

public class Client implements RemoteClient {

    RemoteConnection connection;
    String name, target;

    Client(String name, String target) {

        this.name = name;
        this.target = target;
    }

    public static void main(String[] args) {

        Client client = new Client(args[0], args[1]);
        try {
            Registry reg = LocateRegistry.getRegistry();
            RemoteLogin login = (RemoteLogin) reg.lookup("LoginServerName");
            UnicastRemoteObject.exportObject(client, Integer.parseInt(args[2]));
            client.connection = login.login(client.name, client);
        } catch (RemoteException | NotBoundException e) {
            e.printStackTrace();
        }

        if ("Client1".equals(client.name)) {
            try {
                client.connection.passInt(client.target, 120);
            } catch (RemoteException e) {
                e.printStackTrace();
            }
        }
    }

    @Override
    public void half(int i) throws RemoteException {

        String result = connection.passInt(target, i / 2);
        System.out.println(name + " received: \"" + result + "\"");
    }
}
```

Running the example:

1. Run the login server.
2. Run a client with the arguments `Client2 Client1 1097`.
3. Run a client with the arguments `Client1 Client2 1098`.

The outputs will appear in 3 consoles since there are 3 JVMs. here they are lumped together:

> Client2 logged in
> Client1 logged in
> Server received from Client1:120
> Server received from Client2:60
> Server received from Client1:30
> Server received from Client2:15
> Server received from Client1:7

> Client1 received: "7"
> Client2 received: "15"
> Client1 received: "30"
> Client2 received: "60"

Section 101.2: Simple RMI example with Client and Server implementation

This is a simple RMI example with five Java classes and two packages, *server* and *client*.

Server Package

PersonListInterface.java

```
public interface PersonListInterface extends Remote
{
    /**
     * This interface is used by both client and server
     * @return List of Persons
     * @throws RemoteException
     */
    ArrayList<String> getPersonList() throws RemoteException;
}
```

PersonListImplementation.java

```
public class PersonListImplementation
extends UnicastRemoteObject
implements PersonListInterface
{

    private static final long serialVersionUID = 1L;

    // standard constructor needs to be available
    public PersonListImplementation() throws RemoteException
    {}

    /**
     * Implementation of "PersonListInterface"
     * @throws RemoteException
     */
    @Override
    public ArrayList<String> getPersonList() throws RemoteException
    {
        ArrayList<String> personList = new ArrayList<String>();

        personList.add("Peter Pan");
        personList.add("Pippi Langstrumpf");
        // add your name here :)

        return personList;
    }
}
```

Server.java

```
public class Server {
```

```
    /**
     * Register servicer to the known public methods
     */
    private static void createServer() {
        try {
            // Register registry with standard port 1099
            LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
            System.out.println("Server : Registry created.");

            // Register PersonList to registry
            Naming.rebind("PersonList", new PersonListImplementation());
            System.out.println("Server : PersonList registered");

        } catch (final IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(final String[] args) {
        createServer();
    }
}
```

Client package

PersonListLocal.java

```
public class PersonListLocal {
    private static PersonListLocal instance;
    private PersonListInterface personList;

    /**
     * Create a singleton instance
     */
    private PersonListLocal() {
        try {
            // Lookup to the local running server with port 1099
            final Registry registry = LocateRegistry.getRegistry("localhost",
                    Registry.REGISTRY_PORT);

            // Lookup to the registered "PersonList"
            personList = (PersonListInterface) registry.lookup("PersonList");
        } catch (final RemoteException e) {
            e.printStackTrace();
        } catch (final NotBoundException e) {
            e.printStackTrace();
        }
    }

    public static PersonListLocal getInstance() {
        if (instance == null) {
            instance = new PersonListLocal();
        }

        return instance;
    }

    /**
     * Returns the servers PersonList
     */
    public ArrayList<String> getPersonList() {
```

```
        if (instance != null) {
            try {
                return personList.getPersonList();
            } catch (final RemoteException e) {
                e.printStackTrace();
            }
        }

        return new ArrayList<>();
    }
       }
```

PersonTest.java

```
public class PersonTest
{
    public static void main(String[] args)
    {
        // get (local) PersonList
        ArrayList<String> personList = PersonListLocal.getInstance().getPersonList();

        // print all persons
        for(String person : personList)
        {
            System.out.println(person);
        }
    }
}
```

Test your application

- Start main method of Server.java. Output:

```
Server : Registry created.
Server : PersonList registered
```

- Start main method of PersonTest.java. Output:

```
Peter Pan
Pippi Langstrumpf
```

Section 101.3: Client-Server: invoking methods in one JVM from another

The shared remote interface:

```
package remote;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteServer extends Remote {

    int stringToInt(String string) throws RemoteException;
}
```

The server implementing the shared remote interface:

```
package server;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

import remote.RemoteServer;

public class Server implements RemoteServer {

    @Override
    public int stringToInt(String string) throws RemoteException {

        System.out.println("Server received: \"" + string + "\"");
        return Integer.parseInt(string);
    }

    public static void main(String[] args) {

        try {
            Registry reg = LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
            Server server = new Server();
            UnicastRemoteObject.exportObject(server, Registry.REGISTRY_PORT);
            reg.rebind("ServerName", server);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

The client invoking a method on the server (remotely):

```
package client;

import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

import remote.RemoteServer;

public class Client {

    static RemoteServer server;

    public static void main(String[] args) {

        try {
            Registry reg = LocateRegistry.getRegistry();
            server = (RemoteServer) reg.lookup("ServerName");
        } catch (RemoteException | NotBoundException e) {
            e.printStackTrace();
        }

        Client client = new Client();
        client.callServer();
    }
```

```
    void callServer() {

        try {
            int i = server.stringToInt("120");
            System.out.println("Client received: " + i);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

> Server received: "120"
> Client received: 120

Chapter 102: Iterator and Iterable

The `java.util.Iterator` is the standard Java SE interface for object that implement the Iterator design pattern. The `java.lang.Iterable` interface is for objects that can *provide* an iterator.

Section 102.1: Removing elements using an iterator

The `Iterator.remove()` method is an optional method that removes the element returned by the previous call to `Iterator.next()`. For example, the following code populates a list of strings and then removes all of the empty strings.

```
List<String> names = new ArrayList<>();
names.add("name 1");
names.add("name 2");
names.add("");
names.add("name 3");
names.add("");
System.out.println("Old Size : " + names.size());
Iterator<String> it = names.iterator();
while (it.hasNext()) {
  String el = it.next();
  if (el.equals("")) {
    it.remove();
  }
}
System.out.println("New Size : " + names.size());
```

Output :

```
Old Size : 5
New Size : 3
```

Note that is the code above is the safe way to remove elements while iterating a typical collection. If instead, you attempt to do remove elements from a collection like this:

```
for (String el: names) {
    if (el.equals("")) {
        names.remove(el); // WRONG!
    }
}
```

a typical collection (such as `ArrayList`) which provides iterators with *fail fast* iterator semantics will throw a `ConcurrentModificationException`.

The `remove()` method can only called (once) following a `next()` call. If it is called before calling `next()` or if it is called twice following a `next()` call, then the `remove()` call will throw an `IllegalStateException`.

The `remove` operation is described as an *optional* operation; i.e. not all iterators will allow it. Examples where it is not supported include iterators for immutable collections, read-only views of collections, or fixed sized collections. If `remove()` is called when the iterator does not support removal, it will throw an `UnsupportedOperationException`.

Section 102.2: Creating your own Iterable

To create your own Iterable as with any interface you just implement the abstract methods in the interface. For

`Iterable` there is only one which is called `iterator()`. But its return type `Iterator` is itself an interface with three abstract methods. You can return an iterator associated with some collection or create your own custom implementation:

```
public static class Alphabet implements Iterable<Character> {

    @Override
    public Iterator<Character> iterator() {
        return new Iterator<Character>() {
            char letter = 'a';

            @Override
            public boolean hasNext() {
                return letter <= 'z';
            }

            @Override
            public Character next() {
                return letter++;
            }

            @Override
            public void remove() {
                throw new UnsupportedOperationException("Doesn't make sense to remove a letter");
            }
        };
    }
}
```

To use:

```
public static void main(String[] args) {
    for(char c : new Alphabet()) {
        System.out.println("c = " + c);
    }
}
```

The new `Iterator` should come with a state pointing to the first item, each call to next updates its state to point to the next one. The `hasNext()` checks to see if the iterator is at the end. If the iterator were connected to a modifiable collection then the iterator's optional `remove()` method might be implemented to remove the item currently pointed to from the underlying collection.

Section 102.3: Using Iterable in for loop

Classes implementing `Iterable<>` interface can be used in **`for`** loops. This is actually only syntactic sugar for getting an iterator from the object and using it to get all elements sequentially; it makes code clearer, faster to write end less error-prone.

```
public class UsingIterable {

    public static void main(String[] args) {
        List<Integer> intList = Arrays.asList(1,2,3,4,5,6,7);

        // List extends Collection, Collection extends Iterable
        Iterable<Integer> iterable = intList;

        // foreach-like loop
        for (Integer i: iterable) {
```

```
            System.out.println(i);
        }

        // pre java 5 way of iterating loops
        for(Iterator<Integer> i = iterable.iterator(); i.hasNext(); ) {
            Integer item = i.next();
            System.out.println(item);
        }
    }
}
```

Section 102.4: Using the raw iterator

While using the foreach loop (or "extended for loop") is simple, it's sometimes beneficial to use the iterator directly. For example, if you want to output a bunch of comma-separated values, but don't want the last item to have a comma:

```
List<String> yourData = //...
Iterator<String> iterator = yourData.iterator();
while (iterator.hasNext()){
    // next() "moves" the iterator to the next entry and returns it's value.
    String entry = iterator.next();
    System.out.print(entry);
    if (iterator.hasNext()){
        // If the iterator has another element after the current one:
        System.out.print(",");
    }
}
```

This is much easier and clearer than having a `isLastEntry` variable or doing calculations with the loop index.

Chapter 103: Reflection API

Reflection is commonly used by programs which require the ability to examine or modify the runtime behavior of applications running in the JVM. Java Reflection API is used for that purpose where it makes it possible to inspect classes, interfaces, fields and methods at runtime, without knowing their names at compile time. And It also makes it possible to instantiate new objects, and to invoke methods using reflection.

Section 103.1: Dynamic Proxies

Dynamic Proxies do not really have much to do with Reflection but they are part of the API. It's basically a way to create a dynamic implementation of an interface. This could be helpful when creating mockup services.
A Dynamic Proxy is an instance of an interface that is created with a so-called invocation handler that intercepts all method calls and allows the handling of their invocation manually.

```
public class DynamicProxyTest {

    public interface MyInterface1{
        public void someMethod1();
        public int someMethod2(String s);
    }

    public interface MyInterface2{
        public void anotherMethod();
    }

    public static void main(String args[]) throws Exception {
        // the dynamic proxy class
        Class<?> proxyClass = Proxy.getProxyClass(
                ClassLoader.getSystemClassLoader(),
                new Class[] {MyInterface1.class, MyInterface2.class});
        // the dynamic proxy class constructor
        Constructor<?> proxyConstructor =
            proxyClass.getConstructor(InvocationHandler.class);

        // the invocation handler
        InvocationHandler handler = new InvocationHandler(){
            // this method is invoked for every proxy method call
            // method is the invoked method, args holds the method parameters
            // it must return the method result
            @Override
            public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                String methodName = method.getName();

                if(methodName.equals("someMethod1")){
                    System.out.println("someMethod1 was invoked!");
                    return null;
                }
                if(methodName.equals("someMethod2")){
                    System.out.println("someMethod2 was invoked!");
                    System.out.println("Parameter: " + args[0]);
                    return 42;
                }
                if(methodName.equals("anotherMethod")){
                    System.out.println("anotherMethod was invoked!");
                    return null;
                }
                System.out.println("Unkown method!");
                return null;
```

```
            }
        };

        // create the dynamic proxy instances
        MyInterface1 i1 = (MyInterface1) proxyConstructor.newInstance(handler);
        MyInterface2 i2 = (MyInterface2) proxyConstructor.newInstance(handler);

        // and invoke some methods
        i1.someMethod1();
        i1.someMethod2("stackoverflow");
        i2.anotherMethod();
    }
}
```

The result of this code is this:

```
someMethod1 was invoked!
someMethod2 was invoked!
Parameter: stackoverflow
anotherMethod was invoked!
```

Section 103.2: Introduction

Basics

The Reflection API allows one to check the class structure of the code at runtime and invoke code dynamically. This is very powerful, but it is also dangerous since the compiler is not able to statically determine whether dynamic invocations are valid.

A simple example would be to get the public constructors and methods of a given class:

```
import java.lang.reflect.Constructor;
import java.lang.reflect.Method;

// This is a object representing the String class (not an instance of String!)
Class<String> clazz = String.class;

Constructor<?>[] constructors = clazz.getConstructors(); // returns all public constructors of
String
Method[] methods = clazz.getMethods(); // returns all public methods from String and parents
```

With this information it is possible to instance the object and call different methods dynamically.

Reflection and Generic Types

Generic type information is available for:

- method parameters, using `getGenericParameterTypes()`.
- method return types, using `getGenericReturnType()`.
- **public** fields, using `getGenericType`.

The following example shows how to extract the generic type information in all three cases:

```
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.ParameterizedType;
import java.lang.reflect.Type;
import java.util.List;
```

```java
import java.util.Map;

public class GenericTest {

    public static void main(final String[] args) throws Exception {
        final Method method = GenericTest.class.getMethod("testMethod", Map.class);
        final Field field = GenericTest.class.getField("testField");

        System.out.println("Method parameter:");
        final Type parameterType = method.getGenericParameterTypes()[0];
        displayGenericType(parameterType, "\t");

        System.out.println("Method return type:");
        final Type returnType = method.getGenericReturnType();
        displayGenericType(returnType, "\t");

        System.out.println("Field type:");
        final Type fieldType = field.getGenericType();
        displayGenericType(fieldType, "\t");

    }

    private static void displayGenericType(final Type type, final String prefix) {
        System.out.println(prefix + type.getTypeName());
        if (type instanceof ParameterizedType) {
            for (final Type subtype : ((ParameterizedType) type).getActualTypeArguments()) {
                displayGenericType(subtype, prefix + "\t");
            }
        }

    }

    public Map<String, Map<Integer, List<String>>> testField;

    public List<Number> testMethod(final Map<String, Double> arg) {
        return null;
    }

}
```

This results in the following output:

```
Method parameter:
java.util.Map
java.lang.String
java.lang.Double
Method return type:
java.util.List
java.lang.Number
Field type:
java.util.Map
java.lang.String
java.util.Map
java.lang.Integer
java.util.List
java.lang.String
```

Section 103.3: Evil Java hacks with Reflection

The Reflection API could be used to change values of private and final fields even in the JDK default library. This

could be used to manipulate the behaviour of some well known classes as we will see.

What is not possible

Lets start first with the only limitation means the only field we can't change with Reflection. That is the Java `SecurityManager`. It is declared in java.lang.System as

```
private static volatile SecurityManager security = null;
```

But it won't be listed in the System class if we run this code

```
for(Field f : System.class.getDeclaredFields())
    System.out.println(f);
```

Thats because of the `fieldFilterMap` in `sun.reflect.Reflection` that holds the map itself and the security field in the `System.class` and protects them against any access with Reflection. So we could not deactivate the `SecurityManager`.

Crazy Strings

Each Java String is represented by the JVM as an instance of the `String` class. However, in some situations the JVM saves heap space by using the same instance for Strings that are. This happens for string literals, and also for strings that have been "interned" by calling `String.intern()`. So if you have `"hello"` in your code multiple times it is always the same object instance.

Strings are supposed to be immutable, but it is possible to use "evil" reflection to change them. The example below show how we can change the characters in a String by replacing its `value` field.

```
public class CrazyStrings {
    static {
        try {
            Field f = String.class.getDeclaredField("value");
            f.setAccessible(true);
            f.set("hello", "you stink!".toCharArray());
        } catch (Exception e) {
        }
    }
    public static void main(String args[])  {
        System.out.println("hello");
    }
}
```

So this code will print "you stink!"

1 = 42

The same idea could be used with the Integer Class

```
public class CrazyMath {
    static {
        try {
            Field value = Integer.class.getDeclaredField("value");
            value.setAccessible(true);
            value.setInt(Integer.valueOf(1), 42);
        } catch (Exception e) {
        }
    }
```

```
    public static void main(String args[])  {
        System.out.println(Integer.valueOf(1));
    }
}
```

Everything is true

And according to this stackoverflow post we can use reflection to do something really evil.

```
public class Evil {
    static {
        try {
            Field field = Boolean.class.getField("FALSE");
            field.setAccessible(true);
            Field modifiersField = Field.class.getDeclaredField("modifiers");
            modifiersField.setAccessible(true);
            modifiersField.setInt(field, field.getModifiers() & ~Modifier.FINAL);
            field.set(null, true);
        } catch (Exception e) {
        }
    }
    public static void main(String args[]){
        System.out.format("Everything is %s", false);
    }
}
```

Note that what we are doing here is going to cause the JVM to behave in inexplicable ways. This is very dangerous.

Section 103.4: Misuse of Reflection API to change private and final variables

Reflection is useful when it is properly used for right purpose. By using reflection, you can access private variables and re-initialize final variables.

Below is the code snippet, which is **not** recommended.

```
import java.lang.reflect.*;

public class ReflectionDemo{
    public static void main(String args[]){
        try{
            Field[] fields = A.class.getDeclaredFields();
            A a = new A();
            for ( Field field:fields ) {
                if(field.getName().equalsIgnoreCase("name")){
                    field.setAccessible(true);
                    field.set(a, "StackOverFlow");
                    System.out.println("A.name="+field.get(a));
                }
                if(field.getName().equalsIgnoreCase("age")){
                    field.set(a, 20);
                    System.out.println("A.age="+field.get(a));
                }
                if(field.getName().equalsIgnoreCase("rep")){
                    field.setAccessible(true);
                    field.set(a,"New Reputation");
                    System.out.println("A.rep="+field.get(a));
                }
                if(field.getName().equalsIgnoreCase("count")){
```

```
                    field.set(a,25);
                    System.out.println("A.count="+field.get(a));
                }
            }
        }catch(Exception err){
            err.printStackTrace();
        }
    }
}

class A {
    private String name;
    public int age;
    public final String rep;
    public static int count=0;

    public A(){
        name = "Unset";
        age = 0;
        rep = "Reputation";
        count++;
    }
}
```

Output:

```
A.name=StackOverFlow
A.age=20
A.rep=New Reputation
A.count=25
```

Explanation:

In normal scenario, **`private`** variables can't be accessed outside of declared class (without getter and setter methods). **`final`** variables can't be re-assigned after initialization.

`Reflection` breaks both barriers can be abused to change both private and final variables as explained above.

`field.setAccessible(true)` is the key to achieve desired functionality.

Section 103.5: Getting and Setting fields

Using the Reflection API, it is possible to change or get the value of a field at runtime. For example, you could use it in an API to retrieve different fields based on a factor, like the OS. You can also remove modifiers like **`final`** to allow modifing fields that are final.

To do so, you will need to use the method Class#getField() in a way such as the one shown below:

```
// Get the field in class SomeClass "NAME".
Field nameField = SomeClass.class.getDeclaredField("NAME");

// Get the field in class Field "modifiers". Note that it does not
// need to be static
Field modifiersField = Field.class.getDeclaredField("modifiers");

// Allow access from anyone even if it's declared private
modifiersField.setAccessible(true);
```

```
// Get the modifiers on the "NAME" field as an int.
int existingModifiersOnNameField = nameField.getModifiers();

// Bitwise AND NOT Modifier.FINAL (16) on the existing modifiers
// Readup here https://en.wikipedia.org/wiki/Bitwise_operations_in_C
// if you're unsure what bitwise operations are.
int newModifiersOnNameField = existingModifiersOnNameField & ~Modifier.FINAL;

// Set the value of the modifiers field under an object for non-static fields
modifiersField.setInt(nameField, newModifiersOnNameField);

// Set it to be accessible. This overrides normal Java
// private/protected/package/etc access control checks.
nameField.setAccessible(true);

 // Set the value of "NAME" here. Note the null argument.
 // Pass null when modifying static fields, as there is no instance object
nameField.set(null, "Hacked by reflection...");

// Here I can directly access it. If needed, use reflection to get it. (Below)
System.out.println(SomeClass.NAME);
```

Getting fields is much easier. We can use Field#get() and its variants to get its value:

```
// Get the field in class SomeClass "NAME".
Field nameField = SomeClass.class.getDeclaredField("NAME");

// Set accessible for private fields
nameField.setAccessible(true);

// Pass null as there is no instance, remember?
String name = (String) nameField.get(null);
```

Do note this:

When using Class#getDeclaredField, use it to get a field in the class itself:

```
class HackMe extends Hacked {
    public String iAmDeclared;
}

class Hacked {
    public String someState;
}
```

Here, `HackMe#iAmDeclared` is declared field. However, `HackMe#someState` is not a declared field as it is inherited from its superclass, Hacked.

Section 103.6: Call constructor

Getting the Constructor Object

You can obtain `Constructor` class from the **`Class`** object like this:

```
Class myClass = ... // get a class object
Constructor[] constructors = myClass.getConstructors();
```

Where the `constructors` variable will have one `Constructor` instance for each public constructor declared in the

class.

If you know the precise parameter types of the constructor you want to access, you can filter the specific constructor. The next example returns the public constructor of the given class which takes a `Integer` as parameter:

```
Class myClass = ... // get a class object
Constructor constructor = myClass.getConstructor(new Class[]{Integer.class});
```

If no constructor matches the given constructor arguments a `NoSuchMethodException` is thrown.

New Instance using Constructor Object

```
Class myClass = MyObj.class // get a class object
Constructor constructor = myClass.getConstructor(Integer.class);
MyObj myObj = (MyObj) constructor.newInstance(Integer.valueOf(123));
```

Section 103.7: Call constructor of nested class

If you want to create an instance of an inner nested class you need to provide a class object of the enclosing class as an extra parameter with Class#getDeclaredConstructor.

```
public class Enclosing{
    public class Nested{
    public Nested(String a){
            System.out.println("Constructor :String => "+a);
        }
    }
    public static void main(String args[]) throws Exception {
        Class<?> clazzEnclosing = Class.forName("Enclosing");
        Class<?> clazzNested = Class.forName("Enclosing$Nested");
        Enclosing objEnclosing = (Enclosing)clazzEnclosing.newInstance();
        Constructor<?> constructor = clazzNested.getDeclaredConstructor(new
Class[]{Enclosing.class, String.class});
        Nested objInner = (Nested)constructor.newInstance(new Object[]{objEnclosing,
"StackOverFlow"});
    }
}
```

If the nested class is static you will not need this enclosing instance.

Section 103.8: Invoking a method

Using reflection, a method of an object can be invoked during runtime.

The example shows how to invoke the methods of a `String` object.

```
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

String s = "Hello World!";

// method without parameters
// invoke s.length()
Method method1 = String.class.getMethod("length");
int length = (int) method1.invoke(s); // variable length contains "12"

// method with parameters
```

```
// invoke s.substring(6)
Method method2 = String.class.getMethod("substring", int.class);
String substring = (String) method2.invoke(s, 6); // variable substring contains "World!"
```

Section 103.9: Get Class given its (fully qualified) name

Given a `String` containing the name of a class, it's **`Class`** object can be accessed using **`Class`**`.forName`:

```
Class clazz = null;
try {
    clazz = Class.forName("java.lang.Integer");
} catch (ClassNotFoundException ex) {
    throw new IllegalStateException(ex);
}
```

Version ≥ Java SE 1.2

It can be specified, if the class should be initialized (second parameter of `forName`) and which `ClassLoader` should be used (third parameter):

```
ClassLoader classLoader = ...
boolean initialize = ...
Class clazz = null;
try {
    clazz = Class.forName("java.lang.Integer", initialize, classLoader);
} catch (ClassNotFoundException ex) {
    throw new IllegalStateException(ex);
}
```

Section 103.10: Getting the Constants of an Enumeration

Giving this enumeration as Example:

```
enum Compass {
    NORTH(0),
    EAST(90),
    SOUTH(180),
    WEST(270);
    private int degree;
    Compass(int deg){
        degree = deg;
    }
    public int getDegree(){
        return degree;
    }
}
```

In Java an enum class is like any other class but has some definied constants for the enum values. Additionally it has a field that is an array that holds all the values and two static methods with name `values()` and `valueOf(String)`. We can see this if we use Reflection to print all fields in this class

```
for(Field f : Compass.class.getDeclaredFields())
    System.out.println(f.getName());
```

the output will be:

> NORTH

> EAST
> SOUTH
> WEST
> degree
> ENUM$VALUES

So we could examine enum classes with Reflection like any other class. But the Reflection API offers three enum-specific methods.

enum check

```
Compass.class.isEnum();
```

Returns true for classes that represents an enum type.

retrieving values

```
Object[] values = Compass.class.getEnumConstants();
```

Returns an array of all enum values like Compass.values() but without the need of an instance.

enum constant check

```
for(Field f : Compass.class.getDeclaredFields()){
    if(f.isEnumConstant())
        System.out.println(f.getName());
}
```

Lists all the class fields that are enum values.

Section 103.11: Call overloaded constructors using reflection

Example: Invoke different constructors by passing relevant parameters

```
import java.lang.reflect.*;

class NewInstanceWithReflection{
    public NewInstanceWithReflection(){
        System.out.println("Default constructor");
    }
    public NewInstanceWithReflection( String a){
        System.out.println("Constructor :String => "+a);
    }
    public static void main(String args[]) throws Exception {

        NewInstanceWithReflection object =
(NewInstanceWithReflection)Class.forName("NewInstanceWithReflection").newInstance();
        Constructor constructor = NewInstanceWithReflection.class.getDeclaredConstructor( new
Class[] {String.class});
        NewInstanceWithReflection object1 = (NewInstanceWithReflection)constructor.newInstance(new
Object[]{"StackOverFlow"});

    }
}
```

output:

```
Default constructor
Constructor :String => StackOverFlow
```

Explanation:

1. Create instance of class using **`Class`**`.forName` : It calls default constructor
2. Invoke `getDeclaredConstructor` of the class by passing type of parameters as **`Class`** `array`
3. After getting the constructor, create `newInstance` by passing parameter value as `Object` `array`

Chapter 104: ByteBuffer

The `ByteBuffer` class was introduced in java 1.4 to ease working on binary data. It's especially suited to use with primitive type data. It allows the creation, but also subsequent manipulation of a `byte[]`s on a higher abstraction level

Section 104.1: Basic Usage - Using DirectByteBuffer

`DirectByteBuffer` is special implementation of `ByteBuffer` that has no `byte[]` laying underneath.

We can allocate such ByteBuffer by calling:

```
ByteBuffer directBuffer = ByteBuffer.allocateDirect(16);
```

This operation will allocate 16 bytes of memory. The contents of direct buffers *may* reside outside of the normal garbage-collected heap.

We can verify whether ByteBuffer is direct by calling:

```
directBuffer.isDirect(); // true
```

The main characteristics of `DirectByteBuffer` is that JVM will try to natively work on allocated memory without any additional buffering so operations performed on it may be faster then those performed on ByteBuffers with arrays lying underneath.

It is recomended to use `DirectByteBuffer` with heavy IO operations that rely on speed of execution, like real time communication.

We have to be aware that if we try using `array()` method we will get `UnsupportedOperationException`. So it is a good practice to chech whether our ByteBuffer has it (byte array) before we try to access it:

```
byte[] arrayOfBytes;
if(buffer.hasArray()) {
    arrayOfBytes = buffer.array();
}
```

Another use of direct byte buffer is interop through JNI. Since a direct byte buffer does not use a `byte[]`, but an actual block of memory, it is possible to access that memory directly through a pointer in native code. This can save a bit of trouble and overhead on marshalling between the Java and native representation of data.

The JNI interface defines several functions to handle direct byte buffers: NIO Support.

Section 104.2: Basic Usage - Creating a ByteBuffer

There's two ways to create a `ByteBuffer`, where one can be subdivided again.

If you have an already existing `byte[]`, you can *"wrap"* it into a `ByteBuffer` to simplify processing:

```
byte[] reqBuffer = new byte[BUFFER_SIZE];
int readBytes = socketInputStream.read(reqBuffer);
final ByteBuffer reqBufferWrapper = ByteBuffer.wrap(reqBuffer);
```

This would be a possibility for code that handles low-level networking interactions

If you do not have an already existing `byte[]`, you can create a `ByteBuffer` over an array that's specifically allocated for the buffer like this:

```
final ByteBuffer respBuffer = ByteBuffer.allocate(RESPONSE_BUFFER_SIZE);
putResponseData(respBuffer);
socketOutputStream.write(respBuffer.array());
```

If the code-path is extremely performance critical and you need **direct system memory access**, the `ByteBuffer` can even allocate *direct* buffers using `#allocateDirect()`

Section 104.3: Basic Usage - Write Data to the Buffer

Given a `ByteBuffer` instance one can write primitive-type data to it using *relative* and *absolute* `put`. The striking difference is that putting data using the *relative* method keeps track of the index the data is inserted at for you, while the absolute method always requires giving an index to `put` the data at.

Both methods allow *"chaining"* calls. Given a sufficiently sized buffer one can accordingly do the following:

```
buffer.putInt(0xCAFEBABE).putChar('c').putFloat(0.25).putLong(0xDEADBEEFCAFEBABE);
```

which is equivalent to:

```
buffer.putInt(0xCAFEBABE);
buffer.putChar('c');
buffer.putFloat(0.25);
buffer.putLong(0xDEADBEEFCAFEBABE);
```

Do note that the method operating on **`byte`**s is not named specially. Additionally note that it's also valid to pass both a `ByteBuffer` and a **`byte`**`[]` to `put`. Other than that, all primitive types have specialized `put`-methods.

An additional note: The index given when using absolute `put*` is always counted in **`byte`**s.

Chapter 105: Applets

Applets have been part of Java since its official release and have been used to teach Java and programming for a number of years.

Recent years have seen an active push to move away from Applets and other browser plugins, with some browsers blocking them or actively not supporting them.

In 2016, Oracle announced their plans to deprecate the plugin, Moving to a Plugin-Free Web

Newer and better APIs are now available

Section 105.1: Minimal Applet

A very simple applet draws a rectangle and prints a string something on the screen.

```
public class MyApplet extends JApplet{

    private String str = "StackOverflow";

    @Override
    public void init() {
        setBackground(Color.gray);
    }
    @Override
    public void destroy() {}
    @Override
    public void start() {}
    @Override
    public void stop() {}
    @Override
    public void paint(Graphics g) {
        g.setColor(Color.yellow);
        g.fillRect(1,1,300,150);
        g.setColor(Color.red);
        g.setFont(new Font("TimesRoman", Font.PLAIN, 48));
        g.drawString(str, 10, 80);
    }
}
```

The main class of an applet extends from `javax.swing.JApplet`.

Version ≤ Java SE 1.2

Before Java 1.2 and the introduction of the swing API applets had extended from `java.applet.Applet`.

Applets don't require a main method. The entry point is controlled by the life cycle. To use them, they need to be embedded in a HTML document. This is also the point where their size is defined.

```
<html>
  <head></head>
  <body>
     <applet code="MyApplet.class" width="400" height="200"></applet>
  </body>
</html>
```

Section 105.2: Creating a GUI

Applets could easily be used to create a GUI. They act like a `Container` and have an `add()` method that takes any `awt` or `swing` component.

```
public class MyGUIApplet extends JApplet{

    private JPanel panel;
    private JButton button;
    private JComboBox<String> cmbBox;
    private JTextField textField;

    @Override
    public void init(){
        panel = new JPanel();
        button = new JButton("ClickMe!");
        button.addActionListener(new ActionListener(){
            @Override
            public void actionPerformed(ActionEvent ae) {
                if(((String)cmbBox.getSelectedItem()).equals("greet")) {
                    JOptionPane.showMessageDialog(null,"Hello " + textField.getText());
                } else {
                    JOptionPane.showMessageDialog(null,textField.getText() + " stinks!");
                }
            }
        });
        cmbBox = new JComboBox<>(new String[]{"greet", "offend"});
        textField = new JTextField("John Doe");
        panel.add(cmbBox);
        panel.add(textField);
        panel.add(button);
        add(panel);
    }
}
```

Section 105.3: Open links from within the applet

You can use the method `getAppletContext()` to get an `AppletContext` object that allows you to request the browser to open a link. For this you use the method `showDocument()`. Its second parameter tells the browser to use a new window `_blank` or the one that shows the applet `_self`.

```
public class MyLinkApplet extends JApplet{
    @Override
    public void init(){
        JButton button = new JButton("ClickMe!");
        button.addActionListener(new ActionListener(){
            @Override
            public void actionPerformed(ActionEvent ae) {
                AppletContext a = getAppletContext();
                try {
                    URL url = new URL("http://stackoverflow.com/");
                    a.showDocument(url,"_blank");
                } catch (Exception e) { /* omitted for brevity */ }
            }
        });
        add(button);
    }
}
```

Section 105.4: Loading images, audio and other resources

Java applets are able to load different resources. But since they are running in the web browser of the client you need to make sure that these resources are accessible. Applets are not able to access client resources as the local file system.

If you want to load resources from the same URL the Applet is stored you can use the method `getCodeBase()` to retrieve the base URL. To load resources, applets offer the methods `getImage()` and `getAudioClip()` to load images or audio files.

Load and show an image

```
public class MyImgApplet extends JApplet{

    private Image img;

    @Override
    public void init(){
        try {
            img = getImage(new URL("http://cdn.sstatic.net/stackexchange/img/logos/so/so-
logo.png"));
        } catch (MalformedURLException e) { /* omitted for brevity */ }
    }
    @Override
        public void paint(Graphics g) {
            g.drawImage(img, 0, 0, this);
        }
}
```

Load and play an audio file

```
public class MyAudioApplet extends JApplet{

    private AudioClip audioClip;

    @Override
    public void init(){
        try {
            audioClip = getAudioClip(new URL("URL/TO/AN/AUDIO/FILE.WAV"));
        } catch (MalformedURLException e) { /* omitted for brevity */ }
    }
    @Override
    public void start() {
        audioClip.play();
    }
    @Override
    public void stop(){
        audioClip.stop();
    }
}
```

Load and display a text file

```
public class MyTextApplet extends JApplet{
    @Override
    public void init(){
        JTextArea textArea = new JTextArea();
        JScrollPane sp = new JScrollPane(textArea);
        add(sp);
```

```
        // load text
        try {
            URL url = new URL("http://www.textfiles.com/fun/quotes.txt");
            InputStream in = url.openStream();
            BufferedReader bf = new BufferedReader(new InputStreamReader(in));
            String line = "";
            while((line = bf.readLine()) != null) {
                textArea.append(line + "\n");
            }
        } catch(Exception e) { /* omitted for brevity */ }
    }
}
```

Chapter 106: Expressions

Expressions in Java are the primary construct for doing calculations.

Section 106.1: Operator Precedence

When an expression contains multiple operators, it can potentially be read in different ways. For example, the mathematical expression `1 + 2 x 3` could be read in two ways:

1. Add `1` and `2` and multiply the result by `3`. This gives the answer `9`. If we added parentheses, this would look like `( 1 + 2 ) x 3`.
2. Add `1` to the result of multiplying `2` and `3`. This gives the answer `7`. If we added parentheses, this would look like `1 + ( 2 x 3 )`.

In mathematics, the convention is to read the expression the second way. The general rule is that multiplication and division are done before addition and subtraction. When more advanced mathematical notation is used, either the meaning is either "self-evident" (to a trained mathematician!), or parentheses are added to disambiguate. In either case, the effectiveness of the notation to convey meaning depends on the intelligence and shared knowledge of the mathematicians.

Java has the same clear rules on how to read an expression, based on the *precedence* of the operators that are used.

In general, each operator is ascribed a *precedence* value; see the table below.

For example:

```
1 + 2 * 3
```

The precedence of `+` is lower than the precedence of `*`, so the result of the expression is 7, not 9.

Description	Operators / constructs (primary)	Precedence	Associativity
Qualifier Parentheses Instance creation Field access Array access Method invocation Method reference	name.name (expr) **new** primary.name primary[expr] primary(expr, ...) primary::name	15	Left to right
Post increment	expr++, expr--	14	-
Pre increment Unary Cast1	++expr, --expr, +expr, -expr, ~expr, !expr, (type)expr	13	- Right to left Right to left
Multiplicative	* / %	12	Left to right
Additive	+ -	11	Left to right
Shift	<< >> >>>	10	Left to right
Relational	< > <= >= **instanceof**	9	Left to right
Equality	== !=	8	Left to right
Bitwise AND	&	7	Left to right
Bitwise exclusive OR	^	6	Left to right
Bitwise inclusive OR	|	5	Left to right
Logical AND	&&	4	Left to right

Logical OR	||	3	Left to right
Conditional1	? :	2	Right to left
Assignment Lambda1	= *= /= %= += -= <<= >>= >>>= &= ^= |= ->	1	Right to left

1 Lambda expression precedence is complex, as it can also occur after a cast, or as the third part of the conditional ternary operator.

Section 106.2: Expression Basics

Expressions in Java are the primary construct for doing calculations. Here are some examples:

```
1                // A simple literal is an expression
1 + 2            // A simple expression that adds two numbers
(i + j) / k      // An expression with multiple operations
(flag) ? c : d   // An expression using the "conditional" operator
(String) s       // A type-cast is an expression
obj.test()       // A method call is an expression
new Object()     // Creation of an object is an expression
new int[]        // Creation of an object is an expression
```

In general, an expression consists of the following forms:

- Expression names which consist of:
  - Simple identifiers; e.g. `someIdentifier`
  - Qualified identifiers; e.g. `MyClass.someField`
- Primaries which consist of:
  - Literals; e.g. `1`, `1.0`, `'X'`, `"hello"`, **`false`** and **`null`**
  - Class literal expressions; e.g. `MyClass.`**`class`**
  - **`this`** and `<TypeName> . this`
  - Parenthesized expressions; e.g. `( a + b )`
  - Class instance creation expressions; e.g. **`new`** `MyClass(1, 2, 3)`
  - Array instance creation expressions; e.g. **`new`** **`int`**`[3]`
  - Field access expressions; e.g. `obj.someField` or **`this`**`.someField`
  - Array access expressions; e.g. `vector[21]`
  - Method invocations; e.g. `obj.doIt(1, 2, 3)`
  - Method references (Java 8 and later); e.g. `MyClass::doIt`
- Unary operator expressions; e.g. `!a` or `i++`
- Binary operator expressions; e.g. `a + b` or `obj ==` **`null`**
- Ternary operator expressions; e.g. `(obj ==` **`null`**`) ? 1 : obj.getCount()`
- Lambda expressions (Java 8 and later); e.g. `obj -> obj.getCount()`

The details of the different forms of expressions may be found in other Topics.

- The Operators topic covers unary, binary and ternary operator expressions.
- The Lambda expressions topic covers lambda expressions and method reference expressions.
- The Classes and Objects topic covers class instance creation expressions.
- The Arrays topic covers array access expressions and array instance creation expressions.
- The Literals topic covers the different kinds of literals expressions.

The Type of an Expression

In most cases, an expression has a static type that can be determined at compile time by examining and its subexpressions. These are referred to as *stand-alone* expressions.

However, (in Java 8 and later) the following kinds of expressions may be *poly expressions*:

- Parenthesized expressions
- Class instance creation expressions
- Method invocation expressions
- Method reference expressions
- Conditional expressions
- Lambda expressions

When an expression is a poly expression, its type may be influenced by the expression's *target type*; i.e. what it is being used for.

The value of an Expression

The value of an expression is assignment compatible with its type. The exception to this is when *heap pollution* has occurred; e.g. because "unsafe conversion" warnings have been (inappropriately) suppressed or ignored.

Expression Statements

Unlike many other languages, Java does not generally allow expressions to be used as statements. For example:

```
public void compute(int i, int j) {
    i + j;   // ERROR
}
```

Since the result of evaluating an expression like cannot be use, and since it cannot affect the execution of the program in any other way, the Java designers took the position that such usage is either a mistake, or misguided.

However, this does not apply to all expressions. A subset of expressions are (in fact) legal as statements. The set comprises:

- Assignment expression, including *operation-and-becomes* assignments.
- Pre and post increment and decrement expressions.
- Method calls (`void` or non-`void`).
- Class instance creation expressions.

Section 106.3: Expression evaluation order

Java expressions are evaluated following the following rules:

- Operands are evaluated from left to right.
- The operands of an operator are evaluated before the operator.
- Operators are evaluated according to operator precedence
- Argument lists are evaluated from left to right.

Simple Example

In the following example:

```
int i = method1() + method2();
```

the order of evaluation is:

1. The left operand of `=` operator is evaluated to the address of `i`.
2. The left operand of the `+` operator (`method1()`) is evaluated.

3. The right operand of the `+` operator (`method2()`) is evaluated.
4. The `+` operation is evaluated.
5. The `=` operation is evaluated, assigning the result of the addition to `i`.

Note that if the effects of the calls are observable, you will be able to observe that the call to `method1` occurs before the call to `method2`.

Example with an operator which has a side-effect

In the following example:

```
int i = 1;
intArray[i] = ++i + 1;
```

the order of evaluation is:

1. The left operand of `=` operator is evaluated. This gives the address of `intArray[1]`.
2. The pre-increment is evaluated. This adds `1` to `i`, and evaluates to `2`.
3. The right hand operand of the `+` is evaluated.
4. The `+` operation is evaluated to: `2 + 1` -> `3`.
5. The `=` operation is evaluated, assigning `3` to `intArray[1]`.

Note that since the left-hand operand of the `=` is evaluated first, it is not influenced by the side-effect of the `++i` subexpression.

Reference:

- JLS 15.7 - Evaluation Order

Section 106.4: Constant Expressions

A constant expression is an expression that yields a primitive type or a String, and whose value can be evaluated at compile time to a literal. The expression must evaluate without throwing an exception, and it must be composed of only the following:

- Primitive and String literals.
- Type casts to primitive types or `String`.
- The following unary operators: `+`, `-`, `~` and `!`.
- The following binary operators: `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `>>>`, `<`, `<=`, `>`, `>=`, `==`, `!=`, `&`, `^`, `|`, `&&` and `||`.
- The ternary conditional operator `? :`.
- Parenthesized constant expressions.
- Simple names that refer to constant variables. (A constant variable is a variable declared as **`final`** where the initializer expression is itself a constant expression.)
- Qualified names of the form **`<TypeName>`** `.` **`<Identifier>`** that refer to constant variables.

Note that the above list *excludes* `++` and `--`, the assignment operators, **`class`** and **`instanceof`**, method calls and references to general variables or fields.

Constant expressions of type `String` result in an "interned" `String`, and floating point operations in constant expressions are evaluated with FP-strict semantics.

Uses for Constant Expressions

Constant expressions can be used (just about) anywhere that a normal expression can be used. However, they have a special significance in the following contexts.

Constant expressions are required for case labels in switch statements. For example:

```
switch (someValue) {
case 1 + 1:             // OK
case Math.min(2, 3):    // Error - not a constant expression
    doSomething();
}
```

When the expression on the right hand side of an assignment is a constant expression, then the assignment can perform a primitive narrowing conversion. This is allowed provided that the value of the constant expression is within the range of the type on the left hand side. (See JLS 5.1.3 and 5.2) For example:

```
byte b1 = 1 + 1;              // OK - primitive narrowing conversion.
byte b2 = 127 + 1;            // Error - out of range
byte b3 = b1 + 1;             // Error - not a constant expession
byte b4 = (byte) (b1 + 1);    // OK
```

When a constant expression is used as the condition in a `do`, **`while`** or **`for`**, then it affects the readability analysis. For example:

```
while (false) {
    doSomething();            // Error - statenent not reachable
}
boolean flag = false;
while (flag) {
    doSomething();            // OK
}
```

(Note that this does not apply `if` statements. The Java compiler allows the `then` or **`else`** block of an `if` statement to be unreachable. This is the Java analog of conditional compilation in C and C++.)

Finally, **`static final`** fields in an class or interface with constant expression initializers are initialized eagerly. Thus, it is guaranteed that these constants will be observed in the initialized state, even when there is a cycle in the class initialization dependency graph.

For more information, refer to JLS 15.28. Constant Expressions.

Chapter 107: JSON in Java

JSON (JavaScript Object Notation) is a lightweight, text-based, language-independent data exchange format that is easy for humans and machines to read and write. JSON can represent two structured types: objects and arrays. JSON is often used in Ajax applications, configurations, databases, and RESTful web services. The Java API for JSON Processing provides portable APIs to parse, generate, transform, and query JSON.

Section 107.1: Using Jackson Object Mapper

Pojo Model

```
public class Model {
    private String firstName;
    private String lastName;
    private int age;
    /* Getters and setters not shown for brevity */
}
```

Example: String to Object

```
Model outputObject = objectMapper.readValue(
     "{\"firstName\":\"John\",\"lastName\":\"Doe\",\"age\":23}",
     Model.class);
System.out.println(outputObject.getFirstName());
//result: John
```

Example: Object to String

```
String jsonString = objectMapper.writeValueAsString(inputObject));
//result: {"firstName":"John","lastName":"Doe","age":23}
```

Details

Import statement needed:

```
import com.fasterxml.jackson.databind.ObjectMapper;
```

Maven dependency: jackson-databind

`ObjectMapper` instance

```
//creating one
ObjectMapper objectMapper = new ObjectMapper();
```

- `ObjectMapper` is threadsafe
- recommended: have a shared, static instance

Deserialization:

```
<T> T readValue(String content, Class<T> valueType)
```

- `valueType` needs to be specified -- the return will be of this type
- Throws
    - `IOException` - in case of a low-level I/O problem
    - `JsonParseException` - if underlying input contains invalid content
    - `JsonMappingException` - if the input JSON structure does not match object structure

Usage example (jsonString is the input string):

```
Model fromJson = objectMapper.readValue(jsonString, Model.class);
```

Method for serialization:

String writeValueAsString(Object value)

- Throws
  - `JsonProcessingException` in case of an error
  - Note: prior to version 2.1, throws clause included IOException; 2.1 removed it.

Section 107.2: JSON To Object (Gson Library)

Lets assume you have a class called `Person` with just `name`

```
private class Person {
    public String name;

    public Person(String name) {
        this.name = name;
    }
}
```

Code:

```
Gson gson = new Gson();
String json = "{\"name\": \"John\"}";

Person person = gson.fromJson(json, Person.class);
System.out.println(person.name); //John
```

You must have gson library in your classpath.

Section 107.3: JSONObject.NULL

If you need to add a property with a **`null`** value, you should use the predefined static final `JSONObject.NULL` and not the standard Java **`null`** reference.

`JSONObject.NULL` is a sentinel value used to explicitly define a property with an empty value.

```
JSONObject obj = new JSONObject();
obj.put("some", JSONObject.NULL);   //Creates: {"some":null}
System.out.println(obj.get("some"));//prints: null
```

Note

```
JSONObject.NULL.equals(null); //returns true
```

Which is a **clear violation** of `Java.equals()` contract:

> For any non-null reference value x, x.equals(null) should return false

Section 107.4: JSON Builder - chaining methods

You can use method chaining while working with `JSONObject` and `JSONArray`.

JSONObject example

```
JSONObject obj = new JSONObject();//Initialize an empty JSON object
//Before: {}
obj.put("name","Nikita").put("age","30").put("isMarried","true");
//After: {"name":"Nikita","age":30,"isMarried":true}
```

JSONArray

```
JSONArray arr = new JSONArray();//Initialize an empty array
//Before: []
arr.put("Stack").put("Over").put("Flow");
//After: ["Stack","Over","Flow"]
```

Section 107.5: Object To JSON (Gson Library)

Lets assume you have a class called `Person` with just `name`

```
private class Person {
    public String name;

    public Person(String name) {
        this.name = name;
    }
}
```

Code:

```
Gson g = new Gson();

Person person = new Person("John");
System.out.println(g.toJson(person)); // {"name":"John"}
```

Of course the Gson jar must be on the classpath.

Section 107.6: JSON Iteration

Iterate over `JSONObject` properties

```
JSONObject obj = new JSONObject("{\"isMarried\":\"true\", \"name\":\"Nikita\", \"age\":\"30\"}");
Iterator<String> keys = obj.keys();//all keys: isMarried, name & age
while (keys.hasNext()) {                    //as long as there is another key
      String key = keys.next();             //get next key
      Object value = obj.get(key);          //get next value by key
      System.out.println(key + " : " + value);//print key : value
}
```

Iterate over `JSONArray` values

```
JSONArray arr = new JSONArray();        //Initialize an empty array
//push (append) some values in:
arr.put("Stack");
```

```
arr.put("Over");
arr.put("Flow");
for (int i = 0; i < arr.length(); i++) {//iterate over all values
    Object value = arr.get(i);          //get value
    System.out.println(value);          //print each value
}
```

Section 107.7: optXXX vs getXXX methods

`JSONObject` and `JSONArray` have a few methods that are very useful while dealing with a possibility that a value your are trying to get does not exist or is of another type.

```
JSONObject obj = new JSONObject();
obj.putString("foo", "bar");

// For existing properties of the correct type, there is no difference
obj.getString("foo");        // returns "bar"
obj.optString("foo");        // returns "bar"
obj.optString("foo", "tux"); // returns "bar"

// However, if a value cannot be coerced to the required type, the behavior differs
obj.getInt("foo");      // throws JSONException
obj.optInt("foo");      // returns 0
obj.optInt("foo", 123); // returns 123

// Same if a property does not exist
obj.getString("undefined");        // throws JSONException
obj.optString("undefined");        // returns ""
obj.optString("undefined", "tux"); // returns "tux"
```

The same rules apply to the `getXXX` / `optXXX` methods of `JSONArray`.

Section 107.8: Extract single element from JSON

```
String json = "{\"name\": \"John\", \"age\":21}";

JsonObject jsonObject = new JsonParser().parse(json).getAsJsonObject();

System.out.println(jsonObject.get("name").getAsString()); //John
System.out.println(jsonObject.get("age").getAsInt()); //21
```

Section 107.9: JsonArray to Java List (Gson Library)

Here is a simple JsonArray which you would like to convert to a Java `ArrayList`:

```
{
    "list": [
                "Test_String_1",
                "Test_String_2"
            ]
}
```

Now pass the `JsonArray` 'list' to the following method which returns a corresponding Java `ArrayList`:

```
public ArrayList<String> getListString(String jsonList){
    Type listType = new TypeToken<List<String>>() {}.getType();
    //make sure the name 'list' matches the name of 'JsonArray' in your 'Json'.
    ArrayList<String> list = new Gson().fromJson(jsonList, listType);
```

```
    return list;
}
```

You should add the following maven dependency to your `POM.xml` file:

```
<!-- https://mvnrepository.com/artifact/com.google.code.gson/gson -->
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.7</version>
</dependency>
```

Or you should have the jar `com.google.code.gson:gson:jar:<version>` in your classpath.

Section 107.10: Encoding data as JSON

If you need to create a `JSONObject` and put data in it, consider the following example:

```
// Create a new javax.json.JSONObject instance.
JSONObject first = new JSONObject();

first.put("foo", "bar");
first.put("temperature", 21.5);
first.put("year", 2016);

// Add a second object.
JSONObject second = new JSONObject();
second.put("Hello", "world");
first.put("message", second);

// Create a new JSONArray with some values
JSONArray someMonths = new JSONArray(new String[] { "January", "February" });
someMonths.put("March");
// Add another month as the fifth element, leaving the 4th element unset.
someMonths.put(4, "May");

// Add the array to our object
object.put("months", someMonths);

// Encode
String json = object.toString();

// An exercise for the reader: Add pretty-printing!
/* {
       "foo":"bar",
       "temperature":21.5,
       "year":2016,
       "message":{"Hello":"world"},
       "months":["January","February","March",null,"May"]
   }
*/
```

Section 107.11: Decoding JSON data

If you need to get data from a `JSONObject`, consider the following example:

```
String json =
"{\"foo\":\"bar\",\"temperature\":21.5,\"year\":2016,\"message\":{\"Hello\":\"world\"},\"months\":[
\"January\",\"February\",\"March\",null,\"May\"]}";
```

```
// Decode the JSON-encoded string
JSONObject object = new JSONObject(json);

// Retrieve some values
String foo = object.getString("foo");
double temperature = object.getDouble("temperature");
int year = object.getInt("year");

// Retrieve another object
JSONObject secondary = object.getJSONObject("message");
String world = secondary.getString("Hello");

// Retrieve an array
JSONArray someMonths = object.getJSONArray("months");
// Get some values from the array
int nMonths = someMonths.length();
String february = someMonths.getString(1);
```

Chapter 108: XML Parsing using the JAXP APIs

Section 108.1: Parsing a document using the StAX API

Considering the following document :

```
<?xml version='1.0' encoding='UTF-8' ?>
<library>
   <book id='1'>Effective Java</book>
   <book id='2'>Java Concurrency In Practice</book>
   <notABook id='3'>This is not a book element</notABook>
</library>
```

One can use the following code to parse it and build a map of book titles by book id.

```
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamConstants;
import javax.xml.stream.XMLStreamReader;
import java.io.StringReader;
import java.util.HashMap;
import java.util.Map;

public class StaxDemo {

public static void main(String[] args) throws Exception {
    String xmlDocument = "<?xml version='1.0' encoding='UTF-8' ?>"
            + "<library>"
                + "<book id='1'>Effective Java</book>"
                + "<book id='2'>Java Concurrency In Practice</book>"
                + "<notABook id='3'>This is not a book element </notABook>"
            + "</library>";

    XMLInputFactory xmlInputFactory = XMLInputFactory.newFactory();
    // Various flavors are possible, e.g. from an InputStream, a Source, ...
    XMLStreamReader xmlStreamReader = xmlInputFactory.createXMLStreamReader(new
StringReader(xmlDocument));

    Map<Integer, String> bookTitlesById = new HashMap<>();

    // We go through each event using a loop
    while (xmlStreamReader.hasNext()) {
        switch (xmlStreamReader.getEventType()) {
            case XMLStreamConstants.START_ELEMENT:
                System.out.println("Found start of element: " + xmlStreamReader.getLocalName());
                // Check if we are at the start of a <book> element
                if ("book".equals(xmlStreamReader.getLocalName())) {
                    int bookId = Integer.parseInt(xmlStreamReader.getAttributeValue("", "id"));
                    String bookTitle = xmlStreamReader.getElementText();
                    bookTitlesById.put(bookId, bookTitle);
                }
                break;
            // A bunch of other things are possible : comments, processing instructions,
Whitespace...
            default:
                break;
        }
        xmlStreamReader.next();
```

```
    }

    System.out.println(bookTitlesById);
}
```

This outputs :

```
Found start of element: library
Found start of element: book
Found start of element: book
Found start of element: notABook
{1=Effective Java, 2=Java Concurrency In Practice}
```

In this sample, one must be carreful of a few things :

1. The use of `xmlStreamReader.getAttributeValue` works because we have checked first that the parser is in the `START_ELEMENT` state. In evey other states (except `ATTRIBUTES`), the parser is mandated to throw `IllegalStateException`, because attributes can only appear at the beginning of elements.
2. same goes for `xmlStreamReader.getTextContent()`, it works because we are at a `START_ELEMENT` and we know in this document that the **<book>** element has no non-text child nodes.

For more complex documents parsing (deeper, nested elements, ...), it is a good practice to "delegate" the parser to sub-methods or other objets, e.g. have a `BookParser` class or method, and have it deal with every element from the START_ELEMENT to the END_ELEMENT of the book XML tag.

One can also use a `Stack` object to keep around important datas up and down the tree.

Section 108.2: Parsing and navigating a document using the DOM API

Considering the following document :

```
<?xml version='1.0' encoding='UTF-8' ?>
<library>
   <book id='1'>Effective Java</book>
   <book id='2'>Java Concurrency In Practice</book>
</library>
```

One can use the following code to build a DOM tree out of a `String` :

```
import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.InputSource;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import java.io.StringReader;

public class DOMDemo {

public static void main(String[] args) throws Exception {
    String xmlDocument = "<?xml version='1.0' encoding='UTF-8' ?>"
            + "<library>"
            + "<book id='1'>Effective Java</book>"
```

```
            + "<book id='2'>Java Concurrency In Practice</book>"
            + "</library>";

    DocumentBuilderFactory documentBuilderFactory = DocumentBuilderFactory.newInstance();
    // This is useless here, because the XML does not have namespaces, but this option is usefull to
know in cas
    documentBuilderFactory.setNamespaceAware(true);
    DocumentBuilder documentBuilder = documentBuilderFactory.newDocumentBuilder();
    // There are various options here, to read from an InputStream, from a file, ...
    Document document = documentBuilder.parse(new InputSource(new StringReader(xmlDocument)));

    // Root of the document
    System.out.println("Root of the XML Document: " +
document.getDocumentElement().getLocalName());

    // Iterate the contents
    NodeList firstLevelChildren = document.getDocumentElement().getChildNodes();
    for (int i = 0; i < firstLevelChildren.getLength(); i++) {
        Node item = firstLevelChildren.item(i);
        System.out.println("First level child found, XML tag name is: " + item.getLocalName());
        System.out.println("\tid attribute of this tag is : " +
item.getAttributes().getNamedItem("id").getTextContent());
    }

    // Another way would have been
    NodeList allBooks = document.getDocumentElement().getElementsByTagName("book");
}
}
```

The code yields the following :

```
Root of the XML Document: library
First level child found, XML tag name is: book
id attribute of this tag is : 1
First level child found, XML tag name is: book
id attribute of this tag is : 2
```

Chapter 109: XML XPath Evaluation

Section 109.1: Parsing multiple XPath Expressions in a single XML

Using the same example as **Evaluating a NodeList in an XML document**, here is how you would make multiple XPath calls efficiently:

Given the following XML document:

```
<documentation>
    <tags>
        <tag name="Java">
            <topic name="Regular expressions">
                <example>Matching groups</example>
                <example>Escaping metacharacters</example>
            </topic>
            <topic name="Arrays">
                <example>Looping over arrays</example>
                <example>Converting an array to a list</example>
            </topic>
        </tag>
        <tag name="Android">
            <topic name="Building Android projects">
                <example>Building an Android application using Gradle</example>
                <example>Building an Android application using Maven</example>
            </topic>
            <topic name="Layout resources">
                <example>Including layout resources</example>
                <example>Supporting multiple device screens</example>
            </topic>
        </tag>
    </tags>
</documentation>
```

This is how you would use XPath to evaluate multiple expressions in one document:

```
XPath xPath = XPathFactory.newInstance().newXPath(); //Make new XPath
DocumentBuilder builder = DocumentBuilderFactory.newInstance();
Document doc = builder.parse(new File("path/to/xml.xml")); //Specify XML file path

NodeList javaExampleNodes = (NodeList)
xPath.evaluate("/documentation/tags/tag[@name='Java']//example", doc, XPathConstants.NODESET);
//Evaluate the XPath
xPath.reset(); //Resets the xPath so it can be used again
NodeList androidExampleNodes = (NodeList)
xPath.evaluate("/documentation/tags/tag[@name='Android']//example", doc, XPathConstants.NODESET);
//Evaluate the XPath

...
```

Section 109.2: Parsing single XPath Expression multiple times in an XML

In this case, you want to have the expression compiled before the evaluations, so that each call to `evaluate` does not `compile` the same expression. The simple syntax would be:

```
XPath xPath = XPathFactory.newInstance().newXPath(); //Make new XPath
XPathExpression exp = xPath.compile("/documentation/tags/tag[@name='Java']//example");
DocumentBuilder builder = DocumentBuilderFactory.newInstance();
Document doc = builder.parse(new File("path/to/xml.xml")); //Specify XML file path

NodeList javaExampleNodes = (NodeList) exp.evaluate(doc, XPathConstants.NODESET); //Evaluate the
XPath from the already-compiled expression

NodeList javaExampleNodes2 = (NodeList) exp.evaluate(doc, XPathConstants.NODESET); //Do it again
```

Overall, two calls to `XPathExpression.evaluate()` will be much more efficient than two calls to `XPath.evaluate()`.

Section 109.3: Evaluating a NodeList in an XML document

Given the following XML document:

```
<documentation>
    <tags>
        <tag name="Java">
            <topic name="Regular expressions">
                <example>Matching groups</example>
                <example>Escaping metacharacters</example>
            </topic>
            <topic name="Arrays">
                <example>Looping over arrays</example>
                <example>Converting an array to a list</example>
            </topic>
        </tag>
        <tag name="Android">
            <topic name="Building Android projects">
                <example>Building an Android application using Gradle</example>
                <example>Building an Android application using Maven</example>
            </topic>
            <topic name="Layout resources">
                <example>Including layout resources</example>
                <example>Supporting multiple device screens</example>
            </topic>
        </tag>
    </tags>
</documentation>
```

The following retrieves all `example` nodes for the Java tag (Use this method if only evaluating XPath in the XML once. See other example for when multiple XPath calls are evaluated in the same XML file.):

```
XPathFactory xPathFactory = XPathFactory.newInstance();
XPath xPath = xPathFactory.newXPath(); //Make new XPath
InputSource inputSource = new InputSource("path/to/xml.xml"); //Specify XML file path

NodeList javaExampleNodes = (NodeList)
xPath.evaluate("/documentation/tags/tag[@name='Java']//example", inputSource,
XPathConstants.NODESET); //Evaluate the XPath
...
```

Chapter 110: XOM - XML Object Model

Section 110.1: Reading a XML file

In order to load the XML data with XOM you will need to make a `Builder` from which you can build it into a `Document`.

```
Builder builder = new Builder();
Document doc = builder.build(file);
```

To get the root element, the highest parent in the xml file, you need to use the `getRootElement()` on the `Document` instance.

```
Element root = doc.getRootElement();
```

Now the Element class has a lot of handy methods that make reading xml really easy. Some of the most useful are listed below:

- `getChildElements(String name)` - returns an `Elements` instance that acts as an array of elements
- `getFirstChildElement(String name)` - returns the first child element with that tag.
- `getValue()` - returns the value inside the element.
- `getAttributeValue(String name)` - returns the value of an attribute with the specified name.

When you call the `getChildElements()` you get a `Elements` instance. From this you can loop through and call the `get(int index)` method on it to retrieve all the elements inside.

```
Elements colors = root.getChildElements("color");
for (int q = 0; q < colors.size(); q++){
    Element color = colors.get(q);
}
```

Example: Here is an example of reading an XML File:

XML File:

```
<example>
    <person>
        <name>
            <first>Dan</first>
            <last>Smith</last>
        </name>
        <age unit="years">23</age>
        <fav_color>green</fav_color>
    </person>
    <person>
        <name>
            <first>Bob</first>
            <last>Autry</last>
        </name>
        <age unit="months">3</age>
        <fav_color>N/A</fav_color>
    </person>
</example>
```

Code for reading and printing it:

```java
import java.io.File;
import java.io.IOException;
import nu.xom.Builder;
import nu.xom.Document;
import nu.xom.Element;
import nu.xom.Elements;
import nu.xom.ParsingException;

public class XMLReader {

    public static void main(String[] args) throws ParsingException, IOException{
        File file = new File("insert path here");
        // builder builds xml data
        Builder builder = new Builder();
        Document doc = builder.build(file);

        // get the root element <example>
        Element root = doc.getRootElement();

        // gets all element with tag <person>
        Elements people = root.getChildElements("person");

        for (int q = 0; q < people.size(); q++){
            // get the current person element
            Element person = people.get(q);

            // get the name element and its children: first and last
            Element nameElement = person.getFirstChildElement("name");
            Element firstNameElement = nameElement.getFirstChildElement("first");
            Element lastNameElement = nameElement.getFirstChildElement("last");

            // get the age element
            Element ageElement = person.getFirstChildElement("age");

            // get the favorite color element
            Element favColorElement = person.getFirstChildElement("fav_color");

            String fName, lName, ageUnit, favColor;
            int age;

            try {
                fName = firstNameElement.getValue();
                lName = lastNameElement.getValue();
                age = Integer.parseInt(ageElement.getValue());
                ageUnit = ageElement.getAttributeValue("unit");
                favColor = favColorElement.getValue();

                System.out.println("Name: " + lName + ", " + fName);
                System.out.println("Age: " + age + " (" + ageUnit + ")");
                System.out.println("Favorite Color: " + favColor);
                System.out.println("----------------");

            } catch (NullPointerException ex){
                ex.printStackTrace();
            } catch (NumberFormatException ex){
                ex.printStackTrace();
            }
        }
    }

}
```

This will print out in the console:

```
Name: Smith, Dan
Age: 23 (years)
Favorite Color: green
----------------
Name: Autry, Bob
Age: 3 (months)
Favorite Color: N/A
----------------
```

Section 110.2: Writing to a XML File

Writing to a XML File using XOM is very similar to reading it except in this case we are making the instances instead of retrieving them off the root.

To make a new Element use the constructor `Element(String name)`. You will want to make a root element so that you can easily add it to a `Document`.

```
Element root = new Element("root");
```

The `Element` class has some handy methods for editing elements. They are listed below:

- `appendChild(String name)` - this will basically set the value of the element to name.
- `appendChild(Node node)` - this will make `node` the elements parent. (Elements are nodes so you can parse elements).
- `addAttribute(Attribute attribute)` - will add an attribute to the element.

The `Attribute` class has a couple of different constructors. The simplest one is `Attribute(String name, String value)`.

Once you have all of your elements add to your root element you can turn it into a `Document`. `Document` will take a `Element` as an argument in it's constructor.

You can use a `Serializer` to write your XML to a file. You will need to make a new output stream to parse in the constructor of `Serializer`.

```
FileOutputStream fileOutputStream = new FileOutputStream(file);
Serializer serializer = new Serializer(fileOutputStream, "UTF-8");
serializer.setIndent(4);
serializer.write(doc);
```

Example

Code:

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.UnsupportedEncodingException;
import nu.xom.Attribute;
import nu.xom.Builder;
import nu.xom.Document;
import nu.xom.Element;
import nu.xom.Elements;
import nu.xom.ParsingException;
import nu.xom.Serializer;
```

```
public class XMLWriter{

    public static void main(String[] args) throws UnsupportedEncodingException,
            IOException{
        // root element <example>
        Element root = new Element("example");

        // make a array of people to store
        Person[] people = {new Person("Smith", "Dan", "years", "green", 23),
            new Person("Autry", "Bob", "months", "N/A", 3)};

        // add all the people
        for (Person person : people){

            // make the main person element <person>
            Element personElement = new Element("person");

            // make the name element and it's children: first and last
            Element nameElement = new Element("name");
            Element firstNameElement = new Element("first");
            Element lastNameElement = new Element("last");

            // make age element
            Element ageElement = new Element("age");

            // make favorite color element
            Element favColorElement = new Element("fav_color");

            // add value to names
            firstNameElement.appendChild(person.getFirstName());
            lastNameElement.appendChild(person.getLastName());

            // add names to name
            nameElement.appendChild(firstNameElement);
            nameElement.appendChild(lastNameElement);

            // add value to age
            ageElement.appendChild(String.valueOf(person.getAge()));

            // add unit attribute to age
            ageElement.addAttribute(new Attribute("unit", person.getAgeUnit()));

            // add value to favColor
            favColorElement.appendChild(person.getFavoriteColor());

            // add all contents to person
            personElement.appendChild(nameElement);
            personElement.appendChild(ageElement);
            personElement.appendChild(favColorElement);

            // add person to root
            root.appendChild(personElement);
        }

        // create doc off of root
        Document doc = new Document(root);

        // the file it will be stored in
        File file = new File("out.xml");
        if (!file.exists()){
            file.createNewFile();
        }
```

```
        // get a file output stream ready
        FileOutputStream fileOutputStream = new FileOutputStream(file);

        // use the serializer class to write it all
        Serializer serializer = new Serializer(fileOutputStream, "UTF-8");
        serializer.setIndent(4);
        serializer.write(doc);
    }

    private static class Person {

        private String lName, fName, ageUnit, favColor;
        private int age;

        public Person(String lName, String fName, String ageUnit, String favColor, int age){
            this.lName = lName;
            this.fName = fName;
            this.age = age;
            this.ageUnit = ageUnit;
            this.favColor = favColor;
        }

        public String getLastName() { return lName; }
        public String getFirstName() { return fName; }
        public String getAgeUnit() { return ageUnit; }
        public String getFavoriteColor() { return favColor; }
        public int getAge() { return age; }
    }

}
```

This will be the contents of "out.xml":

```
<?xml version="1.0" encoding="UTF-8"?>
<example>
    <person>
        <name>
            <first>Dan</first>
            <last>Smith</last>
        </name>
        <age unit="years">23</age>
        <fav_color>green</fav_color>
    </person>
    <person>
        <name>
            <first>Bob</first>
            <last>Autry</last>
        </name>
        <age unit="months">3</age>
        <fav_color>N/A</fav_color>
    </person>
</example>
```

Chapter 111: Polymorphism

Polymorphism is one of main OOP(object oriented programming) concepts. Polymorphism word was derived from the greek words "poly" and "morphs". Poly means "many" and morphs means "forms" (many forms).

There are two ways to perform polymorphism. **Method Overloading** and **Method Overriding**.

Section 111.1: Method Overriding

Method overriding is the ability of subtypes to redefine (override) the behavior of their supertypes.

In Java, this translates to subclasses overriding the methods defined in the super class. In Java, all non-primitive variables are actually `references`, which are akin to pointers to the location of the actual object in memory. The `references` only have one type, which is the type they were declared with. However, they can point to an object of either their declared type or any of its subtypes.

When a method is called on a `reference`, the corresponding **method of the actual object being pointed to is invoked**.

```
class SuperType {
    public void sayHello(){
        System.out.println("Hello from SuperType");
    }

    public void sayBye(){
        System.out.println("Bye from SuperType");
    }
}

class SubType extends SuperType {
    // override the superclass method
    public void sayHello(){
        System.out.println("Hello from SubType");
    }
}

class Test {
    public static void main(String... args){
        SuperType superType = new SuperType();
        superType.sayHello(); // -> Hello from SuperType

        // make the reference point to an object of the subclass
        superType = new SubType();
        // behaviour is governed by the object, not by the reference
        superType.sayHello(); // -> Hello from SubType

        // non-overridden method is simply inherited
        superType.sayBye(); // -> Bye from SuperType
    }
}
```

Rules to keep in mind

To override a method in the subclass, the overriding method (i.e. the one in the subclass) **MUST HAVE**:

- same name
- same return type in case of primitives (a subclass is allowed for classes, this is also known as covariant return

types).
- same type and order of parameters
- it may throw only those exceptions that are declared in the throws clause of the superclass's method or exceptions that are subclasses of the declared exceptions. It may also choose NOT to throw any exception. The names of the parameter types do not matter. For example, void methodX(int i) is same as void methodX(int k)
- We are unable to Override final or Static methods. Only thing that we can do change only method body.

Section 111.2: Method Overloading

Method overloading, also known as **function overloading**, is the ability of a class to have multiple methods with the same name, granted that they differ in either number or type of arguments.

Compiler checks ***method signature*** for method overloading.

Method signature consists of three things -

1. Method name
2. Number of parameters
3. Types of parameters

If these three are same for any two methods in a class, then compiler throws **duplicate method error**.

This type of polymorphism is called *static* or *compile time* polymorphism because the appropriate method to be called is decided by the compiler during the compile time based on the argument list.

```
class Polymorph {

    public int add(int a, int b){
        return a + b;
    }

    public int add(int a, int b, int c){
        return a + b + c;
    }

    public float add(float a, float b){
        return a + b;
    }

    public static void main(String... args){
        Polymorph poly = new Polymorph();
        int a = 1, b = 2, c = 3;
        float d = 1.5, e = 2.5;

        System.out.println(poly.add(a, b));
        System.out.println(poly.add(a, b, c));
        System.out.println(poly.add(d, e));
    }

}
```

This will result in:

```
2
6
4.000000
```

Overloaded methods may be static or non-static. This also does not effect method overloading.

```
public class Polymorph {

    private static void methodOverloaded()
    {
        //No argument, private static method
    }

    private int methodOverloaded(int i)
    {
        //One argument private non-static method
        return i;
    }

    static int methodOverloaded(double d)
    {
        //static Method
        return 0;
    }

    public void methodOverloaded(int i, double d)
    {
        //Public non-static Method
    }
}
```

Also if you change the return type of method, we are unable to get it as method overloading.

```
public class Polymorph {

void methodOverloaded(){
    //No argument and No return type
}

int methodOverloaded(){
    //No argument and int return type
    return 0;
}
```

Section 111.3: Polymorphism and different types of overriding

From java tutorial

> The dictionary definition of polymorphism refers to a principle in biology in which an organism or species can have many different forms or stages. This principle can also be applied to object-oriented programming and languages like the Java language. ***Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.***

Have a look at this example to understand different types of overriding.

1. Base class provides no implementation and sub-class has to override complete method - (abstract)
2. Base class provides default implementation and sub-class can change the behaviour
3. Sub-class adds extension to base class implementation by calling `super.methodName()` as first statement
4. Base class defines structure of the algorithm (Template method) and sub-class will override a part of algorithm

code snippet:

```
import java.util.HashMap;

abstract class Game implements Runnable{

    protected boolean runGame = true;
    protected Player player1 = null;
    protected Player player2 = null;
    protected Player currentPlayer = null;

    public Game(){
        player1 = new Player("Player 1");
        player2 = new Player("Player 2");
        currentPlayer = player1;
        initializeGame();
    }

    /* Type 1: Let subclass define own implementation. Base class defines abstract method to force
        sub-classes to define implementation
    */

    protected abstract void initializeGame();

    /* Type 2: Sub-class can change the behaviour. If not, base class behaviour is applicable */
    protected void logTimeBetweenMoves(Player player){
        System.out.println("Base class: Move Duration: player.PlayerActTime -
player.MoveShownTime");
    }

    /* Type 3: Base class provides implementation. Sub-class can enhance base class implementation by
calling
        super.methodName() in first line of the child class method and specific implementation later
*/
    protected void logGameStatistics(){
        System.out.println("Base class: logGameStatistics:");
    }
    /* Type 4: Template method: Structure of base class can't be changed but sub-class can some part
of behaviour */
    protected void runGame() throws Exception{
        System.out.println("Base class: Defining the flow for Game:");
        while (runGame) {
            /*
            1. Set current player
            2. Get Player Move
            */
            validatePlayerMove(currentPlayer);
            logTimeBetweenMoves(currentPlayer);
            Thread.sleep(500);
            setNextPlayer();
        }
        logGameStatistics();
    }
    /* sub-part of the template method, which define child class behaviour */
    protected abstract void validatePlayerMove(Player p);

    protected void setRunGame(boolean status){
        this.runGame = status;
    }
    public void setCurrentPlayer(Player p){
        this.currentPlayer = p;
    }
```

```
    public void setNextPlayer(){
        if (currentPlayer == player1) {
            currentPlayer = player2;
        }else{
            currentPlayer = player1;
        }
    }
    public void run(){
        try{
            runGame();
        }catch(Exception err){
            err.printStackTrace();
        }
    }
}

class Player{
    String name;
    Player(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }
}

/* Concrete Game implementation  */
class Chess extends Game{
    public Chess(){
        super();
    }
    public void initializeGame(){
        System.out.println("Child class: Initialized Chess game");
    }
    protected void validatePlayerMove(Player p){
        System.out.println("Child class: Validate Chess move:" + p.getName());
    }
    protected void logGameStatistics(){
        super.logGameStatistics();
        System.out.println("Child class: Add Chess specific logGameStatistics:");
    }
}
class TicTacToe extends Game{
    public TicTacToe(){
        super();
    }
    public void initializeGame(){
        System.out.println("Child class: Initialized TicTacToe game");
    }
    protected void validatePlayerMove(Player p){
        System.out.println("Child class: Validate TicTacToe move:" + p.getName());
    }
}

public class Polymorphism{
    public static void main(String args[]){
        try{

            Game game = new Chess();
            Thread t1 = new Thread(game);
            t1.start();
            Thread.sleep(1000);
```

```
            game.setRunGame(false);
            Thread.sleep(1000);

            game = new TicTacToe();
            Thread t2 = new Thread(game);
            t2.start();
            Thread.sleep(1000);
            game.setRunGame(false);

        }catch(Exception err){
            err.printStackTrace();
        }
    }
}
```

Output:

```
Child class: Initialized Chess game
Base class: Defining the flow for Game:
Child class: Validate Chess move:Player 1
Base class: Move Duration: player.PlayerActTime - player.MoveShownTime
Child class: Validate Chess move:Player 2
Base class: Move Duration: player.PlayerActTime - player.MoveShownTime
Base class: logGameStatistics:
Child class: Add Chess specific logGameStatistics:

Child class: Initialized TicTacToe game
Base class: Defining the flow for Game:
Child class: Validate TicTacToe move:Player 1
Base class: Move Duration: player.PlayerActTime - player.MoveShownTime
Child class: Validate TicTacToe move:Player 2
Base class: Move Duration: player.PlayerActTime - player.MoveShownTime
Base class: logGameStatistics:
```

Section 111.4: Virtual functions

Virtual Methods are methods in Java that are non-static and without the keyword Final in front. All methods by default are virtual in Java. Virtual Methods play important roles in Polymorphism because children classes in Java can override their parent classes' methods if the function being overridden is non-static and has the same method signature.

There are, however, some methods that are not virtual. For example, if the method is declared private or with the keyword final, then the method is not Virtual.

Consider the following modified example of inheritance with Virtual Methods from this StackOverflow post How do virtual functions work in C# and Java? :

```
public class A{
    public void hello(){
        System.out.println("Hello");
    }

    public void boo(){
        System.out.println("Say boo");

    }
}
```

```
public class B extends A{
     public void hello(){
        System.out.println("No");
     }

    public void boo(){
        System.out.println("Say haha");

    }
}
```

If we invoke class B and call hello() and boo(), we would get "No" and "Say haha" as the resulting output because B overrides the same methods from A. Even though the example above is almost exactly the same as method overriding, it is important to understand that the methods in class A are all, by default, Virtual.

Additionally, we can implement Virtual methods using the abstract keyword. Methods declared with the keyword "abstract" does not have a method definition, meaning the method's body is not yet implemented. Consider the example from above again, except the boo() method is declared abstract:

```
public class A{
   public void hello(){
        System.out.println("Hello");
    }

    abstract void boo();
}

public class B extends A{
     public void hello(){
        System.out.println("No");
     }

    public void boo(){
        System.out.println("Say haha");

    }
}
```

If we invoke boo() from B, the output will still be "Say haha" since B inherits the abstract method boo() and makes boo () output "Say haha".

Sources used and further readings:

How do virtual functions work in C# and Java?

Check out this great answer that gives a much more complete information about Virtual functions:

Can you write virtual functions / methods in Java?

Section 111.5: Adding behaviour by adding classes without touching existing code

```
import java.util.ArrayList;
import java.util.List;

import static java.lang.System.out;

public class PolymorphismDemo {
```

```
    public static void main(String[] args) {
        List<FlyingMachine> machines = new ArrayList<FlyingMachine>();
        machines.add(new FlyingMachine());
        machines.add(new Jet());
        machines.add(new Helicopter());
        machines.add(new Jet());

        new MakeThingsFly().letTheMachinesFly(machines);
    }
}

class MakeThingsFly {
    public void letTheMachinesFly(List<FlyingMachine> flyingMachines) {
        for (FlyingMachine flyingMachine : flyingMachines) {
            flyingMachine.fly();
        }
    }
}

class FlyingMachine {
    public void fly() {
        out.println("No implementation");
    }
}

class Jet extends FlyingMachine {
    @Override
    public void fly() {
        out.println("Start, taxi, fly");
    }

    public void bombardment() {
        out.println("Fire missile");
    }
}

class Helicopter extends FlyingMachine {
    @Override
    public void fly() {
        out.println("Start vertically, hover, fly");
    }
}
```

Explanation

a) The `MakeThingsFly` class can work with everything that is of type `FlyingMachine`.

b) The method `letTheMachinesFly` also works without any change (!) when you add a new class, for example `PropellerPlane`:

```
public void letTheMachinesFly(List<FlyingMachine> flyingMachines) {
        for (FlyingMachine flyingMachine : flyingMachines) {
            flyingMachine.fly();
        }
    }
}
```

That's the power of polymorphism. You can implement the open-closed-principle with it.

Chapter 112: Encapsulation

Imagine you had a class with some pretty important variables and they were set (by other programmers from their code) to unacceptable values.Their code brought errors in your code. As a solution, In OOP, you allow the state of an object (stored in its variables) to be modified only through methods. Hiding the state of an object and providing all interaction through an objects methods is known as Data Encapsulation.

Section 112.1: Encapsulation to maintain invariants

There are two parts of a class: the interface and the implementation.

The interface is the exposed functionality of the class. Its public methods and variables are part of the interface.

The implementation is the internal workings of a class. Other classes shouldn't need to know about the implementation of a class.

Encapsulation refers to the practice of hiding the implementation of a class from any users of that class. This allows the class to make assumptions about its internal state.

For example, take this class representing an Angle:

```
public class Angle {

    private double angleInDegrees;
    private double angleInRadians;

    public static Angle angleFromDegrees(double degrees){
        Angle a = new Angle();
        a.angleInDegrees = degrees;
        a.angleInRadians = Math.PI*degrees/180;
        return a;
    }

    public static Angle angleFromRadians(double radians){
        Angle a = new Angle();
        a.angleInRadians = radians;
        a.angleInDegrees = radians*180/Math.PI;
        return a;
    }

    public double getDegrees(){
        return angleInDegrees;
    }

    public double getRadians(){
        return angleInRadians;
    }

    public void setDegrees(double degrees){
        this.angleInDegrees = degrees;
        this.angleInRadians = Math.PI*degrees/180;
    }

    public void setRadians(double radians){
        this.angleInRadians = radians;
        this.angleInDegrees = radians*180/Math.PI;
    }
    private Angle(){}
```

```
}
```

This class relies on a basic assumption (or *invariant*): **angleInDegrees and angleInRadians are always in sync**. If the class members were public, there would be no guarantees that the two representations of angles are correlated.

Section 112.2: Encapsulation to reduce coupling

Encapsulation allows you to make internal changes to a class without affecting any code that calls the class. This reduces *coupling*, or how much any given class relies on the implementation of another class.

For example, let's change the implementation of the Angle class from the previous example:

```
public class Angle {

    private double angleInDegrees;

    public static Angle angleFromDegrees(double degrees){
        Angle a = new Angle();
        a.angleInDegrees = degrees;
        return a;
    }

    public static Angle angleFromRadians(double radians){
        Angle a = new Angle();
        a.angleInDegrees = radians*180/Math.PI;
        return a;
    }

    public double getDegrees(){
        return angleInDegrees;
    }

    public double getRadians(){
        return angleInDegrees*Math.PI / 180;
    }

    public void setDegrees(double degrees){
        this.angleInDegrees = degrees;
    }

    public void setRadians(double radians){
        this.angleInDegrees = radians*180/Math.PI;
    }

    private Angle(){}
}
```

The implementation of this class has changed so that it only stores one representation of the angle and calculates the other angle when needed.

However, **the implementation changed, but the interface didn't**. If a calling class relied on accessing the angleInRadians method, it would need to be changed to use the new version of `Angle`. Calling classes shouldn't care about the internal representation of a class.

Chapter 113: Java Agents

Section 113.1: Modifying classes with agents

Firstly, make sure that the agent being used has the following attributes in the Manifest.mf:

```
Can-Redefine-Classes: true
Can-Retransform-Classes: true
```

Starting a java agent will let the agent access the class Instrumentation. With Instrumentation you can call *addTransformer(ClassFileTransformer transformer)*. ClassFileTransformers will let you rewrite the bytes of classes. The class has only a single method which supplies the ClassLoader that loads the class, the class's name, a java.lang.Class instance of it, it's ProtectionDomain, and lastly the bytes of the class itself.

It looks like this:

```
byte[] transform(ClassLoader loader, String className, Class<?> classBeingRedefined,
          ProtectionDomain protectionDomain, byte[] classfileBuffer)
```

Modifying a class purely from bytes can take ages. To remedy this there are libraries that can be used to convert the class bytes into something more usable.

In this example I'll be using ASM, but other alternatives like Javassist and BCEL have similar features.

```
ClassNode getNode(byte[] bytes) {
    // Create a ClassReader that will parse the byte array into a ClassNode
    ClassReader cr = new ClassReader(bytes);
    ClassNode cn = new ClassNode();
    try {
        // This populates the ClassNode
        cr.accept(cn, ClassReader.EXPAND_FRAMES);
        cr = null;
    } catch (Exception e) {
        e.printStackTrace();
    }
    return cn;
}
```

From here changes can be made to the ClassNode object. This makes changing field/method access incredibly easy. Plus with ASM's Tree API modifying the bytecode of methods is a breeze.

Once the edits are finished you can convert the ClassNode back into bytes with the following method and return them in the *transform* method:

```
public static byte[] getNodeBytes(ClassNode cn, boolean useMaxs) {
    ClassWriter cw = new ClassWriter(useMaxs ? ClassWriter.COMPUTE_MAXS :
ClassWriter.COMPUTE_FRAMES);
    cn.accept(cw);
    byte[] b = cw.toByteArray();
    return b;
}
```

Section 113.2: Adding an agent at runtime

Agents can be added to a JVM at runtime. To load an agent you will need to use the Attach API's

VirtualMachine.attatch(String id). You can then load a compiled agent jar with the following method:

```
public static void loadAgent(String agentPath) {
    String vmName = ManagementFactory.getRuntimeMXBean().getName();
    int index = vmName.indexOf('@');
    String pid = vmName.substring(0, index);
    try {
        File agentFile = new File(agentPath);
        VirtualMachine vm = VirtualMachine.attach(pid);
        vm.loadAgent(agentFile.getAbsolutePath(), "");
        VirtualMachine.attach(vm.id());
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

This will not call *premain((String agentArgs, Instrumentation inst)* in the loaded agent, but instead will call *agentmain(String agentArgs, Instrumentation inst)*. This requires *Agent-Class* to be set in the agent Manifest.mf.

Section 113.3: Setting up a basic agent

The Premain class will contain the method *"premain(String agentArgs Instrumentation inst)"*

Here is an example:

```
import java.lang.instrument.Instrumentation;

public class PremainExample {
    public static void premain(String agentArgs, Instrumentation inst) {
        System.out.println(agentArgs);
    }
}
```

When compiled into a jar file open the Manifest and ensure that it has the Premain-Class attribute.

Here is an example:

```
Premain-Class: PremainExample
```

To use the agent with another java program "myProgram" you must define the agent in the JVM arguments:

```
java -javaagent:PremainAgent.jar -jar myProgram.jar
```

Chapter 114: Varargs (Variable Argument)

Section 114.1: Working with Varargs parameters

Using varargs as a parameter for a method definition, it is possible to pass either an array or a sequence of arguments. If a sequence of arguments are passed, they are converted into an array automatically.

This example shows both an array and a sequence of arguments being passed into the `printVarArgArray()` method, and how they are treated identically in the code inside the method:

```
public class VarArgs {

    // this method will print the entire contents of the parameter passed in

    void printVarArgArray(int... x) {
        for (int i = 0; i < x.length; i++) {
            System.out.print(x[i] + ",");
        }
    }

    public static void main(String args[]) {
        VarArgs obj = new VarArgs();

        //Using an array:
        int[] testArray = new int[]{10, 20};
        obj.printVarArgArray(testArray);

        System.out.println(" ");

        //Using a sequence of arguments
        obj.printVarArgArray(5, 6, 5, 8, 6, 31);
    }
}
```

Output:

```
10,20,
5,6,5,8,6,31
```

If you define the method like this, it will give compile-time errors.

```
void method(String... a, int... b , int c){} //Compile time error (multiple varargs )

void method(int... a, String b){} //Compile time error (varargs must be the last argument
```

Section 114.2: Specifying a varargs parameter

```
void doSomething(String... strings) {
    for (String s : strings) {
        System.out.println(s);
    }
}
```

The three periods after the final parameter's type indicate that the final argument may be passed as an array or as a sequence of arguments. Varargs can be used only in the final argument position.

Chapter 115: Logging (java.util.logging)

Section 115.1: Logging complex messages (efficiently)

Let's look at a sample of logging which you can see in many programs:

```
public class LoggingComplex {

    private static final Logger logger =
        Logger.getLogger(LoggingComplex.class.getName());

    private int total = 50, orders = 20;
    private String username = "Bob";

    public void takeOrder() {
        // (...) making some stuff
        logger.fine(String.format("User %s ordered %d things (%d in total)",
                                  username, orders, total));
        // (...) some other stuff
    }

    // some other methods and calculations
}
```

The above example looks perfectly fine, but many programmers forgets that Java VM is stack machine. This means that all method's parameters are calculated **before** executing the method.

This fact is crucial for logging in Java, especially for logging something in low levels like `FINE`, `FINER`, `FINEST` which are disabled by default. Let's look at Java bytecode for the `takeOrder()` method.

The result for `javap -c LoggingComplex.class` is something like this:

```
public void takeOrder();
    Code:
       0: getstatic     #27 // Field logger:Ljava/util/logging/Logger;
       3: ldc           #45 // String User %s ordered %d things (%d in total)
       5: iconst_3
       6: anewarray     #3  // class java/lang/Object
       9: dup
      10: iconst_0
      11: aload_0
      12: getfield      #40 // Field username:Ljava/lang/String;
      15: aastore
      16: dup
      17: iconst_1
      18: aload_0
      19: getfield      #36 // Field orders:I
      22: invokestatic  #47 // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
      25: aastore
      26: dup
      27: iconst_2
      28: aload_0
      29: getfield      #34 // Field total:I
      32: invokestatic  #47 // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
      35: aastore
      36: invokestatic  #53 // Method
java/lang/String.format:(Ljava/lang/String;[Ljava/lang/Object;)Ljava/lang/String;
      39: invokevirtual #59 // Method java/util/logging/Logger.fine:(Ljava/lang/String;)V
```

```
42: return
```

Line 39 runs the actual logging. All of the previous work (loading variables, creating new objects, concatenating Strings in `format` method) can be for nothing if logging level is set higher then `FINE` (and by default it is). Such logging can be very inefficient, consuming unnecessary memory and processor resources.

That's why you should ask if the level you want to use is enabled.

The right way should be:

```
public void takeOrder() {
    // making some stuff
    if (logger.isLoggable(Level.FINE)) {
        // no action taken when there's no need for it
        logger.fine(String.format("User %s ordered %d things (%d in total)",
                                  username, orders, total));
    }
    // some other stuff
}
```

Since Java 8:

The Logger class has additional methods that take a `Supplier<String>` as parameter, which can simply be provided by a lambda:

```
public void takeOrder() {
    // making some stuff
    logger.fine(() -> String.format("User %s ordered %d things (%d in total)",
            username, orders, total));
    // some other stuff
}
```

The Suppliers `get()`method - in this case the lambda - is only called when the corresponding level is enabled and so the `if`construction is not needed anymore.

Section 115.2: Using the default logger

This example shows how to use the default logging api.

```
import java.util.logging.Level;
import java.util.logging.Logger;

public class MyClass {

    // retrieve the logger for the current class
    private static final Logger LOG = Logger.getLogger(MyClass.class.getName());

    public void foo() {
        LOG.info("A log message");
        LOG.log(Level.INFO, "Another log message");

        LOG.fine("A fine message");

        // logging an exception
        try {
            // code might throw an exception
        } catch (SomeException ex) {
            // log a warning printing "Something went wrong"
```

```
            // together with the exception message and stacktrace
            LOG.log(Level.WARNING, "Something went wrong", ex);
        }

        String s = "Hello World!";

        // logging an object
        LOG.log(Level.FINER, "String s: {0}", s);

        // logging several objects
        LOG.log(Level.FINEST, "String s: {0} has length {1}", new Object[]{s, s.length()});
    }

}
```

Section 115.3: Logging levels

Java Logging Api has 7 levels. The levels in descending order are:

- `SEVERE` (highest value)
- `WARNING`
- `INFO`
- `CONFIG`
- `FINE`
- `FINER`
- `FINEST` (lowest value)

The default level is `INFO` (but this depends on the system and used a virtual machine).

Note: There are also levels `OFF` (can be used to turn logging off) and `ALL` (the opposite of `OFF`).

Code example for this:

```
import java.util.logging.Logger;

public class Levels {
    private static final Logger logger = Logger.getLogger(Levels.class.getName());

    public static void main(String[] args) {

        logger.severe("Message logged by SEVERE");
        logger.warning("Message logged by WARNING");
        logger.info("Message logged by INFO");
        logger.config("Message logged by CONFIG");
        logger.fine("Message logged by FINE");
        logger.finer("Message logged by FINER");
        logger.finest("Message logged by FINEST");

        // All of above methods are really just shortcut for
        // public void log(Level level, String msg):
        logger.log(Level.FINEST, "Message logged by FINEST");
    }
}
```

By default running this class will output only messages with level higher then `CONFIG`:

```
Jul 23, 2016 9:16:11 PM LevelsExample main
SEVERE: Message logged by SEVERE
```

```
Jul 23, 2016 9:16:11 PM LevelsExample main
WARNING: Message logged by WARNING
Jul 23, 2016 9:16:11 PM LevelsExample main
INFO: Message logged by INFO
```

Chapter 116: log4j / log4j2

Apache Log4j is a Java-based logging utility, it is one of several Java logging frameworks. This topic is to show how to setup and configure Log4j in Java with detailed examples on all of its possible aspects of usage.

Section 116.1: Properties-File to log to DB

For this example to work you'll need a JDBC driver compatible to the system the database is running on. An opensource one that allows you to connect to DB2 databases on an IBM System i can be found here: JT400

Even though this example is DB2 specific, it works for almost every other system if you exchange the driver and adapt the JDBC URL.

```
# Root logger option
log4j.rootLogger= ERROR, DB

# Redirect log messages to a DB2
# Define the DB appender
log4j.appender.DB=org.apache.log4j.jdbc.JDBCAppender

# Set JDBC URL (!!! adapt to your target system !!!)
log4j.appender.DB.URL=jdbc:as400://10.10.10.1:446/DATABASENAME;naming=system;errors=full;

# Set Database Driver (!!! adapt to your target system !!!)
log4j.appender.DB.driver=com.ibm.as400.access.AS400JDBCDriver

# Set database user name and password
log4j.appender.DB.user=USER
log4j.appender.DB.password=PASSWORD

# Set the SQL statement to be executed.
log4j.appender.DB.sql=INSERT INTO DB.TABLENAME VALUES('%d{yyyy-MM-
dd}','%d{HH:mm:ss}','%C','%p','%m')

# Define the layout for file appender
log4j.appender.DB.layout=org.apache.log4j.PatternLayout
```

Section 116.2: How to get Log4j

Current version (log4j2)

Using Maven:
Add the following dependency to your `POM.xml` file:

```
<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.6.2</version>
  </dependency>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.6.2</version>
  </dependency>
</dependencies>
```

Using Ivy:

```
<dependencies>
  <dependency org="org.apache.logging.log4j" name="log4j-api" rev="2.6.2" />
  <dependency org="org.apache.logging.log4j" name="log4j-core" rev="2.6.2" />
</dependencies>
```

Using Gradle:

```
dependencies {
  compile group: 'org.apache.logging.log4j', name: 'log4j-api', version: '2.6.2'
  compile group: 'org.apache.logging.log4j', name: 'log4j-core', version: '2.6.2'
}
```

Getting log4j 1.x

Note: Log4j 1.x has reached End-of-Life (EOL) (see Remarks).

Using Maven:

Declare this dependency in the `POM.xml` file:

```
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
```

Using Ivy:

```
<dependency org="log4j" name="log4j" rev="1.2.17"/>
```

Usign Gradle:

```
compile group: 'log4j', name: 'log4j', version: '1.2.17'
```

Using Buildr:

```
'log4j:log4j:jar:1.2.17'
```

Adding manually in path build:

Download from Log4j website project

Section 116.3: Setting up property file

Log4j gives you posibility to log data into console and file at same time. Create a `log4j.properties` file and put inside this basic configuration:

```
# Root logger option
log4j.rootLogger=DEBUG, stdout, file

# Redirect log messages to console
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
```

```
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n

# Redirect log messages to a log file, support file rolling.
log4j.appender.file=org.apache.log4j.RollingFileAppender
log4j.appender.file.File=C:\\log4j-application.log
log4j.appender.file.MaxFileSize=5MB
log4j.appender.file.MaxBackupIndex=10
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n
```

If you are using maven, put this propertie file in path:

```
/ProjectFolder/src/java/resources
```

Section 116.4: Basic log4j2.xml configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
  <Appenders>
    <Console name="STDOUT" target="SYSTEM_OUT">
      <PatternLayout pattern="%d %-5p [%t] %C{2} %m%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Root level="debug">
      <AppenderRef ref="STDOUT"/>
    </Root>
  </Loggers>
</Configuration>
```

This is a basic log4j2.xml configuration which has a console appender and a root logger. The pattern layout specifies which pattern should be used for logging the statements.
In order to debug the loading of log4j2.xml you can add the attribute `status = <WARN | DEBUG | ERROR | FATAL | TRACE | INFO>` in the configuration tag of your log4j2.xml.
You can also add a monitor interval so that it loads the configuration again after the specified interval period. The monitor interval can be added to the configuration tag as follows: `monitorInterval = 30`. It means that the config will be loaded every 30 seconds.

Section 116.5: How to use Log4j in Java code

First need to create a **`final static`** `logger` object:

```
final static Logger logger = Logger.getLogger(classname.class);
```

Then, call logging methods:

```
//logs an error message
logger.info("Information about some param: " + parameter); // Note that this line could throw a
NullPointerException!

//in order to improve performance, it is advised to use the `isXXXEnabled()` Methods
if( logger.isInfoEnabled() ){
    logger.info("Information about some param: " + parameter);
}

// In log4j2 parameter substitution is preferable due to readability and performance
// The parameter substitution only takes place if info level is active which obsoletes the use of
```

```
isXXXEnabled().
logger.info("Information about some param: {}" , parameter);

//logs an exception
logger.error("Information about some error: ", exception);
```

Section 116.6: Migrating from log4j 1.x to 2.x

If you want to migrate from existing log4j 1.x in your project to log4j 2.x then remove all existing log4j 1.x dependencies and add the following dependency:

Log4j 1.x API Bridge

Maven Build

```
<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-1.2-api</artifactId>
    <version>2.6.2</version>
  </dependency>
</dependencies>
```

Ivy Build

```
<dependencies>
  <dependency org="org.apache.logging.log4j" name="log4j-1.2-api" rev="2.6.2" />
</dependencies>
```

Gradle Build

```
dependencies {
  compile group: 'org.apache.logging.log4j', name: 'log4j-1.2-api', version: '2.6.2'
}
```

Apache Commons Logging Bridge If your project is using Apache Commons Logging which use log4j 1.x and you want to migrate it to log4j 2.x then add the following dependencies:

Maven Build

```
<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-jcl</artifactId>
    <version>2.6.2</version>
  </dependency>
</dependencies>
```

Ivy Build

```
<dependencies>
  <dependency org="org.apache.logging.log4j" name="log4j-jcl" rev="2.6.2" />
</dependencies>
```

Gradle Build

```
dependencies {
  compile group: 'org.apache.logging.log4j', name: 'log4j-jcl', version: '2.6.2'
}
```

Note: Do not remove any existing dependencies of Apache commons logging

Reference: https://logging.apache.org/log4j/2.x/maven-artifacts.html

Section 116.7: Filter Logoutput by level (log4j 1.x)

You can use a filter to log only messages "lower" than e.g. ERROR level. **But the filter is not supported by PropertyConfigurator. So you must change to XML config to use it**. See log4j-Wiki about filters.

Example "specific level"

```
<appender name="info-out" class="org.apache.log4j.FileAppender">
            <param name="File" value="info.log"/>
            <layout class="org.apache.log4j.PatternLayout">
                    <param name="ConversionPattern" value="%m%n"/>
            </layout>
            <filter class="org.apache.log4j.varia.LevelMatchFilter">
                    <param name="LevelToMatch" value="info" />
                    <param name="AcceptOnMatch" value="true"/>
            </filter>
            <filter class="org.apache.log4j.varia.DenyAllFilter" />
</appender>
```

Or "Level range"

```
<appender name="info-out" class="org.apache.log4j.FileAppender">
            <param name="File" value="info.log"/>
            <layout class="org.apache.log4j.PatternLayout">
                    <param name="ConversionPattern" value="%m%n"/>
            </layout>
            <filter class="org.apache.log4j.varia.LevelRangeFilter">
                    <param name="LevelMax" value="info"/>
                    <param name="LevelMin" value="info"/>
                    <param name="AcceptOnMatch" value="true"/>
            </filter>
</appender>
```

Chapter 117: Oracle Official Code Standard

Oracle official style guide for the Java Programming Language is a standard followed by developers at Oracle and recommended to be followed by any other Java developer. It covers filenames, file organization, indentation, comments, declarations, statements, white space, naming conventions, programming practices and includes a code example.

Section 117.1: Naming Conventions

Package names

- Package names should be all lower case without underscores or other special characters.
- Package names begin with the reversed authority part of the web address of the company of the developer. This part can be followed a by project/program structure dependent package substructure.
- Don't use plural form. Follow the convention of the standard API which uses for instance `java.lang.annotation` and not `java.lang.annotations`.
- **Examples:** `com.yourcompany.widget.button`, `com.yourcompany.core.api`

Class, Interface and Enum Names

- Class and enum names should typically be nouns.
- Interface names should typically be nouns or adjectives ending with …able.
- Use mixed case with the first letter in each word in upper case (i.e. CamelCase).
- Match the regular expression `^[A-Z][a-zA-Z0-9]*$`.
- Use whole words and avoid using abbreviations unless the abbreviation is more widely used than the long form.
- Format an abbreviation as a word if the it is part of a longer class name.
- **Examples:** `ArrayList`, `BigInteger`, `ArrayIndexOutOfBoundsException`, `Iterable`.

Method Names

Method names should typically be verbs or other descriptions of actions

- They should match the regular expression `^[a-z][a-zA-Z0-9]*$`.
- Use mixed case with the first letter in lower case.
- **Examples:** `toString`, `hashCode`

Variables

Variable names should be in mixed case with the first letter in lower case

- Match the regular expression `^[a-z][a-zA-Z0-9]*$`
- Further recommendation: Variables
- **Examples:** `elements`, `currentIndex`

Type Variables

For simple cases where there are few type variables involved use a single upper case letter.

- Match the regular expression `^[A-Z][0-9]?$`
- If one letter is more descriptive than another (such as `K` and `V` for keys and values in maps or `R` for a function return type) use that, otherwise use `T`.
- For complex cases where single letter type variables become confusing, use longer names written in all capital letters and use underscore (_) to separate words.

- **Examples:** `T`, `V`, `SRC_VERTEX`

Constants

Constants (**`static final`** fields whose content is immutable, by language rules or by convention) should be named with all capital letters and underscore (_) to separate words.

- Match the regular expression `^[A-Z][A-Z0-9]*(_[A-Z0-9]+)*$`
- **Examples:** `BUFFER_SIZE`, `MAX_LEVEL`

Other guidelines on naming

- Avoid hiding/shadowing methods, variables and type variables in outer scopes.
- Let the verbosity of the name correlate to the size of the scope. (For instance, use descriptive names for fields of large classes and brief names for local short-lived variables.)
- When naming public static members, let the identifier be self descriptive if you believe they will be statically imported.
- Further reading: Naming Section (in the official Java Style Guide)

Source: Java Style Guidelines from Oracle

Section 117.2: Class Structure

Order of class members

Class members should be ordered as follows:

1. Fields (in order of public, protected and private)
2. Constructors
3. Factory methods
4. Other Methods (in order of public, protected and private)

Ordering fields and methods primarily by their access modifiers or identifier is not required.

Here is an example of this order:

```
class Example {

    private int i;

    Example(int i) {
        this.i = i;
    }

    static Example getExample(int i) {
        return new Example(i);
    }

    @Override
    public String toString() {
        return "An example [" + i + "]";
    }

}
```

Grouping of class members

- Related fields should be grouped together.
- A nested type may be declared right before its first use; otherwise it should be declared before the fields.
- Constructors and overloaded methods should be grouped together by functionality and ordered with increasing arity. This implies that delegation among these constructs flow downward in the code.
- Constructors should be grouped together without other members between.
- Overloaded variants of a method should be grouped together without other members between.

Section 117.3: Annotations

Declaration annotations should be put on a separate line from the declaration being annotated.

```
@SuppressWarnings("unchecked")
public T[] toArray(T[] typeHolder) {
    ...
}
```

However, few or short annotations annotating a single-line method may be put on the same line as the method if it improves readability. For example, one may write:

```
@Nullable String getName() { return name; }
```

For a matter of consistency and readability, either all annotations should be put on the same line or each annotation should be put on a separate line.

```
// Bad.
@Deprecated @SafeVarargs
@CustomAnnotation
public final Tuple<T> extend(T... elements) {
    ...
}

// Even worse.
@Deprecated @SafeVarargs
@CustomAnnotation public final Tuple<T> extend(T... elements) {
    ...
}

// Good.
@Deprecated
@SafeVarargs
@CustomAnnotation
public final Tuple<T> extend(T... elements) {
    ...
}

// Good.
@Deprecated @SafeVarargs @CustomAnnotation
public final Tuple<T> extend(T... elements) {
    ...
}
```

Section 117.4: Import statements

```
// First java/javax packages
import java.util.ArrayList;
import javax.tools.JavaCompiler;
```

```
// Then third party libraries
import com.fasterxml.jackson.annotation.JsonProperty;

// Then project imports
import com.example.my.package.ClassA;
import com.example.my.package.ClassB;

// Then static imports (in the same order as above)
import static java.util.stream.Collectors.toList;
```

- Import statements should be sorted…
    - …primarily by non-static / static with non-static imports first.
    - …secondarily by package origin according to the following order
        - java packages
        - javax packages
        - external packages (e.g. org.xml)
        - internal packages (e.g. com.sun)
    - …tertiary by package and class identifier in lexicographical order
- Import statements should not be line wrapped, regardless of whether it exceeds the recommended maximum length of a line.
- No unused imports should be present.

Wildcard imports

- Wildcard imports should in general not be used.
- When importing a large number of closely-related classes (such as implementing a visitor over a tree with dozens of distinct “node” classes), a wildcard import may be used.
- In any case, no more than one wildcard import per file should be used.

Section 117.5: Braces

```
class Example {
    void method(boolean error) {
        if (error) {
            Log.error("Error occurred!");
            System.out.println("Error!");
        } else { // Use braces since the other block uses braces.
            System.out.println("No error");
        }
    }
}
```

- Opening braces should be put on the end of the current line rather than on a line by its own.
- There should be a new line in front of a closing brace unless the block is empty (see Short Forms below)
- Braces are recommended even where the language makes them optional, such as single-line if and loop bodies.
    - If a block spans more than one line (including comments) it must have braces.
    - If one of the blocks in a `if` / **`else`** statement has braces, the other block must too.
    - If the block comes last in an enclosing block, it must have braces.
- The **`else`**, **`catch`** and the **`while`** keyword in **`do`**…**`while`** loops go on the same line as the closing brace of the

preceding block.

Short forms

```
enum Response { YES, NO, MAYBE }
public boolean isReference() { return true; }
```

The above recommendations are intended to improve uniformity (and thus increase familiarity / readability). In some cases "short forms" that deviate from the above guidelines are just as readable and may be used instead. These cases include for instance simple enum declarations and trivial methods and lambda expressions.

Section 117.6: Redundant Parentheses

```
return flag ? "yes" : "no";

String cmp = (flag1 != flag2) ? "not equal" : "equal";

// Don't do this
return (flag ? "yes" : "no");
```

- Redundant grouping parentheses (i.e. parentheses that does not affect evaluation) may be used if they improve readability.
- Redundant grouping parentheses should typically be left out in shorter expressions involving common operators but included in longer expressions or expressions involving operators whose precedence and associativity is unclear without parentheses. Ternary expressions with non-trivial conditions belong to the latter.
- The entire expression following a `return` keyword must not be surrounded by parentheses.

Section 117.7: Modifiers

```
class ExampleClass {
    // Access modifiers first (don't do for instance "static public")
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}

interface ExampleInterface {
    // Avoid 'public' and 'abstract' since they are implicit
    void sayHello();
}
```

- Modifiers should go in the following order
    - Access modifier (**`public`** / **`private`** / **`protected`**)
    - **`abstract`**
    - **`static`**
    - **`final`**
    - **`transient`**
    - **`volatile`**
    - **`default`**
    - **`synchronized`**
    - **`native`**
    - **`strictfp`**

- Modifiers should not be written out when they are implicit. For example, interface methods should neither be declared `public` nor `abstract`, and nested enums and interfaces should not be declared static.
- Method parameters and local variables should not be declared `final` unless it improves readability or documents an actual design decision.
- Fields should be declared `final` unless there is a compelling reason to make them mutable.

Section 117.8: Indentation

- Indentation level is **four spaces**.
- Only space characters may be used for indentation. **No tabs.**
- Empty lines must not be indented. (This is implied by the no trailing white space rule.)
- `case` lines should be indented with four spaces, and statements within the case should be indented with another four spaces.

```
switch (var) {
    case TWO:
        setChoice("two");
        break;
    case THREE:
        setChoice("three");
        break;
    default:
        throw new IllegalArgumentException();
}
```

Refer to Wrapping statements for guidelines on how to indent continuation lines.

Section 117.9: Literals

```
long l = 5432L;
int i = 0x123 + 0xABC;
byte b = 0b1010;
float f1 = 1 / 5432f;
float f2 = 0.123e4f;
double d1 = 1 / 5432d;  // or 1 / 5432.0
double d2 = 0x1.3p2;
```

- `long` literals should use the upper case letter `L` suffix.
- Hexadecimal literals should use upper case letters `A-F`.
- All other numerical prefixes, infixes, and suffixes should use lowercase letters.

Section 117.10: Package declaration

```
package com.example.my.package;
```

The package declaration should not be line wrapped, regardless of whether it exceeds the recommended maximum length of a line.

Section 117.11: Lambda Expressions

```
Runnable r = () -> System.out.println("Hello World");

Supplier<String> c = () -> "Hello World";
```

```
// Collection::contains is a simple unary method and its behavior is
// clear from the context. A method reference is preferred here.
appendFilter(goodStrings::contains);

// A lambda expression is easier to understand than just tempMap::put in this case
trackTemperature((time, temp) -> tempMap.put(time, temp));
```

- Expression lambdas are preferred over single-line block lambdas.
- Method references should generally be preferred over lambda expressions.
- For bound instance method references, or methods with arity greater than one, a lambda expression may be easier to understand and therefore preferred. Especially if the behavior of the method is not clear from the context.
- The parameter types should be omitted unless they improve readability.
- If a lambda expression stretches over more than a few lines, consider creating a method.

Section 117.12: Java Source Files

- All lines must be terminated with a line feed character (LF, ASCII value 10) and not for instance CR or CR+LF.
- There may be no trailing white space at the end of a line.
- The name of a source file must equal the name of the class it contains followed by the `.java` extension, even for files that only contain a package private class. This does not apply to files that do not contain any class declarations, such as `package-info.java`.

Section 117.13: Wrapping statements

- Source code and comments should generally not exceed 80 characters per line and rarely if ever exceed 100 characters per line, including indentation.

  The character limit must be judged on a case by case basis. What really matters is the semantical "density" and readability of the line. Making lines gratuitously long makes them hard to read; similarly, making "heroic attempts" to fit them into 80 columns can also make them hard to read. The flexibility outlined here aims to enable developers to avoid these extremes, not to maximize use of monitor real-estate.

- URLs or example commands should not be wrapped.

```
// Ok even though it might exceed max line width when indented.
Error e = isTypeParam
        ? Errors.InvalidRepeatableAnnotationNotApplicable(targetContainerType, on)
        : Errors.InvalidRepeatableAnnotationNotApplicableInContext(targetContainerType));

// Wrapping preferable
String pretty = Stream.of(args)
                      .map(Argument::prettyPrint)
                      .collectors(joining(", "));

// Too strict interpretation of max line width. Readability suffers.
Error e = isTypeParam
        ? Errors.InvalidRepeatableAnnotationNotApplicable(
                targetContainerType, on)
        : Errors.InvalidRepeatableAnnotationNotApplicableInContext(
                targetContainerType);

// Should be wrapped even though it fits within the character limit
```

```
String pretty = Stream.of(args).map(Argument::prettyPrint).collectors(joining(", "));
```

- Wrapping at a higher syntactical level is preferred over wrapping at a lower syntactical level.
- There should be at most 1 statement per line.
- A continuation line should be indented in one of the following four ways
    - **Variant 1**: With 8 extra spaces relative to the indentation of the previous line.
    - **Variant 2**: With 8 extra spaces relative to the starting column of the wrapped expression.
    - **Variant 3**: Aligned with previous sibling expression (as long as it is clear that it's a continuation line)
    - **Variant 4**: Aligned with previous method call in a chained expression.

Section 117.14: Wrapping Method Declarations

```
int someMethod(String aString,
               List<Integer> aList,
               Map<String, String> aMap,
               int anInt,
               long aLong,
               Set<Number> aSet,
               double aDouble) {
    ...
}

int someMethod(String aString, List<Integer> aList,
        Map<String, String> aMap, int anInt, long aLong,
        double aDouble, long aLong) {
    ...
}

int someMethod(String aString,
               List<Map<Integer, StringBuffer>> aListOfMaps,
               Map<String, String> aMap)
        throws IllegalArgumentException {
    ...
}

int someMethod(String aString, List<Integer> aList,
        Map<String, String> aMap, int anInt)
                throws IllegalArgumentException {
    ...
}
```

- Method declarations can be formatted by listing the arguments vertically, or by a new line and +8 extra spaces
- If a throws clause needs to be wrapped, put the line break in front of the throws clause and make sure it stands out from the argument list, either by indenting +8 relative to the function declaration, or +8 relative to the previous line.

Section 117.15: Wrapping Expressions

- If a line approaches the maximum character limit, always consider breaking it down into multiple statements / expressions instead of wrapping the line.
- Break before operators.
- Break before the . in chained method calls.

```
popupMsg("Inbox notification: You have "
        + newMsgs + " new messages");

// Don't! Looks like two arguments
popupMsg("Inbox notification: You have " +
         newMsgs + " new messages");
```

Section 117.16: Whitespace

Vertical Whitespace

- A single blank line should be used to separate…
    - Package declaration
    - Class declarations
    - Constructors
    - Methods
    - Static initializers
    - Instance initializers
- …and may be used to separate logical groups of
    - import statements
    - fields
    - statements
- Multiple consecutive blank lines should only be used to separate groups of related members and not as the standard inter-member spacing.

Horizontal Whitespace

- A single space should be used…
    - To separate keywords from neighboring opening or closing brackets and braces
    - Before and after all binary operators and operator like symbols such as arrows in lambda expressions and the colon in enhanced for loops (but not before the colon of a label)
    - After `//` that starts a comment.
    - After commas separating arguments and semicolons separating the parts of a for loop.
    - After the closing parenthesis of a cast.
- In variable declarations it is not recommended to align types and variables.

Section 117.17: Special Characters

- Apart from LF the only allowed white space character is Space (ASCII value 32). Note that this implies that other white space characters (in, for instance, string and character literals) must be written in escaped form.
- `\'`, `\"`, `\\`, `\t`, `\b`, `\r`, `\f`, and `\n` should be preferred over corresponding octal (e.g. `\047`) or Unicode (e.g. `\u0027`) escaped characters.
- Should there be a need to go against the above rules for the sake of testing, the test should *generate* the required input programmatically.

Section 117.18: Variable Declarations

- One variable per declaration (and at most one declaration per line)
- Square brackets of arrays should be at the type (`String[] args`) and not on the variable (`String args[]`).
- Declare a local variable right before it is first used, and initialize it as close to the declaration as possible.

Chapter 118: Character encoding

Section 118.1: Reading text from a file encoded in UTF-8

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;

public class ReadingUTF8TextFile {

    public static void main(String[] args) throws IOException {
        //StandardCharsets is available since Java 1.7
        //for ealier version use Charset.forName("UTF-8");
        try (BufferedWriter wr = Files.newBufferedWriter(Paths.get("test.txt"),
StandardCharsets.UTF_8)) {
            wr.write("Strange cyrillic symbol Ы");
        }
        /* First Way. For big files */
        try (BufferedReader reader = Files.newBufferedReader(Paths.get("test.txt"),
StandardCharsets.UTF_8)) {

            String line;
            while ((line = reader.readLine()) != null) {
                System.out.print(line);
            }
        }

        System.out.println(); //just separating output

        /* Second way. For small files */
        String s = new String(Files.readAllBytes(Paths.get("test.txt")), StandardCharsets.UTF_8);
        System.out.print(s);
    }
}
```

Section 118.2: Writing text to a file in UTF-8

```
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;

public class WritingUTF8TextFile {
    public static void main(String[] args) throws IOException {
        //StandardCharsets is available since Java 1.7
        //for ealier version use Charset.forName("UTF-8");
        try (BufferedWriter wr = Files.newBufferedWriter(Paths.get("test2.txt"),
StandardCharsets.UTF_8)) {
            wr.write("Cyrillic symbol Ы");
        }
    }
}
```

Section 118.3: Getting byte representation of a string in UTF-8

```
import java.nio.charset.StandardCharsets;
import java.util.Arrays;

public class GetUtf8BytesFromString {

    public static void main(String[] args) {
        String str = "Cyrillic symbol Ы";
        //StandardCharsets is available since Java 1.7
        //for ealier version use Charset.forName("UTF-8");
        byte[] textInUtf8 = str.getBytes(StandardCharsets.UTF_8);

        System.out.println(Arrays.toString(textInUtf8));
    }
}
```

Chapter 119: Apache Commons Lang

Section 119.1: Implement equals() method

To implement the `equals` method of an object easily you could use the `EqualsBuilder` class.

Selecting the fields:

```
@Override
public boolean equals(Object obj) {

    if(!(obj instanceof MyClass)) {
        return false;
    }
    MyClass theOther = (MyClass) obj;

    EqualsBuilder builder = new EqualsBuilder();
    builder.append(field1, theOther.field1);
    builder.append(field2, theOther.field2);
    builder.append(field3, theOther.field3);

    return builder.isEquals();
}
```

Using reflection:

```
@Override
public boolean equals(Object obj) {
    return EqualsBuilder.reflectionEquals(this, obj, false);
}
```

the boolean parameter is to indicates if the equals should check transient fields.

Using reflection avoiding some fields:

```
@Override
public boolean equals(Object obj) {
    return EqualsBuilder.reflectionEquals(this, obj, "field1", "field2");
}
```

Section 119.2: Implement hashCode() method

To implement the `hashCode` method of an object easily you could use the `HashCodeBuilder` class.

Selecting the fields:

```
@Override
public int hashCode() {

    HashCodeBuilder builder = new HashCodeBuilder();
    builder.append(field1);
    builder.append(field2);
    builder.append(field3);

    return builder.hashCode();
}
```

Using reflection:

```
@Override
public int hashCode() {
    return HashCodeBuilder.reflectionHashCode(this, false);
}
```

the boolean parameter indicates if it should use transient fields.

Using reflection avoiding some fields:

```
@Override
public int hashCode() {
    return HashCodeBuilder.reflectionHashCode(this, "field1", "field2");
}
```

Section 119.3: Implement toString() method

To implement the `toString` method of an object easily you could use the `ToStringBuilder` class.

Selecting the fields:

```
@Override
public String toString() {

    ToStringBuilder builder = new ToStringBuilder(this);
    builder.append(field1);
    builder.append(field2);
    builder.append(field3);

    return builder.toString();
}
```

Example result:

```
ar.com.jonat.lang.MyClass@dd7123[<null>,0,false]
```

Explicitly giving names to the fields:

```
@Override
public String toString() {

    ToStringBuilder builder = new ToStringBuilder(this);
    builder.append("field1",field1);
    builder.append("field2",field2);
    builder.append("field3",field3);

    return builder.toString();
}
```

Example result:

```
ar.com.jonat.lang.MyClass@dd7404[field1=<null>,field2=0,field3=false]
```

You could change the style via parameter:

```
@Override
```

```
public String toString() {

    ToStringBuilder builder = new ToStringBuilder(this,
            ToStringStyle.MULTI_LINE_STYLE);
    builder.append("field1", field1);
    builder.append("field2", field2);
    builder.append("field3", field3);

    return builder.toString();
}
```

Example result:

```
ar.com.bna.lang.MyClass@ebbf5c[
  field1=<null>
  field2=0
  field3=false
]
```

There are some styles, for example JSON, no Classname, short, etc ...

Via reflection:

```
@Override
public String toString() {
    return ToStringBuilder.reflectionToString(this);
}
```

You could also indicate the style:

```
@Override
public String toString() {
    return ToStringBuilder.reflectionToString(this, ToStringStyle.JSON_STYLE);
}
```

Chapter 120: Localization and Internationalization

Section 120.1: Locale

The `java.util.Locale` class is used to represent a "geographical, political or cultural" region to localize a given text, number, date or operation to. A Locale object may thus contain a country, region, language, and also a variant of a language, for instance a dialect spoken in a certain region of a country, or spoken in a different country than the country from which the language originates.

The Locale instance is handed to components that need to localize their actions, whether it is converting the input, output, or just need it for internal operations. The Locale class cannot do any internationalization or localization by itself

Language

The language must be an ISO 639 2 or 3 character language code, or a registered language subtag of up to 8 characters. In case a language has both a 2 and 3 character language code, use the 2 character code. A full list of language codes can be found in the IANA Language Subtag Registry.

Language codes are case insensitive, but the Locale class always use lowercase versions of the language codes

Creating a Locale

Creating a `java.util.Locale` instance can be done in four different ways:

```
Locale constants
Locale constructors
Locale.Builder class
Locale.forLanguageTag factory method
```

Java ResourceBundle

You create a ResourceBundle instance like this:

```
Locale locale = new Locale("en", "US");
ResourceBundle labels = ResourceBundle.getBundle("i18n.properties");
System.out.println(labels.getString("message"));
```

Consider I have a property file `i18n.properties`:

```
message=This is locale
```

Output:

```
This is locale
```

Setting Locale

If you want to reproduce the state using other languages, you can use **`setDefault()`** method. Its usage:

```
setDefault(Locale.JAPANESE); //Set Japanese
```

Section 120.2: Automatically formatted Dates using "locale"

`SimpleDateFormatter` is great in a pinch, but like the name suggests it doesn't scale well.

If you hard-code `"MM/dd/yyyy"` all over your application your international users won't be happy.

Let Java do the work for you

Use the **`static`** methods in `DateFormat` to retrieve the right formatting for your user. For a desktop application (where you'll rely on the default locale), simply call:

```
String localizedDate = DateFormat.getDateInstance(style).format(date);
```

Where `style` is one of the formatting constants (`FULL`, `LONG`, `MEDIUM`, `SHORT`, etc.) specified in `DateFormat`.

For a server-side application where the user specifies their locale as part of the request, you should pass it explicitly to `getDateInstance()` instead:

```
String localizedDate =
    DateFormat.getDateInstance(style, request.getLocale()).format(date);
```

Section 120.3: String Comparison

Compare two Strings ignoring case:

```
"School".equalsIgnoreCase("school"); // true
```

Don't use

```
text1.toLowerCase().equals(text2.toLowerCase());
```

Languages have different rules for converting upper and lower case. A 'I' would be converted to 'i' in English. But in Turkish a 'I' becomes a 'ı'. If you have to use `toLowerCase()` use the overload which expects a `Locale`: `String.toLowerCase(Locale)`.

Comparing two Strings ignoring minor differences:

```
Collator collator = Collator.getInstance(Locale.GERMAN);
collator.setStrength(Collator.PRIMARY);
collator.equals("Gärten", "gaerten"); // returns true
```

Sort Strings respecting natural language order, ignoring case (use collation key to:

```
String[] texts = new String[] {"Birne", "äther", "Apfel"};
Collator collator = Collator.getInstance(Locale.GERMAN);
collator.setStrength(Collator.SECONDARY); // ignore case
Arrays.sort(texts, collator::compare); // will return {"Apfel", "äther", "Birne"}
```

Chapter 121: Parallel programming with Fork/Join framework

Section 121.1: Fork/Join Tasks in Java

The fork/join framework in Java is ideal for a problem that can be divided into smaller pieces and solved in parallel. The fundamental steps of a fork/join problem are:

- Divide the problem into multiple pieces
- Solve each of the pieces in parallel to each other
- Combine each of the sub-solutions into one overall solution

A ForkJoinTask is the interface that defines such a problem. It is generally expected that you will subclass one of its abstract implementations (usually the RecursiveTask) rather than implement the interface directly.

In this example, we are going to sum a collection of integers, dividing until we get to batch sizes of no more than ten.

```
import java.util.List;
import java.util.concurrent.RecursiveTask;

public class SummingTask extends RecursiveTask<Integer> {
    private static final int MAX_BATCH_SIZE = 10;

    private final List<Integer> numbers;
    private final int minInclusive, maxExclusive;

    public SummingTask(List<Integer> numbers) {
        this(numbers, 0, numbers.size());
    }

    // This constructor is only used internally as part of the dividing process
    private SummingTask(List<Integer> numbers, int minInclusive, int maxExclusive) {
        this.numbers = numbers;
        this.minInclusive = minInclusive;
        this.maxExclusive = maxExclusive;
    }

    @Override
    public Integer compute() {
        if (maxExclusive - minInclusive > MAX_BATCH_SIZE) {
            // This is too big for a single batch, so we shall divide into two tasks
            int mid = (minInclusive + maxExclusive) / 2;
            SummingTask leftTask = new SummingTask(numbers, minInclusive, mid);
            SummingTask rightTask = new SummingTask(numbers, mid, maxExclusive);

            // Submit the left hand task as a new task to the same ForkJoinPool
            leftTask.fork();

            // Run the right hand task on the same thread and get the result
            int rightResult = rightTask.compute();

            // Wait for the left hand task to complete and get its result
            int leftResult = leftTask.join();

            // And combine the result
            return leftResult + rightResult;
        } else {
```

```
            // This is fine for a single batch, so we will run it here and now
            int sum = 0;
            for (int i = minInclusive; i < maxExclusive; i++) {
                sum += numbers.get(i);
            }
            return sum;
        }
    }
}
```

An instance of this task can now be passed to an instance of ForkJoinPool.

```
// Because I am not specifying the number of threads
// it will create a thread for each available processor
ForkJoinPool pool = new ForkJoinPool();

// Submit the task to the pool, and get what is effectively the Future
ForkJoinTask<Integer> task = pool.submit(new SummingTask(numbers));

// Wait for the result
int result = task.join();
```

Chapter 122: Non-Access Modifiers

Non-Access Modifiers ***do not change the accessibility of variables*** and methods, but they do provide them ***special properties***.

Section 122.1: final

`final` in Java can refer to variables, methods and classes. There are three simple rules:

- final variable cannot be reassigned
- final method cannot be overridden
- final class cannot be extended

Usages

Good Programming Practice

Some developer consider it good practice to mark a variable final when you can. If you have a variable that should not be changed, you should mark it final.

An important use of `final` keyword if for method parameters. If you want to emphasize that a method doesn't change its input parameters, mark the properties as final.

```
public int sumup(final List<Integer> ints);
```

This emphasizes that the `sumup` method is not going to change the `ints`.

Inner class Access

If your anonymous inner class wants to access a variable, the variable should be marked `final`

```
  public IPrintName printName(){
    String name;
    return new IPrintName(){
        @Override
        public void printName(){
            System.out.println(name);
        }
    };
}
```

This class doesn't compile, as the variable `name`, is not final.

`Version ≥ Java SE 8`

Effectively final variables are an exception. These are local variables that are written to only once and could therefore be made final. Effectively final variables can be accessed from anonymus classes too.

`final static` variable

Even though the code below is completely legal when `final` variable `foo` is not `static`, in case of `static` it will not compile:

```
class TestFinal {
    private final static List foo;
```

```
    public Test() {
        foo = new ArrayList();
    }
}
```

The reason is, let's repeat again, *final variable cannot be reassigned*. Since `foo` is static, it is shared among all instances of class `TestFinal`. When a new instance of a class `TestFinal` is created, its constructor is invoked and therefore foo gets reassigned which compiler does not allow. A correct way to initialize variable `foo` in this case is either:

```
class TestFinal {
    private static final List foo = new ArrayList();
    //..
}
```

or by using a static initializer:

```
class TestFinal {
    private static final List foo;
    static {
        foo = new ArrayList();
    }
    //..
}
```

`final` methods are useful when base class implements some important functionality that derived class is not supposed to change it. They are also faster than non-final methods, because there is no concept of virtual table involved.

All wrapper classes in Java are final, such as `Integer`, `Long` etc. Creators of these classes didn't want that anyone can e.g. extend Integer into his own class and change the basic behavior of Integer class. One of the requirements to make a class immutable is that subclasses may not override methods. The simplest way to do this is to declare the class as **`final`**.

Section 122.2: static

The **`static`** keyword is used on a class, method, or field to make them work independently of any instance of the class.

- Static fields are common to all instances of a class. They do not need an instance to access them.
- Static methods can be run without an instance of the class they are in. However, they can only access static fields of that class.
- Static classes can be declared inside of other classes. They do not need an instance of the class they are in to be instantiated.

```
public class TestStatic
{
    static int staticVariable;

    static {
        // This block of code is run when the class first loads
        staticVariable = 11;
    }

    int nonStaticVariable = 5;

    static void doSomething() {
```

```
        // We can access static variables from static methods
        staticVariable = 10;
    }

    void add() {
        // We can access both static and non-static variables from non-static methods
        nonStaticVariable += staticVariable;
    }

    static class StaticInnerClass {
        int number;
        public StaticInnerClass(int _number) {
            number = _number;
        }

        void doSomething() {
            // We can access number and staticVariable, but not nonStaticVariable
            number += staticVariable;
        }

        int getNumber() {
            return number;
        }
    }
}


// Static fields and methods
TestStatic object1 = new TestStatic();

System.out.println(object1.staticVariable); // 11
System.out.println(TestStatic.staticVariable); // 11

TestStatic.doSomething();

TestStatic object2 = new TestStatic();

System.out.println(object1.staticVariable); // 10
System.out.println(object2.staticVariable); // 10
System.out.println(TestStatic.staticVariable); // 10

object1.add();

System.out.println(object1.nonStaticVariable); // 15
System.out.println(object2.nonStaticVariable); // 10

// Static inner classes
StaticInnerClass object3 = new TestStatic.StaticInnerClass(100);
StaticInnerClass object4 = new TestStatic.StaticInnerClass(200);

System.out.println(object3.getNumber()); // 100
System.out.println(object4.getNumber()); // 200

object3.doSomething();

System.out.println(object3.getNumber()); // 110
System.out.println(object4.getNumber()); // 200
```

Section 122.3: abstract

Abstraction is a process of hiding the implementation details and showing only functionality to the user. An abstract

class can never be instantiated. If a class is declared as abstract then the sole purpose is for the class to be extended.

```
abstract class Car
{
    abstract void tagLine();
}

class Honda extends Car
{
    void tagLine()
    {
        System.out.println("Start Something Special");
    }
}

class Toyota extends Car
{
    void tagLine()
    {
        System.out.println("Drive Your Dreams");
    }
}
```

Section 122.4: strictfp

Version ≥ Java SE 1.2

strictfp modifier is used for floating-point calculations. This modifier makes floating point variable more consistent across multiple platforms and ensure all the floating point calculations are done according to IEEE 754 standards to avoid errors of calculation (round-off errors), overflows and underflows on both 32bit and 64bit architecture. This cannot be applied on abstract methods, variables or constructors.

```
// strictfp keyword can be applied on methods, classes and interfaces.

strictfp class A{}

strictfp interface M{}

class A{
    strictfp void m(){}
}
```

Section 122.5: volatile

The **volatile** modifier is used in multi threaded programming. If you declare a field as **volatile** it is a signal to threads that they must read the most recent value, not a locally cached one. Furthermore, **volatile** reads and writes are guaranteed to be atomic (access to a non-**volatile** **long** or **double** is not atomic), thus avoiding certain read/write errors between multiple threads.

```
public class MyRunnable implements Runnable
{
    private volatile boolean active;

    public void run(){ // run is called in one thread
        active = true;
        while (active){
            // some code here
```

```
        }
    }

    public void stop(){ // stop() is called from another thread
        active = false;
    }
}
```

Section 122.6: synchronized

Synchronized modifier is used to control the access of a particular method or a block by multiple threads. Only one thread can enter into a method or a block which is declared as synchronized. synchronized keyword works on intrinsic lock of an object, in case of a synchronized method current objects lock and static method uses class object. Any thread trying to execute a synchronized block must acquire the object lock first.

```
class Shared
{
    int i;

    synchronized void SharedMethod()
    {
        Thread t = Thread.currentThread();

        for(int i = 0; i <= 1000; i++)
        {
            System.out.println(t.getName()+" : "+i);
        }
    }

    void SharedMethod2()
    {
        synchronized (this)
        {
            System.out.println("Thais access to currect object is synchronize "+this);
        }
    }
}

public class ThreadsInJava
{
    public static void main(String[] args)
    {
        final Shared s1 = new Shared();

        Thread t1 = new Thread("Thread - 1")
        {
            @Override
            public void run()
            {
                s1.SharedMethod();
            }
        };

        Thread t2 = new Thread("Thread - 2")
        {
            @Override
            public void run()
            {
                s1.SharedMethod();
            }
```

```
        };

        t1.start();

        t2.start();
    }
}
```

Section 122.7: transient

A variable which is declared as transient will not be serialized during object serialization.

```
public transient int limit = 55;   // will not persist
public int b; // will persist
```

Chapter 123: Process

Section 123.1: Pitfall: Runtime.exec, Process and ProcessBuilder don't understand shell syntax

The `Runtime.exec(String ...)` and `Runtime.exec(String)` methods allow you to execute a command as an external process1. In the first version, you supply the command name and the command arguments as separate elements of the string array, and the Java runtime requests the OS runtime system to start the external command. The second version is deceptively easy to use, but it has some pitfalls.

First of all, here is an example of using `exec(String)` being used safely:

```
Process p = Runtime.exec("mkdir /tmp/testDir");
p.waitFor();
if (p.exitValue() == 0) {
    System.out.println("created the directory");
}
```

Spaces in pathnames

Suppose that we generalize the example above so that we can create an arbitrary directory:

```
Process p = Runtime.exec("mkdir " + dirPath);
// ...
```

This will typically work, but it will fail if `dirPath` is (for example) "/home/user/My Documents". The problem is that `exec(String)` splits the string into a command and arguments by simply looking for whitespace. The command string:

```
"mkdir /home/user/My Documents"
```

will be split into:

```
"mkdir", "/home/user/My", "Documents"
```

and this will cause the "mkdir" command to fail because it expects one argument, not two.

Faced with this, some programmers try to add quotes around the pathname. This doesn't work either:

```
"mkdir \"/home/user/My Documents\""
```

will be split into:

```
"mkdir", "\"/home/user/My", "Documents\""
```

The extra double-quote characters that were added in attempt to "quote" the spaces are treated like any other non-whitespace characters. Indeed, anything we do quote or escape the spaces is going to fail.

The way to deal with this particular problems is to use the `exec(String ...)` overload.

```
Process p = Runtime.exec("mkdir", dirPath);
// ...
```

This will work if `dirpath` includes whitespace characters because this overload of `exec` does not attempt to split the arguments. The strings are passed through to the OS `exec` system call as-is.

Redirection, pipelines and other shell syntax

Suppose that we want to redirect an external command's input or output, or run a pipeline. For example:

```
Process p = Runtime.exec("find / -name *.java -print 2>/dev/null");
```

or

```
Process p = Runtime.exec("find source -name *.java | xargs grep package");
```

(The first example lists the names of all Java files in the file system, and the second one prints the **`package`** statements2 in the Java files in the "source" tree.)

These are not going to work as expected. In the first case, the "find" command will be run with "2>/dev/null" as a command argument. It will not be interpreted as a redirection. In the second example, the pipe character ("|") and the works following it will be given to the "find" command.

The problem here is that the `exec` methods and `ProcessBuilder` do not understand any shell syntax. This includes redirections, pipelines, variable expansion, globbing, and so on.

In a few cases (for example, simple redirection) you can easily achieve the desired effect using `ProcessBuilder`. However, this is not true in general. An alternative approach is to run the command line in a shell; for example:

```
Process p = Runtime.exec("bash", "-c",
                         "find / -name *.java -print 2>/dev/null");
```

or

```
Process p = Runtime.exec("bash", "-c",
                         "find source -name \\*.java | xargs grep package");
```

But note that in the second example, we needed to escape the wildcard character ("*") because we want the wildcard to be interpreted by "find" rather than the shell.

Shell builtin commands don't work

Suppose the following examples won't work on a system with a UNIX-like shell:

```
Process p = Runtime.exec("cd", "/tmp");     // Change java app's home directory
```

or

```
Process p = Runtime.exec("export", "NAME=value");  // Export NAME to the java app's environment
```

There are a couple of reasons why this won't work:

1. On "cd" and "export" commands are shell builtin commands. They don't exist as distinct executables.
2. For shell builtins to do what they are supposed to do (e.g. change the working directory, update the environment), they need to change the place where that state resides. For a normal application (including a Java application) the state is associated with the application process. So for example, the child process that would run the "cd" command could not change the working directory of its parent "java" process. Similarly,

one `exec`'d process cannot change the working directory for a process that follows it.

This reasoning applies to all shell builtin commands.

1 - You can use `ProcessBuilder` as well, but that is not relevant to the point of this example.

2 - This is a bit rough and ready ... but once again, the failings of this approach are not relevant to the example.

Section 123.2: Simple example (Java version < 1.5)

This example will call the windows calculator. It's important to notice that the exit code will vary accordingly to the program/script that is being called.

```
package process.example;

import java.io.IOException;

public class App {

    public static void main(String[] args) {
        try {
            // Executes windows calculator
            Process p = Runtime.getRuntime().exec("calc.exe");

            // Wait for process until it terminates
            int exitCode = p.waitFor();

            System.out.println(exitCode);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Chapter 124: Java Native Access

Section 124.1: Introduction to JNA

What is JNA?

Java Native Access (JNA) is a community-developed library providing Java programs an easy access to native shared libraries (`.dll` files on windows, `.so` files on Unix ...)

How can I use it?

- Firstly, download the latest release of JNA and reference its jna.jar in your project's CLASSPATH.
- Secondly, copy, compile and run the Java code below

> *For the purpose of this introduction, we suppose the native platform in use is Windows. If you're running on another platform simply replace the string `"msvcrt"` with the string `"c"` in the code below.*

The small Java program below will print a message on the console by calling the C `printf` function.

CRuntimeLibrary.java

```
package jna.introduction;

import com.sun.jna.Library;
import com.sun.jna.Native;

// We declare the printf function we need and the library containing it (msvcrt)...
public interface CRuntimeLibrary extends Library {

   CRuntimeLibrary INSTANCE =
       (CRuntimeLibrary) Native.loadLibrary("msvcrt", CRuntimeLibrary.class);

   void printf(String format, Object... args);
}
```

MyFirstJNAProgram.java

```
package jna.introduction;

// Now we call the printf function...
public class MyFirstJNAProgram {
    public static void main(String args[]) {
         CRuntimeLibrary.INSTANCE.printf("Hello World from JNA !");
    }
}
```

Where to go now?

Jump into another topic here or jump to the official site.

Chapter 125: Modules

Section 125.1: Defining a basic module

Modules are defined in a file named `module-info.java`, named a module descriptor. It should be placed in the source-code root:

```
|-- module-info.java
|-- com
    |-- example
        |-- foo
            |-- Foo.java
        |-- bar
            |-- Bar.java
```

Here is a simple module descriptor:

```
module com.example {
    requires java.httpclient;
    exports com.example.foo;
}
```

The module name should be unique and it is recommended that you use the same Reverse-DNS naming notation as used by packages to help ensure this.

The module `java.base`, which contains Java's basic classes, is implicitly visible to any module and does not need to be included.

The `requires` declaration allows us to use other modules, in the example the module `java.httpclient` is imported.

A module can also specify which packages it `exports` and therefore makes it visible to other modules.

The package `com.example.foo` declared in the `exports` clause will be visible to other modules. Any sub-packages of `com.example.foo` will not be exported, they need their own `export` declarations.

Conversely, `com.example.bar` which is not listed in `exports` clauses will not be visible to other modules.

Chapter 126: Concurrent Programming (Threads)

Concurrent computing is a form of computing in which several computations are executed concurrently instead of sequentially. Java language is designed to support concurrent programming through the usage of threads. Objects and resources can be accessed by multiple threads; each thread can potentially access any object in the program and the programmer must ensure read and write access to objects is properly synchronized between threads.

Section 126.1: Callable and Future

While `Runnable` provides a means to wrap code to be executed in a different thread, it has a limitation in that it cannot return a result from the execution. The only way to get some return value from the execution of a `Runnable` is to assign the result to a variable accessible in a scope outside of the `Runnable`.

`Callable` was introduced in Java 5 as a peer to `Runnable`. `Callable` is essentially the same except it has a `call` method instead of `run`. The `call` method has the additional capability to return a result and is also allowed to throw checked exceptions.

The result from a *Callable task submission* is available to be tapped via a Future

`Future` can be considered a container of sorts that houses the result of the `Callable` computation. Computation of the callable can carry on in another thread, and any attempt to tap the result of a `Future` will block and will only return the result once it is available.

Callable Interface

```
public interface Callable<V> {
    V call() throws Exception;
}
```

Future

```
interface Future<V> {
    V get();
    V get(long timeout, TimeUnit unit);
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
}
```

Using Callable and Future example:

```
public static void main(String[] args) throws Exception {
    ExecutorService es = Executors.newSingleThreadExecutor();

    System.out.println("Time At Task Submission : " + new Date());
    Future<String> result = es.submit(new ComplexCalculator());
    // the call to Future.get() blocks until the result is available.So we are in for about a 10 sec
wait now
    System.out.println("Result of Complex Calculation is : " + result.get());
    System.out.println("Time At the Point of Printing the Result : " + new Date());
}
```

Our Callable that does a lengthy computation

```
public class ComplexCalculator implements Callable<String> {

    @Override
    public String call() throws Exception {
        // just sleep for 10 secs to simulate a lengthy computation
        Thread.sleep(10000);
        System.out.println("Result after a lengthy 10sec calculation");
        return "Complex Result"; // the result
    }
}
```

Output

```
Time At Task Submission : Thu Aug 04 15:05:15 EDT 2016
Result after a lengthy 10sec calculation
Result of Complex Calculation is : Complex Result
Time At the Point of Printing the Result : Thu Aug 04 15:05:25 EDT 2016
```

Other operations permitted on Future

While `get()` is the method to extract the actual result Future has provision

- `get(long timeout, TimeUnit unit)` defines maximum time period during current thread will wait for a result;
- To cancel the task call `cancel(mayInterruptIfRunning)`. The flag `mayInterrupt` indicates that task should be interrupted if it was started and is running right now;
- To check if task is completed/finished by calling `isDone()`;
- To check if the lengthy task were cancelled `isCancelled()`.

Section 126.2: CountDownLatch

CountDownLatch

> A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.

1. A `CountDownLatch` is initialized with a given count.
2. The await methods block until the current count reaches zero due to invocations of the `countDown()` method, after which all waiting threads are released and any subsequent invocations of await return immediately.
3. This is a one-shot phenomenon—the count cannot be reset. If you need a version that resets the count, consider using a `CyclicBarrier`.

Key Methods:

```
public void await() throws InterruptedException
```

> Causes the current thread to wait until the latch has counted down to zero, unless the thread is interrupted.

```
public void countDown()
```

> Decrements the count of the latch, releasing all waiting threads if the count reaches zero.

Example:

```
import java.util.concurrent.*;

class DoSomethingInAThread implements Runnable {
    CountDownLatch latch;
    public DoSomethingInAThread(CountDownLatch latch) {
        this.latch = latch;
    }
    public void run() {
        try {
            System.out.println("Do some thing");
            latch.countDown();
        } catch(Exception err) {
            err.printStackTrace();
        }
    }
}

public class CountDownLatchDemo {
    public static void main(String[] args) {
        try {
            int numberOfThreads = 5;
            if (args.length < 1) {
                System.out.println("Usage: java CountDownLatchDemo numberOfThreads");
                return;
            }
            try {
                numberOfThreads = Integer.parseInt(args[0]);
            } catch(NumberFormatException ne) {

            }
            CountDownLatch latch = new CountDownLatch(numberOfThreads);
            for (int n = 0; n < numberOfThreads; n++) {
                Thread t = new Thread(new DoSomethingInAThread(latch));
                t.start();
            }
            latch.await();
            System.out.println("In Main thread after completion of " + numberOfThreads + "
threads");
        } catch(Exception err) {
            err.printStackTrace();
        }
    }
}
```

output:

```
java CountDownLatchDemo 5
Do some thing
Do some thing
Do some thing
Do some thing
Do some thing
In Main thread after completion of 5 threads
```

Explanation:

1. `CountDownLatch` is initialized with a counter of 5 in Main thread
2. Main thread is waiting by using `await()` method.
3. Five instances of `DoSomethingInAThread` have been created. Each instance decremented the counter with `countDown()` method.
4. Once the counter becomes zero, Main thread will resume

Section 126.3: Basic Multithreading

If you have many tasks to execute, and all these tasks are not dependent of the result of the precedent ones, you can use **Multithreading** for your computer to do all this tasks at the same time using more processors if your computer can. This can make your program execution **faster** if you have some big independent tasks.

```
class CountAndPrint implements Runnable {

    private final String name;

    CountAndPrint(String name) {
        this.name = name;
    }

    /** This is what a CountAndPrint will do */
    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            System.out.println(this.name + ": " + i);
        }
    }

    public static void main(String[] args) {
        // Launching 4 parallel threads
        for (int i = 1; i <= 4; i++) {
            // `start` method will call the `run` method
            // of CountAndPrint in another thread
            new Thread(new CountAndPrint("Instance " + i)).start();
        }

        // Doing some others tasks in the main Thread
        for (int i = 0; i < 10000; i++) {
            System.out.println("Main: " + i);
        }
    }
}
```

The code of the run method of the various `CountAndPrint` instances will execute in non predictable order. A snippet of a sample execution might look like this:

```
Instance 4: 1
Instance 2: 1
Instance 4: 2
Instance 1: 1
Instance 1: 2
Main: 1
Instance 4: 3
Main: 2
Instance 3: 1
Instance 4: 4
...
```

Section 126.4: Locks as Synchronisation aids

Prior to Java 5's concurrent package introduction threading was more low level.The introduction of this package provided several higher level concurrent programming aids/constructs.

Locks are thread synchronisation mechanisms that essentially serve the same purpose as synchronized blocks or key words.

Intrinsic Locking

```
int count = 0; // shared among multiple threads

public void doSomething() {
    synchronized(this) {
        ++count; // a non-atomic operation
    }
}
```

Synchronisation using Locks

```
int count = 0; // shared among multiple threads

Lock lockObj = new ReentrantLock();
public void doSomething() {
    try {
        lockObj.lock();
        ++count; // a non-atomic operation
    } finally {
        lockObj.unlock(); // sure to release the lock without fail
    }
}
```

Locks also have functionality available that intrinsic locking does not offer, such as locking but remaining responsive to interruption, or trying to lock, and not block when unable to.

Locking, responsive to interruption

```
class Locky {
    int count = 0; // shared among multiple threads

    Lock lockObj = new ReentrantLock();

    public void doSomething() {
        try {
            try {
                lockObj.lockInterruptibly();
                ++count; // a non-atomic operation
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt(); // stopping
            }
        } finally {
            if (!Thread.currentThread().isInterrupted()) {
                lockObj.unlock(); // sure to release the lock without fail
            }
        }
    }
}
```

Only do something when able to lock

```
public class Locky2 {
    int count = 0; // shared among multiple threads

    Lock lockObj = new ReentrantLock();

    public void doSomething() {
        boolean locked = lockObj.tryLock(); // returns true upon successful lock
        if (locked) {
            try {
                ++count; // a non-atomic operation
            } finally {
                lockObj.unlock(); // sure to release the lock without fail
            }
        }
    }
}
```

There are several variants of lock available.For more details refer the api docs here

Section 126.5: Semaphore

A Semaphore is a high-level synchronizer that maintains a set of *permits* that can be acquired and released by threads. A Semaphore can be imagined as a counter of *permits* that will be decremented when a thread acquires, and incremented when a thread releases. If the amount of *permits* is `0` when a thread attempts to acquire, then the thread will block until a permit is made available (or until the thread is interrupted).

A semaphore is initialized as:

```
Semaphore semaphore = new Semaphore(1); // The int value being the number of permits
```

The Semaphore constructor accepts an additional boolean parameter for fairness. When set false, this class makes no guarantees about the order in which threads acquire permits. When fairness is set true, the semaphore guarantees that threads invoking any of the acquire methods are selected to obtain permits in the order in which their invocation of those methods was processed. It is declared in the following manner:

```
Semaphore semaphore = new Semaphore(1, true);
```

Now let's look at an example from javadocs, where Semaphore is used to control access to a pool of items. A Semaphore is used in this example to provide blocking functionality in order to ensure that there are always items to be obtained when `getItem()` is called.

```
class Pool {
    /*
     * Note that this DOES NOT bound the amount that may be released!
     * This is only a starting value for the Semaphore and has no other
     * significant meaning UNLESS you enforce this inside of the
     * getNextAvailableItem() and markAsUnused() methods
     */
    private static final int MAX_AVAILABLE = 100;
    private final Semaphore available = new Semaphore(MAX_AVAILABLE, true);

    /**
     * Obtains the next available item and reduces the permit count by 1.
     * If there are no items available, block.
     */
    public Object getItem() throws InterruptedException {
```

```
        available.acquire();
        return getNextAvailableItem();
    }

    /**
     * Puts the item into the pool and add 1 permit.
     */
    public void putItem(Object x) {
        if (markAsUnused(x))
            available.release();
    }

    private Object getNextAvailableItem() {
        // Implementation
    }

    private boolean markAsUnused(Object o) {
        // Implementation
    }
}
```

Section 126.6: Synchronization

In Java, there is a built-in language-level locking mechanism: the `synchronized` block, which can use any Java object as an intrinsic lock (i.e. every Java object may have a monitor associated with it).

Intrinsic locks provide atomicity to groups of statements. To understand what that means for us, let's have a look at an example where `synchronized` is useful:

```
private static int t = 0;
private static Object mutex = new Object();

public static void main(String[] args) {
    ExecutorService executorService = Executors.newFixedThreadPool(400); // The high thread count
is for demonstration purposes.
    for (int i = 0; i < 100; i++) {
        executorService.execute(() -> {
            synchronized (mutex) {
                t++;
                System.out.println(MessageFormat.format("t: {0}", t));
            }
        });
    }
    executorService.shutdown();
}
```

In this case, if it weren't for the `synchronized` block, there would have been multiple concurrency issues involved. The first one would be with the post increment operator (it isn't atomic in itself), and the second would be that we would be observing the value of t after an arbitrary amount of other threads has had the chance to modify it. However, since we acquired an intrinsic lock, there will be no race conditions here and the output will contain numbers from 1 to 100 in their normal order.

Intrinsic locks in Java are *mutexes* (i.e. mutual execution locks). Mutual execution means that if one thread has acquired the lock, the second will be forced to wait for the first one to release it before it can acquire the lock for itself. Note: An operation that may put the thread into the wait (sleep) state is called a *blocking operation*. Thus, acquiring a lock is a blocking operation.

Intrinsic locks in Java are *reentrant*. This means that if a thread attempts to acquire a lock it already owns, it will not

block and it will successfully acquire it. For instance, the following code will *not* block when called:

```
public void bar(){
    synchronized(this){
        ...
    }
}
public void foo(){
    synchronized(this){
        bar();
    }
}
```

Beside **synchronized** blocks, there are also **synchronized** methods.

The following blocks of code are practically equivalent (even though the bytecode seems to be different):

1. **synchronized** block on **this**:

```
public void foo() {
    synchronized(this) {
        doStuff();
    }
}
```

2. **synchronized** method:

```
public synchronized void foo() {
    doStuff();
}
```

Likewise for **static** methods, this:

```
class MyClass {
    ...
    public static void bar() {
        synchronized(MyClass.class) {
            doSomeOtherStuff();
        }
    }
}
```

has the same effect as this:

```
class MyClass {
    ...
    public static synchronized void bar() {
        doSomeOtherStuff();
    }
}
```

Section 126.7: Runnable Object

The `Runnable` interface defines a single method, `run()`, meant to contain the code executed in the thread.

The `Runnable` object is passed to the `Thread` constructor. And Thread's `start()` method is called.

Example

```
public class HelloRunnable implements Runnable {

    @Override
    public void run() {
        System.out.println("Hello from a thread");
    }

    public static void main(String[] args) {
        new Thread(new HelloRunnable()).start();
    }
}
```

Example in Java8:

```
public static void main(String[] args) {
    Runnable r = () -> System.out.println("Hello world");
    new Thread(r).start();
}
```

Runnable vs Thread subclass

A `Runnable` object employment is more general, because the `Runnable` object can subclass a class other than `Thread`.

`Thread` subclassing is easier to use in simple applications, but is limited by the fact that your task class must be a descendant of `Thread`.

A `Runnable` object is applicable to the high-level thread management APIs.

Section 126.8: Creating basic deadlocked system

A deadlock occurs when two competing actions wait for the other to finish, and thus neither ever does. In java there is one lock associated with each object. To avoid concurrent modification done by multiple threads on single object we can use **`synchronized`** keyword, but everything comes at a cost. Using **`synchronized`** keyword wrongly can lead to stuck systems called as deadlocked system.

Consider there are 2 threads working on 1 instance, Lets call threads as First and Second, and lets say we have 2 resources R1 and R2. First acquires R1 and also needs R2 for its completion while Second acquires R2 and needs R1 for completion.

so say at time t=0,

First has R1 and Second has R2. now First is waiting for R2 while Second is waiting for R1. this wait is indefinite and this leads to deadlock.

```
public class Example2 {

    public static void main(String[] args) throws InterruptedException {
        final DeadLock dl = new DeadLock();
        Thread t1 = new Thread(new Runnable() {

            @Override
            public void run() {
                // TODO Auto-generated method stub
                dl.methodA();
```

```
            }
        });

        Thread t2 = new Thread(new Runnable() {

            @Override
            public void run() {
                // TODO Auto-generated method stub
                try {
                    dl.method2();
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        });
        t1.setName("First");
        t2.setName("Second");
        t1.start();
        t2.start();
    }
}

class DeadLock {

    Object mLock1 = new Object();
    Object mLock2 = new Object();


    public void methodA() {
        System.out.println("methodA wait for mLock1  " + Thread.currentThread().getName());
        synchronized (mLock1) {
            System.out.println("methodA mLock1 acquired   " + Thread.currentThread().getName());
            try {
                Thread.sleep(100);
                method2();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }
    public void method2() throws InterruptedException {
        System.out.println("method2 wait for mLock2  " + Thread.currentThread().getName());
        synchronized (mLock2) {
            System.out.println("method2  mLock2 acquired   " + Thread.currentThread().getName());
            Thread.sleep(100);
            method3();
        }
    }
    public void method3() throws InterruptedException {
        System.out.println("method3  mLock1  "+ Thread.currentThread().getName());
        synchronized (mLock1) {
            System.out.println("method3   mLock1 acquired  " + Thread.currentThread().getName());
        }
    }
}
```

Output of this program:

```
methodA wait for mLock1  First
```

```
method2 wait for mLock2  Second
method2  mLock2 acquired   Second
methodA mLock1 acquired   First
method3  mLock1  Second
method2 wait for mLock2  First
```

Section 126.9: Creating a java.lang.Thread instance

There are two main approaches to creating a thread in Java. In essence, creating a thread is as easy as writing the code that will be executed in it. The two approaches differ in where you define that code.

In Java, a thread is represented by an object - an instance of java.lang.Thread or its subclass. So the first approach is to create that subclass and override the *run()* method.

Note: I'll use *Thread* to refer to the java.lang.Thread class and *thread* to refer to the logical concept of threads.

```
class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("Thread running!");
        }
    }
}
```

Now since we've already defined the code to be executed, the thread can be created simply as:

```
MyThread t = new MyThread();
```

The Thread class also contains a constructor accepting a string, which will be used as the thread's name. This can be particulary useful when debugging a multi thread program.

```
class MyThread extends Thread {
    public MyThread(String name) {
        super(name);
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("Thread running! ");
        }
    }
}

MyThread t = new MyThread("Greeting Producer");
```

The second approach is to define the code using java.lang.Runnable and its only method *run()*. The Thread class then allows you to execute that method in a separated thread. To achieve this, create the thread using a constructor accepting an instance of the Runnable interface.

```
Thread t = new Thread(aRunnable);
```

This can be very powerful when combined with lambdas or methods references (Java 8 only):

```
Thread t = new Thread(operator::hardWork);
```

You can specify the thread's name, too.

```
Thread t = new Thread(operator::hardWork, "Pi operator");
```

Practicaly speaking, you can use both approaches without worries. However the general wisdom says to use the latter.

For every of the four mentioned constructors, there is also an alternative accepting an instance of java.lang.ThreadGroup as the first parameter.

```
ThreadGroup tg = new ThreadGroup("Operators");
Thread t = new Thread(tg, operator::hardWork, "PI operator");
```

The ThreadGroup represents a set of threads. You can only add a Thread to a ThreadGroup using a Thread's constructor. The ThreadGroup can then be used to manage all its Threads together, as well as the Thread can gain information from its ThreadGroup.

So to sumarize, the Thread can be created with one of these public constructors:

```
Thread()
Thread(String name)
Thread(Runnable target)
Thread(Runnable target, String name)
Thread(ThreadGroup group, String name)
Thread(ThreadGroup group, Runnable target)
Thread(ThreadGroup group, Runnable target, String name)
Thread(ThreadGroup group, Runnable target, String name, long stackSize)
```

The last one allows us to define desired stack size for the new thread.

Often the code readability suffers when creating and configuring many Threads with same properties or from the same pattern. That's when java.util.concurrent.ThreadFactory can be used. This interface allows you to encapsulate the procedure of creating the thread through the factory pattern and its only method *newThread(Runnable)*.

```
class WorkerFactory implements ThreadFactory {
    private int id = 0;

    @Override
    public Thread newThread(Runnable r) {
        return new Thread(r, "Worker " + id++);
    }
}
```

Section 126.10: Atomic operations

An atomic operation is an operation that is executed "all at once", without any chance of other threads observing or modifying state during the atomic operation's execution.

Lets consider a **BAD EXAMPLE**.

```
private static int t = 0;

public static void main(String[] args) {
    ExecutorService executorService = Executors.newFixedThreadPool(400); // The high thread count
is for demonstration purposes.
    for (int i = 0; i < 100; i++) {
```

```
        executorService.execute(() -> {
            t++;
            System.out.println(MessageFormat.format("t: {0}", t));
        });
    }
    executorService.shutdown();
}
```

In this case, there are two issues. The first issue is that the post increment operator is *not* atomic. It is comprised of multiple operations: get the value, add 1 to the value, set the value. That's why if we run the example, it is likely that we won't see `t: 100` in the output - two threads may concurrently get the value, increment it, and set it: let's say the value of t is 10, and two threads are incrementing t. Both threads will set the value of t to 11, since the second thread observes the value of t before the first thread had finished incrementing it.

The second issue is with how we are observing t. When we are printing the value of t, the value may have already been changed by a different thread after this thread's increment operation.

To fix those issues, we'll use the `java.util.concurrent.atomic.AtomicInteger`, which has many atomic operations for us to use.

```
private static AtomicInteger t = new AtomicInteger(0);

public static void main(String[] args) {
    ExecutorService executorService = Executors.newFixedThreadPool(400); // The high thread count
is for demonstration purposes.
    for (int i = 0; i < 100; i++) {
        executorService.execute(() -> {
            int currentT = t.incrementAndGet();
            System.out.println(MessageFormat.format("t: {0}", currentT));
        });
    }
    executorService.shutdown();
}
```

The `incrementAndGet` method of `AtomicInteger` atomically increments and returns the new value, thus eliminating the previous race condition. Please note that in this example the lines will still be out of order because we make no effort to sequence the `println` calls and that this falls outside the scope of this example, since it would require synchronization and the goal of this example is to show how to use `AtomicInteger` to eliminate race conditions concerning state.

Section 126.11: Exclusive write / Concurrent read access

It is sometimes required for a process to concurrently write and read the same "data".

The `ReadWriteLock` interface, and its `ReentrantReadWriteLock` implementation allows for an access pattern that can be described as follow :

1. There can be any number of concurrent readers of the data. If there is at least one reader access granted, then no writer access is possible.
2. There can be at most one single writer to the data. If there is a writer access granted, then no reader can access the data.

An implementation could look like :

```
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;
```

```
public class Sample {

// Our lock. The constructor allows a "fairness" setting, which guarantees the chronology of lock
attributions.
protected static final ReadWriteLock RW_LOCK = new ReentrantReadWriteLock();

// This is a typical data that needs to be protected for concurrent access
protected static int data = 0;

/** This will write to the data, in an exclusive access */
public static void writeToData() {
    RW_LOCK.writeLock().lock();
    try {
        data++;
    } finally {
        RW_LOCK.writeLock().unlock();
    }
}

public static int readData() {
    RW_LOCK.readLock().lock();
    try {
        return data;
    } finally {
        RW_LOCK.readLock().unlock();
    }
}

}
```

NOTE 1 : This precise use case has a cleaner solution using `AtomicInteger`, but what is described here is an access pattern, that works regardless of the fact that data here is an integer that as an Atomic variant.

NOTE 2 : The lock on the reading part is really needed, although it might not look so to the casual reader. Indeed, if you do not lock on the reader side, any number of things can go wrong, amongst which :

1. The writes of primitive values are not guaranteed to be atomic on all JVMs, so the reader could see e.g. only 32bits of a 64bits write if `data` were a 64bits long type
2. The visibility of the write from a thread that did not perform it is guaranteed by the JVM only if we establish *Happen Before relationship* between the writes and the reads. This relationship is established when both readers and writers use their respective locks, but not otherwise

`Version ≥ Java SE 8`

In case higher performance is required, an under certain types of usage, there is a faster lock type available, called the `StampedLock`, that amongst other things implements an optimistic lock mode. This lock works very differently from the `ReadWriteLock`, and this sample is not transposable.

Section 126.12: Producer-Consumer

A simple example of producer-consumer problem solution. Notice that JDK classes (`AtomicBoolean` and `BlockingQueue`) are used for synchronization, which reduces the chance of creating an invalid solution. Consult Javadoc for various types of BlockingQueue; choosing different implementation may drastically change the behavior of this example (like DelayQueue or Priority Queue).

```
public class Producer implements Runnable {
```

```
    private final BlockingQueue<ProducedData> queue;

    public Producer(BlockingQueue<ProducedData> queue) {
        this.queue = queue;
    }

    public void run() {
        int producedCount = 0;
        try {
            while (true) {
                producedCount++;
                //put throws an InterruptedException when the thread is interrupted
                queue.put(new ProducedData());
            }
        } catch (InterruptedException e) {
            // the thread has been interrupted: cleanup and exit
            producedCount--;
            //re-interrupt the thread in case the interrupt flag is needeed higher up
            Thread.currentThread().interrupt();
        }
        System.out.println("Produced " + producedCount + " objects");
    }
}

public class Consumer implements Runnable {

    private final BlockingQueue<ProducedData> queue;

    public Consumer(BlockingQueue<ProducedData> queue) {
        this.queue = queue;
    }

    public void run() {
        int consumedCount = 0;
        try {
            while (true) {
                //put throws an InterruptedException when the thread is interrupted
                ProducedData data = queue.poll(10, TimeUnit.MILLISECONDS);
                // process data
                consumedCount++;
            }
        } catch (InterruptedException e) {
            // the thread has been interrupted: cleanup and exit
            consumedCount--;
            //re-interrupt the thread in case the interrupt flag is needeed higher up
            Thread.currentThread().interrupt();
        }
        System.out.println("Consumed " + consumedCount + " objects");
    }
}


public class ProducerConsumerExample {
    static class ProducedData {
        // empty data object
    }

    public static void main(String[] args) throws InterruptedException {
        BlockingQueue<ProducedData> queue = new ArrayBlockingQueue<ProducedData>(1000);
        // choice of queue determines the actual behavior: see various BlockingQueue implementations

        Thread producer = new Thread(new Producer(queue));
```

```
        Thread consumer = new Thread(new Consumer(queue));

        producer.start();
        consumer.start();

        Thread.sleep(1000);
        producer.interrupt();
        Thread.sleep(10);
        consumer.interrupt();
    }
}
```

Section 126.13: Visualizing read/write barriers while using synchronized / volatile

As we know that we should use **`synchronized`** keyword to make execution of a method or block exclusive. But few of us may not be aware of one more important aspect of using **`synchronized`** and **`volatile`** keyword: *apart from making a unit of code atomic, it also provides read / write barrier*. What is this read / write barrier? Let's discuss this using an example:

```
class Counter {

  private Integer count = 10;

  public synchronized void incrementCount() {
    count++;
  }

  public Integer getCount() {
    return count;
  }
}
```

Let's suppose a thread *A* calls `incrementCount()` first then another thread *B* calls `getCount()`. In this scenario there is no guarantee that B will see updated value of `count`. It may still see `count` as `10`, even it is also possible that it never sees updated value of `count` ever.

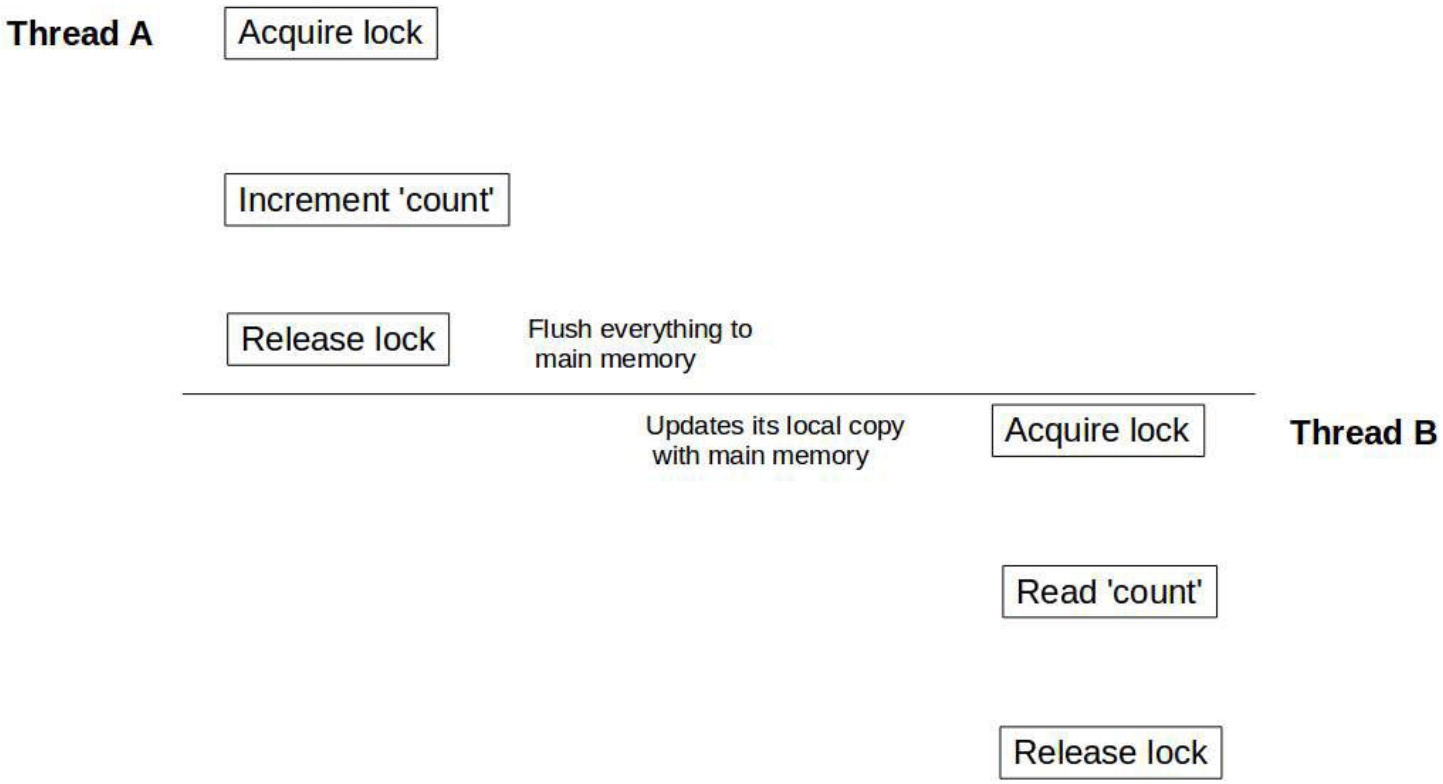
To understand this behavior we need to understand how Java memory model integrates with hardware architecture. In Java, each thread has it's own thread stack. This stack contains: method call stack and local variable created in that thread. In a multi core system, it is quite possible that two threads are running concurrently in separate cores. In such scenario it is possible that part of a thread's stack lies inside register / cache of a core. If inside a thread, an object is accessed using **`synchronized`** (or **`volatile`**) keyword, after **`synchronized`** block that thread syncs it's local copy of that variable with the main memory. This creates a read / write barrier and makes sure that the thread is seeing the latest value of that object.

But in our case, since thread B has not used synchronized access to `count`, it might be refering value of `count` stored in register and may never see updates from thread A. To make sure that B sees latest value of count we need to make `getCount()` synchronized as well.

```
public synchronized Integer getCount() {
  return count;
}
```

Now when thread A is done with updating `count` it unlocks `Counter` instance, at the same time creates write barrier and flushes all changes done inside that block to the main memory. Similarly when thread B acquires lock on the same instance of `Counter`, it enters into read barrier and reads value of `count` from main memory and sees all

updates.



Same visibility effect goes for **`volatile`** read / writes as well. All variables updated prior to write to **`volatile`** will be flushed to main memory and all reads after **`volatile`** variable read will be from main memory.

Section 126.14: Get status of all threads started by your program excluding system threads

Code snippet:

```
import java.util.Set;

public class ThreadStatus {
    public static void main(String args[]) throws Exception {
        for (int i = 0; i < 5; i++){
            Thread t = new Thread(new MyThread());
            t.setName("MyThread:" + i);
            t.start();
        }
        int threadCount = 0;
        Set<Thread> threadSet = Thread.getAllStackTraces().keySet();
        for (Thread t : threadSet) {
            if (t.getThreadGroup() == Thread.currentThread().getThreadGroup()) {
                System.out.println("Thread :" + t + ":" + "state:" + t.getState());
                ++threadCount;
            }
        }
        System.out.println("Thread count started by Main thread:" + threadCount);
    }
}

class MyThread implements Runnable {
    public void run() {
        try {
            Thread.sleep(2000);
        } catch(Exception err) {
```

```
            err.printStackTrace();
        }
    }
}
```

Output:

```
Thread :Thread[MyThread:1,5,main]:state:TIMED_WAITING
Thread :Thread[MyThread:3,5,main]:state:TIMED_WAITING
Thread :Thread[main,5,main]:state:RUNNABLE
Thread :Thread[MyThread:4,5,main]:state:TIMED_WAITING
Thread :Thread[MyThread:0,5,main]:state:TIMED_WAITING
Thread :Thread[MyThread:2,5,main]:state:TIMED_WAITING
Thread count started by Main thread:6
```

Explanation:

`Thread.getAllStackTraces().keySet()` returns all `Thread`s including application threads and system threads. If you are interested only in status of Threads, started by your application, iterate the `Thread` set by checking Thread Group of a particular thread against your main program thread.

In absence of above ThreadGroup condition, the program returns status of below System Threads:

```
Reference Handler
Signal Dispatcher
Attach Listener
Finalizer
```

Section 126.15: Using ThreadLocal

A useful tool in Java Concurrency is `ThreadLocal` – this allows you to have a variable that will be unique to a given thread. Thus, if the same code runs in different threads, these executions will not share the value, but instead each thread has its own variable that is *local to the thread*.

For example, this is frequently used to establish the context (such as authorization information) of handling a request in a servlet. You might do something like this:

```
private static final ThreadLocal<MyUserContext> contexts = new ThreadLocal<>();

public static MyUserContext getContext() {
    return contexts.get(); // get returns the variable unique to this thread
}

public void doGet(...) {
    MyUserContext context = magicGetContextFromRequest(request);
    contexts.put(context); // save that context to our thread-local - other threads
                           // making this call don't overwrite ours
    try {
        // business logic
    } finally {
        contexts.remove(); // 'ensure' removal of thread-local variable
    }
}
```

Now, instead of passing `MyUserContext` into every single method, you can instead use `MyServlet.getContext()` where you need it. Now of course, this does introduce a variable that needs to be documented, but it's thread-safe,

which eliminates a lot of the downsides to using such a highly scoped variable.

The key advantage here is that every thread has its own thread local variable in that `contexts` container. As long as you use it from a defined entry point (like demanding that each servlet maintains its context, or perhaps by adding a servlet filter) you can rely on this context being there when you need it.

Section 126.16: Multiple producer/consumer example with shared global queue

Below code showcases multiple Producer/Consumer program. Both Producer and Consumer threads share same global queue.

```
import java.util.concurrent.*;
import java.util.Random;

public class ProducerConsumerWithES {
    public static void main(String args[]) {
        BlockingQueue<Integer> sharedQueue = new LinkedBlockingQueue<Integer>();

        ExecutorService pes = Executors.newFixedThreadPool(2);
        ExecutorService ces = Executors.newFixedThreadPool(2);

        pes.submit(new Producer(sharedQueue, 1));
        pes.submit(new Producer(sharedQueue, 2));
        ces.submit(new Consumer(sharedQueue, 1));
        ces.submit(new Consumer(sharedQueue, 2));

        pes.shutdown();
        ces.shutdown();
    }
}

/* Different producers produces a stream of integers continuously to a shared queue,
which is shared between all Producers and consumers */

class Producer implements Runnable {
    private final BlockingQueue<Integer> sharedQueue;
    private int threadNo;
    private Random random = new Random();
    public Producer(BlockingQueue<Integer> sharedQueue,int threadNo) {
        this.threadNo = threadNo;
        this.sharedQueue = sharedQueue;
    }
    @Override
    public void run() {
        // Producer produces a continuous stream of numbers for every 200 milli seconds
        while (true) {
            try {
                int number = random.nextInt(1000);
                System.out.println("Produced:" + number + ":by thread:"+ threadNo);
                sharedQueue.put(number);
                Thread.sleep(200);
            } catch (Exception err) {
                err.printStackTrace();
            }
        }
    }
}
/* Different consumers consume data from shared queue, which is shared by both producer and consumer
threads */
```

```
class Consumer implements Runnable {
    private final BlockingQueue<Integer> sharedQueue;
    private int threadNo;
    public Consumer (BlockingQueue<Integer> sharedQueue,int threadNo) {
        this.sharedQueue = sharedQueue;
        this.threadNo = threadNo;
    }
    @Override
    public void run() {
        // Consumer consumes numbers generated from Producer threads continuously
        while(true){
            try {
                int num = sharedQueue.take();
                System.out.println("Consumed: "+ num + ":by thread:"+threadNo);
            } catch (Exception err) {
               err.printStackTrace();
            }
        }
    }
}
```

output:

```
Produced:69:by thread:2
Produced:553:by thread:1
Consumed: 69:by thread:1
Consumed: 553:by thread:2
Produced:41:by thread:2
Produced:796:by thread:1
Consumed: 41:by thread:1
Consumed: 796:by thread:2
Produced:728:by thread:2
Consumed: 728:by thread:1
```

and so on .....

Explanation:

1. `sharedQueue`, which is a `LinkedBlockingQueue` is shared among all Producer and Consumer threads.
2. Producer threads produces one integer for every 200 milli seconds continuously and append it to `sharedQueue`
3. `Consumer` thread consumes integer from `sharedQueue` continuously.
4. This program is implemented with-out explicit **`synchronized`** or `Lock` constructs. BlockingQueue is the key to achieve it.

> BlockingQueue implementations are designed to be used primarily for producer-consumer queues.

> BlockingQueue implementations are thread-safe. All queuing methods achieve their effects atomically using internal locks or other forms of concurrency control.

Section 126.17: Add two `int` arrays using a Threadpool

A Threadpool has a Queue of tasks, of which each will be executed on one these Threads.

The following example shows how to add two `int` arrays using a Threadpool.

Version ≥ Java SE 8

```
int[] firstArray = { 2, 4, 6, 8 };
int[] secondArray = { 1, 3, 5, 7 };
int[] result = { 0, 0, 0, 0 };

ExecutorService pool = Executors.newCachedThreadPool();

// Setup the ThreadPool:
// for each element in the array, submit a worker to the pool that adds elements
for (int i = 0; i < result.length; i++) {
    final int worker = i;
    pool.submit(() -> result[worker] = firstArray[worker] + secondArray[worker] );
}

// Wait for all Workers to finish:
try {
    // execute all submitted tasks
    pool.shutdown();
    // waits until all workers finish, or the timeout ends
    pool.awaitTermination(12, TimeUnit.SECONDS);
}
catch (InterruptedException e) {
    pool.shutdownNow(); //kill thread
}

System.out.println(Arrays.toString(result));
```

Notes:

1. This example is purely illustrative. In practice, there won't be any speedup by using threads for a task this small. A slowdown is likely, since the overheads of task creation and scheduling will swamp the time taken to run a task.
2. If you were using Java 7 and earlier, you would use anonymous classes instead of lambdas to implement the tasks.

Section 126.18: Pausing Execution

`Thread.sleep` causes the current thread to suspend execution for a specified period. This is an efficient means of making processor time available to the other threads of an application or other applications that might be running on a computer system. There are two overloaded `sleep` methods in the Thread class.

One that specifies the sleep time to the millisecond

```
public static void sleep(long millis) throws InterruptedException
```

One that specifies the sleep time to the nanosecond

```
public static void sleep(long millis, int nanos)
```

Pausing Execution for 1 second

```
Thread.sleep(1000);
```

It is important to note that this is a hint to the operating system's kernel's scheduler. This may not necessarily be

precise, and some implementations do not even consider the nanosecond parameter (possibly rounding to the nearest millisecond).

It is recommended to enclose a call to `Thread.sleep` in try/catch and catch `InterruptedException`.

Section 126.19: Thread Interruption / Stopping Threads

Each Java Thread has an interrupt flag, which is initially false. Interrupting a thread, is essentially nothing more than setting that flag to true. The code running on that thread can check the flag on occasion and act upon it. The code can also ignore it completely. But why would each Thread have such a flag? After all, having a boolean flag on a thread is something we can just organize ourselves, if and when we need it. Well, there are methods that behave in a special way when the thread they're running on is interrupted. These methods are called blocking methods. These are methods that put the thread in the WAITING or TIMED_WAITING state. When a thread is in this state, interrupting it, will cause an InterruptedException to be thrown on the interrupted thread, rather than the interrupt flag being set to true, and the thread becomes RUNNABLE again. Code that invokes a blocking method is forced to deal with the InterruptedException, since it is a checked exception. So, and hence its name, an interrupt can have the effect of interrupting a WAIT, effectively ending it. Note that not all methods that are somehow waiting (e.g. blocking IO) respond to interruption in that way, as they don't put the thread in a waiting state. Lastly a thread that has its interrupt flag set, that enters a blocking method (i.e. tries to get into a waiting state), will immediately throw an InterruptedException and the interrupt flag will be cleared.

Other than these mechanics, Java does not assign any special semantic meaning to interruption. Code is free to interpret an interrupt any way it likes. But most often interruption is used to signal to a thread it should stop running at its earliest convenience. But, as should be clear from the above, it is up to the code on that thread to react to that interruption appropriately in order to stop running. Stopping a thread is a collaboration. When a thread is interrupted its running code can be several levels deep into the stacktrace. Most code doesn't call a blocking method, and finishes timely enough to not delay the stopping of the thread unduly. The code that should mostly be concerned with being responsive to interruption, is code that is in a loop handling tasks until there are none left, or until a flag is set signalling it to stop that loop. Loops that handle possibly infinite tasks (i.e. they keep running in principle) should check the interrupt flag in order to exit the loop. For finite loops the semantics may dictate that all tasks must be finished before ending, or it may be appropriate to leave some tasks unhandled. Code that calls blocking methods will be forced to deal with the InterruptedException. If at all semantically possible, it can simply propagate the InterruptedException and declare to throw it. As such it becomes a blocking method itself in regard to its callers. If it cannot propagate the exception, it should at the very least set the interrupted flag, so callers higher up the stack also know the thread was interrupted. In some cases the method needs to continue waiting regardless of the InterruptedException, in which case it must delay setting the interrupted flag until after it is done waiting, this may involve setting a local variable, which is to be checked prior to exiting the method to then interrupt its thread.

Examples :

Example of code that stops handling tasks upon interruption

```
class TaskHandler implements Runnable {

    private final BlockingQueue<Task> queue;

    TaskHandler(BlockingQueue<Task> queue) {
        this.queue = queue;
    }

    @Override
    public void run() {
        while (!Thread.currentThread().isInterrupted()) { // check for interrupt flag, exit loop
```

```
when interrupted
            try {
                Task task = queue.take(); // blocking call, responsive to interruption
                handle(task);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt(); // cannot throw InterruptedException (due to
Runnable interface restriction) so indicating interruption by setting the flag
            }
        }
    }

    private void handle(Task task) {
        // actual handling
    }
}
```

Example of code that delays setting the interrupt flag until completely done :

```
class MustFinishHandler implements Runnable {

    private final BlockingQueue<Task> queue;

    MustFinishHandler(BlockingQueue<Task> queue) {
        this.queue = queue;
    }

    @Override
    public void run() {
        boolean shouldInterrupt = false;

        while (true) {
            try {
                Task task = queue.take();
                if (task.isEndOfTasks()) {
                    if (shouldInterrupt) {
                        Thread.currentThread().interrupt();
                    }
                    return;
                }
                handle(task);
            } catch (InterruptedException e) {
                shouldInterrupt = true; // must finish, remember to set interrupt flag when we're
done
            }
        }
    }

    private void handle(Task task) {
        // actual handling
    }
}
```

Example of code that has a fixed list of tasks but may quit early when interrupted

```
class GetAsFarAsPossible implements Runnable {

    private final List<Task> tasks = new ArrayList<>();

    @Override
    public void run() {
        for (Task task : tasks) {
```

```
            if (Thread.currentThread().isInterrupted()) {
                return;
            }
            handle(task);
        }
    }

    private void handle(Task task) {
        // actual handling
    }
}
```

Chapter 127: Executor, ExecutorService and Thread pools

The Executor interface in Java provides a way of decoupling task submission from the mechanics of how each task will be run, including details of thread use, scheduling, etc. An Executor is normally used instead of explicitly creating threads. With Executors, developers won't have to significantly rewrite their code to be able to easily tune their program's task-execution policy.

Section 127.1: ThreadPoolExecutor

A common Executor used is the `ThreadPoolExecutor`, which takes care of Thread handling. You can configure the minimal amount of Threads the executor always has to maintain when there's not much to do (it's called core size) and a maximal Thread size to which the Pool can grow, if there is more work to do. Once the workload declines, the Pool slowly reduces the Thread count again until it reaches min size.

```
ThreadPoolExecutor pool = new ThreadPoolExecutor(
    1,                                      // keep at least one thread ready,
                                            // even if no Runnables are executed
    5,                                      // at most five Runnables/Threads
                                            // executed in parallel
    1, TimeUnit.MINUTES,                    // idle Threads terminated after one
                                            // minute, when min Pool size exceeded
    new ArrayBlockingQueue<Runnable>(10)); // outstanding Runnables are kept here

pool.execute(new Runnable() {
    @Override public void run() {
        //code to run
    }
});
```

Note If you configure the `ThreadPoolExecutor` with an *unbounded* queue, then the thread count will not exceed `corePoolSize` since new threads are only created if the queue is full:

ThreadPoolExecutor with all parameters:

```
ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime,
TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory,
RejectedExecutionHandler handler)
```

from JavaDoc

> If there are more than corePoolSize but less than maximumPoolSize threads running, a new thread will be created only if the queue is full.

Advantages:

1. BlockingQueue size can be controlled and out-of-memory scenarios can be avoided. Application performance won't be degraded with limited bounded queue size.
2. You can use existing or create new Rejection Handler policies.

    1. In the default ThreadPoolExecutor.AbortPolicy, the handler throws a runtime

RejectedExecutionException upon rejection.

2. In `ThreadPoolExecutor.CallerRunsPolicy`, the thread that invokes execute itself runs the task. This provides a simple feedback control mechanism that will slow down the rate that new tasks are submitted.
3. In `ThreadPoolExecutor.DiscardPolicy`, a task that cannot be executed is simply dropped.
4. In `ThreadPoolExecutor.DiscardOldestPolicy`, if the executor is not shut down, the task at the head of the work queue is dropped, and then execution is retried (which can fail again, causing this to be repeated.)

3. Custom `ThreadFactory` can be configured, which is useful :

    1. To set a more descriptive thread name
    2. To set thread daemon status
    3. To set thread priority

Here is a example of how to use ThreadPoolExecutor

Section 127.2: Retrieving value from computation - Callable

If your computation produces some return value which later is required, a simple Runnable task isn't sufficient. For such cases you can use `ExecutorService.submit(Callable<T>)` which returns a value after execution completes.

The Service will return a `Future` which you can use to retrieve the result of the task execution.

```
// Submit a callable for execution
ExecutorService pool = anExecutorService;
Future<Integer> future = pool.submit(new Callable<Integer>() {
    @Override public Integer call() {
        //do some computation
        return new Random().nextInt();
    }
});
// ... perform other tasks while future is executed in a different thread
```

When you need to get the result of the future, call `future.get()`

- Wait indefinitely for future to finish with a result.

```
try {
    // Blocks current thread until future is completed
    Integer result = future.get();
catch (InterruptedException || ExecutionException e) {
    // handle appropriately
}
```

- Wait for future to finish, but no longer than specified time.

```
try {
    // Blocks current thread for a maximum of 500 milliseconds.
    // If the future finishes before that, result is returned,
    // otherwise TimeoutException is thrown.
    Integer result = future.get(500, TimeUnit.MILLISECONDS);
catch (InterruptedException || ExecutionException || TimeoutException e) {
```

```
        // handle appropriately
    }
```

If the result of a scheduled or running task is no longer required, you can call `Future.cancel(boolean)` to cancel it.

- Calling `cancel(false)` will just remove the task from the queue of tasks to be run.
- Calling `cancel(true)` will *also* interrupt the task if it is currently running.

Section 127.3: submit() vs execute() exception handling differences

Generally execute() command is used for fire and forget calls (without need of analyzing the result) and submit() command is used for analyzing the result of Future object.

We should be aware of key difference of Exception Handling mechanisms between these two commands.

Exceptions from submit() are swallowed by framework if you did not catch them.

Code example to understand the difference:

Case 1: submit the Runnable with execute() command, which reports the Exception.

```
import java.util.concurrent.*;
import java.util.*;

public class ExecuteSubmitDemo {
    public ExecuteSubmitDemo() {
        System.out.println("creating service");
        ExecutorService service = Executors.newFixedThreadPool(2);
        //ExtendedExecutor service = new ExtendedExecutor();
        for (int i = 0; i < 2; i++){
            service.execute(new Runnable(){
                 public void run(){
                    int a = 4, b = 0;
                    System.out.println("a and b=" + a + ":" + b);
                    System.out.println("a/b:" + (a / b));
                    System.out.println("Thread Name in Runnable after divide by
zero:"+Thread.currentThread().getName());
                 }
            });
        }
        service.shutdown();
    }
    public static void main(String args[]){
        ExecuteSubmitDemo demo = new ExecuteSubmitDemo();
    }
}

class ExtendedExecutor extends ThreadPoolExecutor {

   public ExtendedExecutor() {
       super(1, 1, 60, TimeUnit.SECONDS, new ArrayBlockingQueue<Runnable>(100));
   }
   // ...
   protected void afterExecute(Runnable r, Throwable t) {
     super.afterExecute(r, t);
     if (t == null && r instanceof Future<?>) {
       try {
         Object result = ((Future<?>) r).get();
```

```
        } catch (CancellationException ce) {
            t = ce;
        } catch (ExecutionException ee) {
            t = ee.getCause();
        } catch (InterruptedException ie) {
            Thread.currentThread().interrupt(); // ignore/reset
        }
      }
      if (t != null)
        System.out.println(t);
    }
  }
```

output:

```
creating service
a and b=4:0
a and b=4:0
Exception in thread "pool-1-thread-1" Exception in thread "pool-1-thread-2"
java.lang.ArithmeticException: / by zero
at ExecuteSubmitDemo$1.run(ExecuteSubmitDemo.java:15)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
at java.lang.Thread.run(Thread.java:744)
java.lang.ArithmeticException: / by zero
at ExecuteSubmitDemo$1.run(ExecuteSubmitDemo.java:15)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
at java.lang.Thread.run(Thread.java:744)
```

Case 2: Replace execute() with submit() : `service.submit(new Runnable(){` In this case, Exceptions are swallowed by framework since run() method did not catch them explicitly.

output:

```
creating service
a and b=4:0
a and b=4:0
```

Case 3: Change the newFixedThreadPool to ExtendedExecutor

```
//ExecutorService service = Executors.newFixedThreadPool(2);
 ExtendedExecutor service = new ExtendedExecutor();
```

output:

```
creating service
a and b=4:0
java.lang.ArithmeticException: / by zero
a and b=4:0
java.lang.ArithmeticException: / by zero
```

I have demonstrated this example to cover two topics : Use your custom ThreadPoolExecutor and handle Exectpion with custom ThreadPoolExecutor.

Other simple solution to above problem : When you are using normal ExecutorService & submit command, get the Future object from submit() command call get() API on Future. Catch the three exceptions, which have been quoted in afterExecute method implementation. Advantage of custom ThreadPoolExecutor over this approach : You have to handle Exception handling mechanism in only one place - Custom ThreadPoolExecutor.

Section 127.4: Handle Rejected Execution

If

1. you try to submit tasks to a shutdown Executor or
2. the queue is saturated (only possible with bounded ones) and maximum number of Threads has been reached,

`RejectedExecutionHandler.rejectedExecution(Runnable, ThreadPoolExecutor)` will be called.

The default behavior is that you'll get a RejectedExecutionException thrown at the caller. But there are more predefined behaviors available:

- **ThreadPoolExecutor.AbortPolicy** (default, will throw REE)
- **ThreadPoolExecutor.CallerRunsPolicy** (executes task on caller's thread - *blocking it*)
- **ThreadPoolExecutor.DiscardPolicy** (silently discard task)
- **ThreadPoolExecutor.DiscardOldestPolicy** (silently discard **oldest** task in queue and retry execution of the new task)

You can set them using one of the ThreadPool constructors:

```
public ThreadPoolExecutor(int corePoolSize,
                      int maximumPoolSize,
                      long keepAliveTime,
                      TimeUnit unit,
                      BlockingQueue<Runnable> workQueue,
                      RejectedExecutionHandler handler) // <--

public ThreadPoolExecutor(int corePoolSize,
                      int maximumPoolSize,
                      long keepAliveTime,
                      TimeUnit unit,
                      BlockingQueue<Runnable> workQueue,
                      ThreadFactory threadFactory,
                      RejectedExecutionHandler handler) // <--
```

You can as well implement your own behavior by extending RejectedExecutionHandler interface:

```
void rejectedExecution(Runnable r, ThreadPoolExecutor executor)
```

Section 127.5: Fire and Forget - Runnable Tasks

Executors accept a `java.lang.Runnable` which contains (potentially computationally or otherwise long-running or heavy) code to be run in another Thread.

Usage would be:

```
Executor exec = anExecutor;
exec.execute(new Runnable() {
    @Override public void run() {
        //offloaded work, no need to get result back
```

```
    }
});
```

Note that with this executor, you have no means to get any computed value back.
With Java 8, one can utilize lambdas to shorten the code example.

Version ≥ Java SE 8

```
Executor exec = anExecutor;
exec.execute(() -> {
    //offloaded work, no need to get result back
});
```

Section 127.6: Use cases for different types of concurrency constructs

1. ExecutorService

   `ExecutorService executor = Executors.newFixedThreadPool(50);`

   It is simple and easy to use. It hides low level details of `ThreadPoolExecutor`.

   I prefer this one when number of `Callable`/`Runnable` tasks are small in number and piling of tasks in unbounded queue does not increase memory & degrade the performance of the system. If you have `CPU`/`Memory` constraints, I prefer to use `ThreadPoolExecutor` with capacity constraints & `RejectedExecutionHandler` to handle rejection of tasks.

2. CountDownLatch

   `CountDownLatch` will be initialized with a given count. This count is decremented by calls to the `countDown()` method. Threads waiting for this count to reach zero can call one of the `await()` methods. Calling `await()` blocks the thread until the count reaches zero. *This class enables a java thread to wait until other set of threads completes their tasks.*

   Use cases:

   1. Achieving Maximum Parallelism: Sometimes we want to start a number of threads at the same time to achieve maximum parallelism
   2. Wait N threads to completes before start execution
   3. Deadlock detection.

3. ThreadPoolExecutor : It provides more control. If application is constrained by number of pending Runnable/Callable tasks, you can use bounded queue by setting the max capacity. Once the queue reaches maximum capacity, you can define RejectionHandler. Java provides four types of `RejectedExecutionHandler` policies.

   1. `ThreadPoolExecutor.AbortPolicy`, the handler throws a runtime RejectedExecutionException upon rejection.
   2. ThreadPoolExecutor.CallerRunsPolicy`, the thread that invokes execute itself runs the task. This provides a simple feedback control mechanism that will slow down the rate that new tasks are submitted.

3. In `ThreadPoolExecutor.DiscardPolicy`, a task that cannot be executed is simply dropped.
4. `ThreadPoolExecutor.DiscardOldestPolicy`, if the executor is not shut down, the task at the head of the work queue is dropped, and then execution is retried (which can fail again, causing this to be repeated.)

If you want to simulate `CountDownLatch` behaviour, you can use `invokeAll()` method.

4. One more mechanism you did not quote is ForkJoinPool

   The `ForkJoinPool` was added to Java in Java 7. The `ForkJoinPool` is similar to the Java `ExecutorService` but with one difference. The `ForkJoinPool` makes it easy for tasks to split their work up into smaller tasks which are then submitted to the `ForkJoinPool` too. Task stealing happens in `ForkJoinPool` when free worker threads steal tasks from busy worker thread queue.

   Java 8 has introduced one more API in ExecutorService to create work stealing pool. You don't have to create `RecursiveTask` and `RecursiveAction` but still can use `ForkJoinPool`.

   ```
   public static ExecutorService newWorkStealingPool()
   ```

   > Creates a work-stealing thread pool using all available processors as its target parallelism level.

   By default, it will take number of CPU cores as parameter.

All these four mechanism are complimentary to each other. Depending on level of granularity you want to control, you have to chose right ones.

Section 127.7: Wait for completion of all tasks in ExecutorService

Let's have a look at various options to wait for completion of tasks submitted to Executor

1. ExecutorService `invokeAll()`

   > Executes the given tasks, returning a list of Futures holding their status and results when everything is completed.

Example:

```
import java.util.concurrent.*;
import java.util.*;

public class InvokeAllDemo{
    public InvokeAllDemo(){
        System.out.println("creating service");
        ExecutorService service =
Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());

        List<MyCallable> futureList = new ArrayList<MyCallable>();
```

```
        for (int i = 0; i < 10; i++){
            MyCallable myCallable = new MyCallable((long)i);
            futureList.add(myCallable);
        }
        System.out.println("Start");
        try{
            List<Future<Long>> futures = service.invokeAll(futureList);
        } catch(Exception err){
            err.printStackTrace();
        }
        System.out.println("Completed");
        service.shutdown();
    }
    public static void main(String args[]){
        InvokeAllDemo demo = new InvokeAllDemo();
    }
    class MyCallable implements Callable<Long>{
        Long id = 0L;
        public MyCallable(Long val){
            this.id = val;
        }
        public Long call(){
            // Add your business logic
            return id;
        }
    }
}
```

2. CountDownLatch

> A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.

> A **CountDownLatch** is initialized with a given count. The await methods block until the current count reaches zero due to invocations of the `countDown()` method, after which all waiting threads are released and any subsequent invocations of await return immediately. This is a one-shot phenomenon -- the count cannot be reset. If you need a version that resets the count, consider using a **CyclicBarrier**.

3. ForkJoinPool or `newWorkStealingPool()` in Executors
4. Iterate through all `Future` objects created after submitting to `ExecutorService`
5. Recommended way of shutdown from oracle documentation page of ExecutorService:

```
void shutdownAndAwaitTermination(ExecutorService pool) {
    pool.shutdown(); // Disable new tasks from being submitted
    try {
      // Wait a while for existing tasks to terminate
      if (!pool.awaitTermination(60, TimeUnit.SECONDS)) {
        pool.shutdownNow(); // Cancel currently executing tasks
        // Wait a while for tasks to respond to being cancelled
        if (!pool.awaitTermination(60, TimeUnit.SECONDS))
            System.err.println("Pool did not terminate");
      }
```

```
    } catch (InterruptedException ie) {
      // (Re-)Cancel if current thread also interrupted
      pool.shutdownNow();
      // Preserve interrupt status
      Thread.currentThread().interrupt();
    }
```

`shutdown()`: Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.

`shutdownNow()`:Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution.

In above example, if your tasks are taking more time to complete, you can change if condition to while condition

Replace

```
if (!pool.awaitTermination(60, TimeUnit.SECONDS))
```

with

```
while(!pool.awaitTermination(60, TimeUnit.SECONDS)) {
  Thread.sleep(60000);
```

}

Section 127.8: Use cases for different types of ExecutorService

Executors returns different type of ThreadPools catering to specific need.

1. `public static ExecutorService newSingleThreadExecutor()`

   > Creates an Executor that uses a single worker thread operating off an unbounded queue

   There is a difference between `newFixedThreadPool(1)` and `newSingleThreadExecutor()` as the java doc says for the latter:

   > Unlike the otherwise equivalent newFixedThreadPool(1) the returned executor is guaranteed not to be reconfigurable to use additional threads.

   Which means that a `newFixedThreadPool` can be reconfigured later in the program by: `((ThreadPoolExecutor) fixedThreadPool).setMaximumPoolSize(10)` This is not possible for `newSingleThreadExecutor`

   Use cases:

   1. You want to execute the submitted tasks in a sequence.
   2. You need only one Thread to handle all your request

Cons:

1. Unbounded queue is harmful

2. `public static ExecutorService newFixedThreadPool(int nThreads)`

> Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue. At any point, at most nThreads threads will be active processing tasks. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available

Use cases:

1. Effective use of available cores. Configure `nThreads` as `Runtime.getRuntime().availableProcessors()`
2. When you decide that number of thread should not exceed a number in the thread pool

Cons:

1. Unbounded queue is harmful.

3. `public static ExecutorService newCachedThreadPool()`

> Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available

Use cases:

1. For short-lived asynchronous tasks

Cons:

1. Unbounded queue is harmful.
2. Each new task will create a new thread if all existing threads are busy. If the task is taking long duration, more number of threads will be created,which will degrade the performance of the system. Alternative in this case: `newFixedThreadPool`

4. `public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)`

> Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically.

Use cases:

1. Handling recurring events with delays, which will happen in future at certain interval of times

Cons:

1. Unbounded queue is harmful.

5.`public static ExecutorService newWorkStealingPool()`

> Creates a work-stealing thread pool using all available processors as its target parallelism level

Use cases:

1. For divide and conquer type of problems.
2. Effective use of idle threads. Idle threads steals tasks from busy threads.

Cons:

1. Unbounded queue size is harmful.

You can see one common drawbacks in all these ExecutorService : unbounded queue. This will be addressed with ThreadPoolExecutor

```
ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime,
TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory,
RejectedExecutionHandler handler)
```

With `ThreadPoolExecutor`, you can

1. Control Thread pool size dynamically
2. Set the capacity for `BlockingQueue`
3. Define `RejectionExecutionHander` when queue is full
4. `CustomThreadFactory` to add some additional functionality during Thread creation (`public Thread newThread(Runnable r)`

Section 127.9: Scheduling tasks to run at a fixed time, after a delay or repeatedly

The `ScheduledExecutorService` class provides a methods for scheduling single or repeated tasks in a number of ways. The following code sample assume that `pool` has been declared and initialized as follows:

```
ScheduledExecutorService pool = Executors.newScheduledThreadPool(2);
```

In addition to the normal `ExecutorService` methods, the `ScheduledExecutorService` API adds 4 methods that schedule tasks and return `ScheduledFuture` objects. The latter can be used to retrieve results (in some cases) and cancel tasks.

Starting a task after a fixed delay

The following example schedules a task to start after ten minutes.

```
ScheduledFuture<Integer> future = pool.schedule(new Callable<>() {
        @Override public Integer call() {
            // do something
            return 42;
        }
    },
    10, TimeUnit.MINUTES);
```

Starting tasks at a fixed rate

The following example schedules a task to start after ten minutes, and then repeatedly at a rate of once every one minute.

```
ScheduledFuture<?> future = pool.scheduleAtFixedRate(new Runnable() {
        @Override public void run() {
            // do something
        }
    },
    10, 1, TimeUnit.MINUTES);
```

Task execution will continue according to the schedule until the `pool` is shut down, the `future` is canceled, or one of the tasks encounters an exception.

It is guaranteed that the tasks scheduled by a given `scheduledAtFixedRate` call will not overlap in time. If a task takes longer than the prescribed period, then the next and subsequent task executions may start late.

Starting tasks with a fixed delay

The following example schedules a task to start after ten minutes, and then repeatedly with a delay of one minute between one task ending and the next one starting.

```
ScheduledFuture<?> future = pool.scheduleWithFixedDelay(new Runnable() {
        @Override public void run() {
            // do something
        }
    },
    10, 1, TimeUnit.MINUTES);
```

Task execution will continue according to the schedule until the `pool` is shut down, the `future` is canceled, or one of the tasks encounters an exception.

Section 127.10: Using Thread Pools

Thread Pools are used mostly calling methods in `ExecutorService`.

The following methods can be used to submit work for execution:

Method	Description
`submit`	Executes a the submitted work and return a future which can be used to get the result
`execute`	Execute the task sometime in the future without getting any return value
`invokeAll`	Execute a list of tasks and return a list of Futures
`invokeAny`	Executes all the but return only the result of one that has been successful (without exceptions)

Once you are done with the Thread Pool you can call `shutdown()` to terminate the Thread Pool. This executes all pending tasks. To wait for all tasks to execute you can can loop around `awaitTermination` or `isShutdown()`.

Chapter 128: ThreadLocal

Section 128.1: Basic ThreadLocal usage

Java `ThreadLocal` is used to create thread local variables. It is known that threads of an Object share it's variables, so the variable is not thread safe. We can use synchronization for thread safety but if we want to avoid synchronization,ThreadLocal allows us to create variables which are local to the thread, i.e. only that thread can read or write to those variables, so the other threads executing the same piece of code will not be able to access each others ThreadLocal variables.

This can be usedwe can use `ThreadLocal` variables. in situations where you have a thread pool like for example in a web service. For example, Creating a `SimpleDateFormat` object every time for every request is time consuming and a Static one cannot be created as `SimpleDateFormat` is not thread safe, so we can create a ThreadLocal so that we can perform thread safe operations without the overhead of creating `SimpleDateFormat` every time.

The below piece of code shows how it can be used:

Every thread has it's own `ThreadLocal` variable and they can use it's `get()` and `set()` methods to get the default value or change it's value local to Thread.

`ThreadLocal` instances are typically private static fields in classes that wish to associate state with a thread.

Here is a small example showing use of ThreadLocal in java program and proving that every thread has it's own copy of `ThreadLocal` variable.

```
package com.examples.threads;

import java.text.SimpleDateFormat;
import java.util.Random;

public class ThreadLocalExample implements Runnable{

    // SimpleDateFormat is not thread-safe, so give one to each thread
 // SimpleDateFormat is not thread-safe, so give one to each thread
    private static final ThreadLocal<SimpleDateFormat> formatter = new
ThreadLocal<SimpleDateFormat>(){
        @Override
        protected SimpleDateFormat initialValue()
        {
            return new SimpleDateFormat("yyyyMMdd HHmm");
        }
    };

    public static void main(String[] args) throws InterruptedException {
        ThreadLocalExample obj = new ThreadLocalExample();
        for(int i=0 ; i<10; i++){
            Thread t = new Thread(obj, ""+i);
            Thread.sleep(new Random().nextInt(1000));
            t.start();
        }
    }

    @Override
    public void run() {
        System.out.println("Thread Name= "+Thread.currentThread().getName()+" default Formatter =
"+formatter.get().toPattern());
        try {
```

```
            Thread.sleep(new Random().nextInt(1000));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        formatter.set(new SimpleDateFormat());

        System.out.println("Thread Name= "+Thread.currentThread().getName()+" formatter =
"+formatter.get().toPattern());
    }

}
```

Output:

```
Thread Name= 0 default Formatter = yyyyMMdd HHmm

Thread Name= 1 default Formatter = yyyyMMdd HHmm

Thread Name= 0 formatter = M/d/yy h:mm a

Thread Name= 2 default Formatter = yyyyMMdd HHmm

Thread Name= 1 formatter = M/d/yy h:mm a

Thread Name= 3 default Formatter = yyyyMMdd HHmm

Thread Name= 4 default Formatter = yyyyMMdd HHmm

Thread Name= 4 formatter = M/d/yy h:mm a

Thread Name= 5 default Formatter = yyyyMMdd HHmm

Thread Name= 2 formatter = M/d/yy h:mm a

Thread Name= 3 formatter = M/d/yy h:mm a

Thread Name= 6 default Formatter = yyyyMMdd HHmm

Thread Name= 5 formatter = M/d/yy h:mm a

Thread Name= 6 formatter = M/d/yy h:mm a

Thread Name= 7 default Formatter = yyyyMMdd HHmm

Thread Name= 8 default Formatter = yyyyMMdd HHmm

Thread Name= 8 formatter = M/d/yy h:mm a

Thread Name= 7 formatter = M/d/yy h:mm a

Thread Name= 9 default Formatter = yyyyMMdd HHmm

Thread Name= 9 formatter = M/d/yy h:mm a
```

As we can see from the output that Thread-0 has changed the value of formatter but still thread-2 default formatter is same as the initialized value.

Section 128.2: ThreadLocal Java 8 functional initialization

```
public static class ThreadLocalExample
{
```

```
    private static final ThreadLocal<SimpleDateFormat> format =
        ThreadLocal.withInitial(() -> new SimpleDateFormat("yyyyMMdd_HHmm"));

    public String formatDate(Date date)
    {
        return format.get().format(date);
    }
}
```

Section 128.3: Multiple threads with one shared object

In this example we have only one object but it is shared between/executed on different threads. Ordinary usage of fields to save state would not be possible because the other thread would see that too (or probably not see).

```
public class Test {
    public static void main(String[] args) {
        Foo foo = new Foo();
        new Thread(foo, "Thread 1").start();
        new Thread(foo, "Thread 2").start();
    }
}
```

In Foo we count starting from zero. Instead of saving the state to a field we store our current number in the ThreadLocal object which is statically accessible. Note that the synchronization in this example is not related to the usage of ThreadLocal but rather ensures a better console output.

```
public class Foo implements Runnable {
    private static final int ITERATIONS = 10;
    private static final ThreadLocal<Integer> threadLocal = new ThreadLocal<Integer>() {
        @Override
        protected Integer initialValue() {
            return 0;
        }
    };

    @Override
    public void run() {
        for (int i = 0; i < ITERATIONS; i++) {
            synchronized (threadLocal) {
                //Although accessing a static field, we get our own (previously saved) value.
                int value = threadLocal.get();
                System.out.println(Thread.currentThread().getName() + ": " + value);

                //Update our own variable
                threadLocal.set(value + 1);

                try {
                    threadLocal.notifyAll();
                    if (i < ITERATIONS - 1) {
                        threadLocal.wait();
                    }
                } catch (InterruptedException ex) {
                }
            }
        }
    }
}
```

From the output we can see that each thread counts for itself and does not use the value of the other one:

```
Thread 1: 0
Thread 2: 0
Thread 1: 1
Thread 2: 1
Thread 1: 2
Thread 2: 2
Thread 1: 3
Thread 2: 3
Thread 1: 4
Thread 2: 4
Thread 1: 5
Thread 2: 5
Thread 1: 6
Thread 2: 6
Thread 1: 7
Thread 2: 7
Thread 1: 8
Thread 2: 8
Thread 1: 9
Thread 2: 9
```

Chapter 129: Using ThreadPoolExecutor in MultiThreaded applications.

When creating a performant and data-driven application, it can be very helpful to complete time-intensive tasks in an asynchronous manner and to have multiple tasks running concurrently. This topic will introduce the concept of using ThreadPoolExecutors to complete multiple ansynchronous tasks concurrently.

Section 129.1: Performing Asynchronous Tasks Where No Return Value Is Needed Using a Runnable Class Instance

Some applications may want to create so-called "Fire & Forget" tasks which can be periodically triggered and do not need to return any type of value returned upon completion of the assigned task (for example, purging old temp files, rotating logs, autosaving state).

In this example, we will create two classes: One which implements the Runnable interface, and one which contains a main() method.

AsyncMaintenanceTaskCompleter.java

```
import lombok.extern.java.Log;

import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.TimeUnit;

@Log
public class AsyncMaintenanceTaskCompleter implements Runnable {
    private int taskNumber;

    public AsyncMaintenanceTaskCompleter(int taskNumber) {
        this.taskNumber = taskNumber;
    }

    public void run() {
        int timeout = ThreadLocalRandom.current().nextInt(1, 20);
        try {
            log.info(String.format("Task %d is sleeping for %d seconds", taskNumber, timeout));
            TimeUnit.SECONDS.sleep(timeout);
            log.info(String.format("Task %d is done sleeping", taskNumber));

        } catch (InterruptedException e) {
            log.warning(e.getMessage());
        }
    }
}
```

AsyncExample1

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class AsyncExample1 {
    public static void main(String[] args){
        ExecutorService executorService = Executors.newCachedThreadPool();
        for(int i = 0; i < 10; i++){
            executorService.execute(new AsyncMaintenanceTaskCompleter(i));
        }
```

```
        executorService.shutdown();
    }
}
```

Running AsyncExample1.main() resulted in the following output:

```
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 8 is sleeping for 18 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 6 is sleeping for 4 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 2 is sleeping for 6 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 3 is sleeping for 4 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 9 is sleeping for 14 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 4 is sleeping for 9 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 5 is sleeping for 10 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 0 is sleeping for 7 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 1 is sleeping for 9 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 7 is sleeping for 8 seconds
Dec 28, 2016 2:21:07 PM AsyncMaintenanceTaskCompleter run
INFO: Task 6 is done sleeping
Dec 28, 2016 2:21:07 PM AsyncMaintenanceTaskCompleter run
INFO: Task 3 is done sleeping
Dec 28, 2016 2:21:09 PM AsyncMaintenanceTaskCompleter run
INFO: Task 2 is done sleeping
Dec 28, 2016 2:21:10 PM AsyncMaintenanceTaskCompleter run
INFO: Task 0 is done sleeping
Dec 28, 2016 2:21:11 PM AsyncMaintenanceTaskCompleter run
INFO: Task 7 is done sleeping
Dec 28, 2016 2:21:12 PM AsyncMaintenanceTaskCompleter run
INFO: Task 4 is done sleeping
Dec 28, 2016 2:21:12 PM AsyncMaintenanceTaskCompleter run
INFO: Task 1 is done sleeping
Dec 28, 2016 2:21:13 PM AsyncMaintenanceTaskCompleter run
INFO: Task 5 is done sleeping
Dec 28, 2016 2:21:17 PM AsyncMaintenanceTaskCompleter run
INFO: Task 9 is done sleeping
Dec 28, 2016 2:21:21 PM AsyncMaintenanceTaskCompleter run
INFO: Task 8 is done sleeping

Process finished with exit code 0
```

Observations of Note: There are several things to note in the output above,

1. The tasks did not execute in a predictable order.
2. Since each task was sleeping for a (pseudo)random amount of time, they did not necessarily complete in the order in which they were invoked.

Section 129.2: Performing Asynchronous Tasks Where a Return Value Is Needed Using a Callable Class Instance

It is often necessary to execute a long-running task and use the result of that task once it has completed.

In this example, we will create two classes: One which implements the Callable<T> interface (where T is the type we wish to return), and one which contains a main() method.

AsyncValueTypeTaskCompleter.java

```
import lombok.extern.java.Log;

import java.util.concurrent.Callable;
import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.TimeUnit;

@Log
public class AsyncValueTypeTaskCompleter implements Callable<Integer> {
    private int taskNumber;

    public AsyncValueTypeTaskCompleter(int taskNumber) {
        this.taskNumber = taskNumber;
    }

    @Override
    public Integer call() throws Exception {
        int timeout = ThreadLocalRandom.current().nextInt(1, 20);
        try {
            log.info(String.format("Task %d is sleeping", taskNumber));
            TimeUnit.SECONDS.sleep(timeout);
            log.info(String.format("Task %d is done sleeping", taskNumber));

        } catch (InterruptedException e) {
            log.warning(e.getMessage());
        }
        return timeout;
    }
}
```

AsyncExample2.java

```
import lombok.extern.java.Log;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

@Log
public class AsyncExample2 {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newCachedThreadPool();
        List<Future<Integer>> futures = new ArrayList<>();
        for (int i = 0; i < 10; i++){
            Future<Integer> submittedFuture = executorService.submit(new
AsyncValueTypeTaskCompleter(i));
            futures.add(submittedFuture);
        }
        executorService.shutdown();
        while(!futures.isEmpty()){
            for(int j = 0; j < futures.size(); j++){
                Future<Integer> f = futures.get(j);
                if(f.isDone()){
                    try {
```

```
                        int timeout = f.get();
                        log.info(String.format("A task just completed after sleeping for %d
seconds", timeout));
                        futures.remove(f);
                    } catch (InterruptedException | ExecutionException e) {
                        log.warning(e.getMessage());
                    }
                }
            }
        }
    }
}
```

Running AsyncExample2.main() resulted in the following output:

```
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 7 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 8 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 2 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 1 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 4 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 9 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 0 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 6 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 5 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 3 is sleeping
Dec 28, 2016 3:07:16 PM AsyncValueTypeTaskCompleter call
INFO: Task 8 is done sleeping
Dec 28, 2016 3:07:16 PM AsyncExample2 main
INFO: A task just completed after sleeping for 1 seconds
Dec 28, 2016 3:07:17 PM AsyncValueTypeTaskCompleter call
INFO: Task 2 is done sleeping
Dec 28, 2016 3:07:17 PM AsyncExample2 main
INFO: A task just completed after sleeping for 2 seconds
Dec 28, 2016 3:07:17 PM AsyncValueTypeTaskCompleter call
INFO: Task 9 is done sleeping
Dec 28, 2016 3:07:17 PM AsyncExample2 main
INFO: A task just completed after sleeping for 2 seconds
Dec 28, 2016 3:07:19 PM AsyncValueTypeTaskCompleter call
INFO: Task 3 is done sleeping
Dec 28, 2016 3:07:19 PM AsyncExample2 main
INFO: A task just completed after sleeping for 4 seconds
Dec 28, 2016 3:07:20 PM AsyncValueTypeTaskCompleter call
INFO: Task 0 is done sleeping
Dec 28, 2016 3:07:20 PM AsyncExample2 main
INFO: A task just completed after sleeping for 5 seconds
Dec 28, 2016 3:07:21 PM AsyncValueTypeTaskCompleter call
INFO: Task 5 is done sleeping
Dec 28, 2016 3:07:21 PM AsyncExample2 main
INFO: A task just completed after sleeping for 6 seconds
Dec 28, 2016 3:07:25 PM AsyncValueTypeTaskCompleter call
INFO: Task 1 is done sleeping
Dec 28, 2016 3:07:25 PM AsyncExample2 main
```

```
INFO: A task just completed after sleeping for 10 seconds
Dec 28, 2016 3:07:27 PM AsyncValueTypeTaskCompleter call
INFO: Task 6 is done sleeping
Dec 28, 2016 3:07:27 PM AsyncExample2 main
INFO: A task just completed after sleeping for 12 seconds
Dec 28, 2016 3:07:29 PM AsyncValueTypeTaskCompleter call
INFO: Task 7 is done sleeping
Dec 28, 2016 3:07:29 PM AsyncExample2 main
INFO: A task just completed after sleeping for 14 seconds
Dec 28, 2016 3:07:31 PM AsyncValueTypeTaskCompleter call
INFO: Task 4 is done sleeping
Dec 28, 2016 3:07:31 PM AsyncExample2 main
INFO: A task just completed after sleeping for 16 seconds
```

Observations of Note:

There are several things to note in the output above,

1. Each call to ExecutorService.submit() returned an instance of Future, which was stored in a list for later use
2. Future contains a method called isDone() which can be used to check whether our task has been completed before attempting to check it's return value. Calling the Future.get() method on a Future that is not yet done will block the current thread until the task is complete, potentially negating many benefits gained from performing the task Asynchronously.
3. The executorService.shutdown() method was called prior to checking the return values of the Future objects. This is not required, but was done in this way to show that it is possible. The executorService.shutdown() method does not prevent the completion of tasks which have already been submitted to the ExecutorService, but rather prevents new tasks from being added to the Queue.

Section 129.3: Defining Asynchronous Tasks Inline using Lambdas

While good software design often maximizes code reusability, sometimes it can be useful to define asynchronous tasks inline in your code via Lambda expressions to maximize code readability.

In this example, we will create a single class which contains a main() method. Inside this method, we will use Lambda expressions to create and execute instances of Callable and Runnable<T>.

AsyncExample3.java

```
import lombok.extern.java.Log;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.*;

@Log
public class AsyncExample3 {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newCachedThreadPool();
        List<Future<Integer>> futures = new ArrayList<>();
        for(int i = 0; i < 5; i++){
            final int index = i;
            executorService.execute(() -> {
                int timeout = getTimeout();
                log.info(String.format("Runnable %d has been submitted and will sleep for %d
seconds", index, timeout));
                try {
```

```
                    TimeUnit.SECONDS.sleep(timeout);
                } catch (InterruptedException e) {
                    log.warning(e.getMessage());
                }
                log.info(String.format("Runnable %d has finished sleeping", index));
            });
            Future<Integer> submittedFuture = executorService.submit(() -> {
                int timeout = getTimeout();
                log.info(String.format("Callable %d will begin sleeping", index));
                try {
                    TimeUnit.SECONDS.sleep(timeout);
                } catch (InterruptedException e) {
                    log.warning(e.getMessage());
                }
                log.info(String.format("Callable %d is done sleeping", index));
                return timeout;
            });
            futures.add(submittedFuture);
        }
        executorService.shutdown();
        while(!futures.isEmpty()){
            for(int j = 0; j < futures.size(); j++){
                Future<Integer> f = futures.get(j);
                if(f.isDone()){
                    try {
                        int timeout = f.get();
                        log.info(String.format("A task just completed after sleeping for %d
seconds", timeout));
                        futures.remove(f);
                    } catch (InterruptedException | ExecutionException e) {
                        log.warning(e.getMessage());
                    }
                }
            }
        }
    }

    public static int getTimeout(){
        return ThreadLocalRandom.current().nextInt(1, 20);
    }
}
```

Observations of Note:

There are several things to note in the output above,

1. Lambda expressions have access to variables and methods which are available to the scope in which they are defined, but all variables must be final (or effectively final) for use inside a lambda expression.
2. We do not have to specify whether our Lambda expression is a Callable or a Runnable<T> explicitly, the return type is inferred automatically by the return type.

Chapter 130: Common Java Pitfalls

This topic outlines some of the common mistakes made by beginners in Java.

This includes any common mistakes in use of the Java language or understanding of the run-time environment.

Mistakes associated with specific APIs can be described in topics specific to those APIs. Strings are a special case; they're covered in the Java Language Specification. Details other than common mistakes can be described in this topic on Strings.

Section 130.1: Pitfall: using == to compare primitive wrappers objects such as Integer

(This pitfall applies equally to all primitive wrapper types, but we will illustrate it for `Integer` and **`int`**.)

When working with `Integer` objects, it is tempting to use `==` to compare values, because that is what you would do with **`int`** values. And in some cases this will seem to work:

```
Integer int1_1 = Integer.valueOf("1");
Integer int1_2 = Integer.valueOf(1);

System.out.println("int1_1 == int1_2: " + (int1_1 == int1_2));          // true
System.out.println("int1_1 equals int1_2: " + int1_1.equals(int1_2));   // true
```

Here we created two `Integer` objects with the value 1 and compare them (In this case we created one from a `String` and one from an **`int`** literal. There are other alternatives). Also, we observe that the two comparison methods (`==` and `equals`) both yield **`true`**.

This behavior changes when we choose different values:

```
Integer int2_1 = Integer.valueOf("1000");
Integer int2_2 = Integer.valueOf(1000);

System.out.println("int2_1 == int2_2: " + (int2_1 == int2_2));          // false
System.out.println("int2_1 equals int2_2: " + int2_1.equals(int2_2));   // true
```

In this case, only the `equals` comparison yields the correct result.

The reason for this difference in behavior is, that the JVM maintains a cache of `Integer` objects for the range -128 to 127. (The upper value can be overridden with the system property "java.lang.Integer.IntegerCache.high" or the JVM argument "-XX:AutoBoxCacheMax=size"). For values in this range, the `Integer.valueOf()` will return the cached value rather than creating a new one.

Thus, in the first example the `Integer.valueOf(1)` and `Integer.valueOf("1")` calls returned the same cached `Integer` instance. By contrast, in the second example the `Integer.valueOf(1000)` and `Integer.valueOf("1000")` both created and returned new `Integer` objects.

The `==` operator for reference types tests for reference equality (i.e. the same object). Therefore, in the first example `int1_1 == int1_2` is **`true`** because the references are the same. In the second example `int2_1 == int2_2` is false because the references are different.

Section 130.2: Pitfall: using == to compare strings

A common mistake for Java beginners is to use the `==` operator to test if two strings are equal. For example:

```
public class Hello {
    public static void main(String[] args) {
        if (args.length > 0) {
            if (args[0] == "hello") {
                System.out.println("Hello back to you");
            } else {
                System.out.println("Are you feeling grumpy today?");
            }
        }
    }
}
```

The above program is supposed to test the first command line argument and print different messages when it and isn't the word "hello". But the problem is that it won't work. That program will output "Are you feeling grumpy today?" no matter what the first command line argument is.

In this particular case the `String` "hello" is put in the string pool while the `String` args[0] resides on the heap. This means there are two objects representing the same literal, each with its reference. Since `==` tests for references, not actual equality, the comparison will yield a false most of the times. This doesn't mean that it will always do so.

When you use `==` to test strings, what you are actually testing is if two `String` objects are the same Java object. Unfortunately, that is not what string equality means in Java. In fact, the correct way to test strings is to use the `equals(Object)` method. For a pair of strings, we usually want to test if they consist of the same characters in the same order.

```
public class Hello2 {
    public static void main(String[] args) {
        if (args.length > 0) {
            if (args[0].equals("hello")) {
                System.out.println("Hello back to you");
            } else {
                System.out.println("Are you feeling grumpy today?");
            }
        }
    }
}
```

But it actually gets worse. The problem is that `==` *will* give the expected answer in some circumstances. For example

```
public class Test1 {
    public static void main(String[] args) {
        String s1 = "hello";
        String s2 = "hello";
        if (s1 == s2) {
            System.out.println("same");
        } else {
            System.out.println("different");
        }
    }
}
```

Interestingly, this will print "same", even though we are testing the strings the wrong way. Why is that? Because the Java Language Specification (Section 3.10.5: String Literals) stipulates that any two string >>literals<< consisting of the same characters will actually be represented by the same Java object. Hence, the == test will give true for equal literals. (The string literals are "interned" and added to a shared "string pool" when your code is loaded, but that is actually an implementation detail.)

To add to the confusion, the Java Language Specification also stipulates that when you have a compile-time constant expression that concatenates two string literals, that is equivalent to a single literal. Thus:

```
    public class Test1 {
    public static void main(String[] args) {
        String s1 = "hello";
        String s2 = "hel" + "lo";
        String s3 = " mum";
        if (s1 == s2) {
            System.out.println("1. same");
        } else {
            System.out.println("1. different");
        }
        if (s1 + s3 == "hello mum") {
            System.out.println("2. same");
        } else {
            System.out.println("2. different");
        }
    }
}
```

This will output "1. same" and "2. different". In the first case, the `+` expression is evaluated at compile time and we compare one `String` object with itself. In the second case, it is evaluated at run time and we compare two different `String` objects

In summary, using `==` to test strings in Java is almost always incorrect, but it is not guaranteed to give the wrong answer.

Section 130.3: Pitfall: forgetting to free resources

Every time a program opens a resource, such as a file or network connection, it is important to free the resource once you are done using it. Similar caution should be taken if any exception were to be thrown during operations on such resources. One could argue that the `FileInputStream` has a finalizer that invokes the `close()` method on a garbage collection event; however, since we can't be sure when a garbage collection cycle will start, the input stream can consume computer resources for an indefinite period of time. The resource must be closed in a **`finally`** section of a try-catch block:

Version < Java SE 7

```
private static void printFileJava6() throws IOException {
    FileInputStream input;
    try {
        input = new FileInputStream("file.txt");
        int data = input.read();
        while (data != -1){
            System.out.print((char) data);
            data = input.read();
        }
    } finally {
        if (input != null) {
            input.close();
        }
    }
}
```

Since Java 7 there is a really useful and neat statement introduced in Java 7 particularly for this case, called try-with-resources:

Version ≥ Java SE 7

```
private static void printFileJava7() throws IOException {
    try (FileInputStream input = new FileInputStream("file.txt")) {
        int data = input.read();
        while (data != -1){
            System.out.print((char) data);
            data = input.read();
        }
    }
}
```

The *try-with-resources* statement can be used with any object that implements the `Closeable` or `AutoCloseable` interface. It ensures that each resource is closed by the end of the statement. The difference between the two interfaces is, that the `close()` method of `Closeable` throws an `IOException` which has to be handled in some way.

In cases where the resource has already been opened but should be safely closed after use, one can assign it to a local variable inside the try-with-resources

Version ≥ Java SE 7

```
private static void printFileJava7(InputStream extResource) throws IOException {
    try (InputStream input = extResource) {
        ... //access resource
    }
}
```

The local resource variable created in the try-with-resources constructor is effectively final.

Section 130.4: Pitfall: testing a file before attempting to open it

Some people recommend that you should apply various tests to a file before attempting to open it either to provide better diagnostics or avoid dealing with exceptions. For example, this method attempts to check if `path` corresponds to a readable file:

```
public static File getValidatedFile(String path) throws IOException {
    File f = new File(path);
    if (!f.exists()) throw new IOException("Error: not found: " + path);
    if (!f.isFile()) throw new IOException("Error: Is a directory: " + path);
    if (!f.canRead()) throw new IOException("Error: cannot read file: " + path);
    return f;
}
```

You might use the above method like this:

```
File f = null;
try {
    f = getValidatedFile("somefile");
} catch (IOException ex) {
    System.err.println(ex.getMessage());
    return;
}
try (InputStream is = new FileInputStream(file)) {
    // Read data etc.
}
```

The first problem is in the signature for `FileInputStream(File)` because the compiler will still insist we catch `IOException` here, or further up the stack.

The second problem is that checks performed by `getValidatedFile` do not guarantee that the `FileInputStream` will succeed.

- Race conditions: another thread or a separate process could rename the file, delete the file, or remove read access after the `getValidatedFile` returns. That would lead to a "plain" `IOException` without the custom message.
- There are edge cases not covered by those tests. For example, on a system with SELinux in "enforcing" mode, an attempt to read a file can fail despite `canRead()` returning **`true`**.

The third problem is that the tests are inefficient. For example, the `exists`, `isFile` and `canRead` calls will each make a syscall to perform the required check. Another syscall is then made to open the file, which repeats the same checks behind the scenes.

In short, methods like `getValidatedFile` are misguided. It is better to simply attempt to open the file and handle the exception:

```
try (InputStream is = new FileInputStream("somefile")) {
    // Read data etc.
} catch (IOException ex) {
    System.err.println("IO Error processing 'somefile': " + ex.getMessage());
    return;
}
```

If you wanted to distinguish IO errors thrown while opening and reading, you could use a nested try / catch. If you wanted to produce better diagnostics for open failures, you could perform the `exists`, `isFile` and `canRead` checks in the handler.

Section 130.5: Pitfall: thinking of variables as objects

No Java variable represents an object.

```
String foo;   // NOT AN OBJECT
```

Neither does any Java array contain objects.

```
String bar[] = new String[100];  // No member is an object.
```

If you mistakenly think of variables as objects, the actual behavior of the Java language will surprise you.

- For Java variables which have a primitive type (such as **`int`** or **`float`**) the variable holds a copy of the value. All copies of a primitive value are indistinguishable; i.e. there is only one **`int`** value for the number one. Primitive values are not objects and they do not behave like objects.
- For Java variables which have a reference type (either a class or an array type) the variable holds a reference. All copies of a reference are indistinguishable. References may point to objects, or they may be **`null`** which means that they point to no object. However, they are not objects and they don't behave like objects.

Variables are not objects in either case, and they don't contain objects in either case. They may contain *references to objects*, but that is saying something different.

Example class

The examples that follow use this class, which represents a point in 2D space.

```
public final class MutableLocation {
   public int x;
   public int y;

   public MutableLocation(int x, int y) {
       this.x = x;
       this.y = y;
   }

   public boolean equals(Object other) {
       if (!(other instanceof MutableLocation) {
           return false;
       }
       MutableLocation that = (MutableLocation) other;
       return this.x == that.x && this.y == that.y;
   }
}
```

An instance of this class is an object that has two fields `x` and `y` which have the type **`int`**.

We can have many instances of the `MutableLocation` class. Some will represent the same locations in 2D space; i.e. the respective values of `x` and `y` will match. Others will represent different locations.

Multiple variables can point to the same object

```
MutableLocation here = new MutableLocation(1, 2);
MutableLocation there = here;
MutableLocation elsewhere = new MutableLocation(1, 2);
```

In the above, we have declared three variables `here`, `there` and `elsewhere` that can hold references to `MutableLocation` objects.

If you (incorrectly) think of these variables as being objects, then you are likely to misread the statements as saying:

> 1. Copy the location "[1, 2]" to `here`
> 2. Copy the location "[1, 2]" to `there`
> 3. Copy the location "[1, 2]" to `elsewhere`

From that, you are likely to infer we have three independent objects in the three variables. In fact there are *only two objects created* by the above. The variables `here` and `there` actually refer to the same object.

We can demonstrate this. Assuming the variable declarations as above:

```
System.out.println("BEFORE: here.x is " + here.x + ", there.x is " + there.x +
                   "elsewhere.x is " + elsewhere.x);
here.x = 42;
System.out.println("AFTER: here.x is " + here.x + ", there.x is " + there.x +
                   "elsewhere.x is " + elsewhere.x);
```

This will output the following:

```
BEFORE: here.x is 1, there.x is 1, elsewhere.x is 1
AFTER: here.x is 42, there.x is 42, elsewhere.x is 1
```

We assigned a new value to `here.x` and it changed the value that we see via `there.x`. They are referring to the same object. But the value that we see via `elsewhere.x` has not changed, so `elsewhere` must refer to a different

object.

If a variable was an object, then the assignment `here.x = 42` would not change `there.x`.

The equality operator does NOT test that two objects are equal

Applying the equality (==) operator to reference values tests if the values refer to the same object. It does *not* test whether two (different) objects are "equal" in the intuitive sense.

```
MutableLocation here = new MutableLocation(1, 2);
MutableLocation there = here;
MutableLocation elsewhere = new MutableLocation(1, 2);

if (here == there) {
    System.out.println("here is there");
}
if (here == elsewhere) {
    System.out.println("here is elsewhere");
}
```

This will print "here is there", but it won't print "here is elsewhere". (The references in `here` and `elsewhere` are for two distinct objects.)

By contrast, if we call the `equals(Object)` method that we implemented above, we are going to test if two `MutableLocation` instances have an equal location.

```
if (here.equals(there)) {
    System.out.println("here equals there");
}
if (here.equals(elsewhere)) {
    System.out.println("here equals elsewhere");
}
```

This will print both messages. In particular, `here.equals(elsewhere)` returns **`true`** because the semantic criteria we chose for equality of two `MutableLocation` objects has been satisfied.

Method calls do NOT pass objects at all

Java method calls use *pass by value*1 to pass arguments and return a result.

When you pass a reference value to a method, you're actually passing a reference to an object *by value*, which means that it is creating a copy of the object reference.

As long as both object references are still pointing to the same object, you can modify that object from either reference, and this is what causes confusion for some.

However, you are *not* passing an object by reference2. The distinction is that if the object reference copy is modified to point to another object, the original object reference will still point to the original object.

```
void f(MutableLocation foo) {
    foo = new MutableLocation(3, 4);   // Point local foo at a different object.
}

void g() {
    MutableLocation foo = MutableLocation(1, 2);
    f(foo);
    System.out.println("foo.x is " + foo.x); // Prints "foo.x is 1".
```

```
}
```

Neither are you passing a copy of the object.

```
void f(MutableLocation foo) {
    foo.x = 42;
}

void g() {
    MutableLocation foo = new MutableLocation(0, 0);
    f(foo);
    System.out.println("foo.x is " + foo.x); // Prints "foo.x is 42"
}
```

1 - In languages like Python and Ruby, the term "pass by sharing" is preferred for "pass by value" of an object / reference.

2 - The term "pass by reference" or "call by reference" has a very specific meaning in programming language terminology. In effect, it means that you pass the address *of a variable or an array element*, so that when the called method assigns a new value to the formal argument, it changes the value in the original variable. Java does not support this. For a more fulsome description of different mechanisms for passing parameters, please refer to https://en.wikipedia.org/wiki/Evaluation_strategy.

Section 130.6: Pitfall: memory leaks

Java manages memory automatically. You are not required to free memory manually. An object's memory on the heap may be freed by a garbage collector when the object is no longer *reachable* by a live thread.

However, you can prevent memory from being freed, by allowing objects to be reachable that are no longer needed. Whether you call this a memory leak or memory packratting, the result is the same -- an unnecessary increase in allocated memory.

Memory leaks in Java can happen in various ways, but the most common reason is everlasting object references, because the garbage collector can't remove objects from the heap while there are still references to them.

Static fields

One can create such a reference by defining class with a **`static`** field containing some collection of objects, and forgetting to set that **`static`** field to **`null`** after the collection is no longer needed. **`static`** fields are considered GC roots and are never collected. Another issue is leaks in non-heap memory when JNI is used.

Classloader leak

By far, though, the most insidious type of memory leak is the classloader leak. A classloader holds a reference to every class it has loaded, and every class holds a reference to its classloader. Every object holds a reference to its class as well. Therefore, if even a *single* object of a class loaded by a classloader is not garbage, not a single class that that classloader has loaded can be collected. Since each class also refers to its static fields, they cannot be collected either.

Accumulation leak The accumulation leak example could look like the following:

```
final ScheduledExecutorService scheduledExecutorService = Executors.newScheduledThreadPool(1);
final Deque<BigDecimal> numbers = new LinkedBlockingDeque<>();
final BigDecimal divisor = new BigDecimal(51);

scheduledExecutorService.scheduleAtFixedRate(() -> {
```

```
    BigDecimal number = numbers.peekLast();
    if (number != null && number.remainder(divisor).byteValue() == 0) {
        System.out.println("Number: " + number);
        System.out.println("Deque size: " + numbers.size());
    }
}, 10, 10, TimeUnit.MILLISECONDS);

scheduledExecutorService.scheduleAtFixedRate(() -> {
    numbers.add(new BigDecimal(System.currentTimeMillis()));
}, 10, 10, TimeUnit.MILLISECONDS);

try {
    scheduledExecutorService.awaitTermination(1, TimeUnit.DAYS);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

This example creates two scheduled tasks. The first task takes the last number from a deque called `numbers`, and, if the number is divisible by 51, it prints the number and the deque's size. The second task puts numbers into the deque. Both tasks are scheduled at a fixed rate, and they run every 10 ms.

If the code is executed, you'll see that the size of the deque is permanently increasing. This will eventually cause the deque to be filled with objects that consume all available heap memory.

To prevent this while preserving the semantics of this program, we can use a different method for taking numbers from the deque: `pollLast`. Contrary to the method `peekLast`, `pollLast` returns the element and removes it from the deque while `peekLast` only returns the last element.

Section 130.7: Pitfall: Not understanding that String is an immutable class

New Java programmers often forget, or fail to fully comprehend, that the Java `String` class is immutable. This leads to problems like the one in the following example:

```
public class Shout {
    public static void main(String[] args) {
        for (String s : args) {
            s.toUpperCase();
            System.out.print(s);
            System.out.print(" ");
        }
        System.out.println();
    }
}
```

The above code is supposed to print command line arguments in upper case. Unfortunately, it does not work, the case of the arguments is not changed. The problem is this statement:

```
s.toUpperCase();
```

You might think that calling `toUpperCase()` is going to change `s` to an uppercase string. It doesn't. It can't! `String` objects are immutable. They cannot be changed.

In reality, the `toUpperCase()` method *returns* a `String` object which is an uppercase version of the `String` that you call it on. This will probably be a new `String` object, but if `s` was already all uppercase, the result could be the existing string.

So in order to use this method effectively, you need to use the object returned by the method call; for example:

```
s = s.toUpperCase();
```

In fact, the "strings never change" rule applies to all `String` methods. If you remember that, then you can avoid a whole category of beginner's mistakes.

Section 130.8: Pitfall: combining assignment and side-effects

Occasionally we see StackOverflow Java questions (and C or C++ questions) that ask what something like this:

```
i += a[i++] + b[i--];
```

evaluates to ... for some known initial states of `i`, `a` and `b`.

Generally speaking:

- for Java the answer is always specified1, but non-obvious, and often difficult to figure out
- for C and C++ the answer is often unspecified.

Such examples are often used in exams or job interviews as an attempt to see if the student or interviewee understands how expression evaluation really works in the Java programming language. This is arguably legitimate as a "test of knowledge", but that does not mean that you should ever do this in a real program.

To illustrate, the following seemingly simple example has appeared a few times in StackOverflow questions (like this one). In some cases, it appears as a genuine mistake in someone's code.

```
int a = 1;
a = a++;
System.out.println(a);    // What does this print.
```

Most programmers (including Java experts) reading those statements *quickly* would say that it outputs `2`. In fact, it outputs `1`. For a detailed explanation of why, please read this Answer.

However the real takeaway from this and similar examples is that *any* Java statement that *both* assigns to *and* side-effects the same variable is going to be *at best* hard to understand, and *at worst* downright misleading. You should avoid writing code like this.

1 - modulo potential issues with the Java Memory Model if the variables or objects are visible to other threads.

Chapter 131: Java Pitfalls - Exception usage

Several Java programming language misusage might conduct a program to generate incorrect results despite being compiled correctly. This topic main purpose is to list common **pitfalls** related to **exception handling**, and to propose the correct way to avoid having such pitfalls.

Section 131.1: Pitfall - Catching Throwable, Exception, Error or RuntimeException

A common thought pattern for inexperienced Java programmers is that exceptions are "a problem" or "a burden" and the best way to deal with this is catch them all1 as soon as possible. This leads to code like this:

```
....
try {
    InputStream is = new FileInputStream(fileName);
    // process the input
} catch (Exception ex) {
    System.out.println("Could not open file " + fileName);
}
```

The above code has a significant flaw. The `catch` is actually going to catch more exceptions than the programmer is expecting. Suppose that the value of the `fileName` is `null`, due to a bug elsewhere in the application. This will cause the `FileInputStream` constructor to throw a `NullPointerException`. The handler will catch this, and report to the user:

```
    Could not open file null
```

which is unhelpful and confusing. Worse still, suppose that the it was the "process the input" code that threw the unexpected exception (checked or unchecked!). Now the user will get the misleading message for a problem that didn't occur while opening the file, and may not be related to I/O at all.

The root of the problem is that the programmer has coded a handler for `Exception`. This is almost always a mistake:

- Catching `Exception` will catch all checked exceptions, and most unchecked exceptions as well.
- Catching `RuntimeException` will catch most unchecked exceptions.
- Catching `Error` will catch unchecked exceptions that signal JVM internal errors. These errors are generally not recoverable, and should not be caught.
- Catching `Throwable` will catch all possible exceptions.

The problem with catching too broad a set of exceptions is that the handler typically cannot handle all of them appropriately. In the case of the `Exception` and so on, it is difficult for the programmer to predict what *could* be caught; i.e. what to expect.

In general, the correct solution is to deal with the exceptions that *are* thrown. For example, you can catch them and handle them in situ:

```
try {
    InputStream is = new FileInputStream(fileName);
    // process the input
} catch (FileNotFoundException ex) {
    System.out.println("Could not open file " + fileName);
}
```

or you can declare them as `thrown` by the enclosing method.

There are very few situations where catching `Exception` is appropriate. The only one that arises commonly is something like this:

```
public static void main(String[] args) {
    try {
        // do stuff
    } catch (Exception ex) {
        System.err.println("Unfortunately an error has occurred. " +
                           "Please report this to X Y Z");
        // Write stacktrace to a log file.
        System.exit(1);
    }
}
```

Here we genuinely want to deal with all exceptions, so catching `Exception` (or even `Throwable`) is correct.

1 - Also known as Pokemon Exception Handling.

Section 131.2: Pitfall - Ignoring or squashing exceptions

This example is about deliberately ignoring or "squashing" exceptions. Or to be more precise, it is about how to catch and handle an exception in a way that ignores it. However, before we describe how to do this, we should first point out that squashing exceptions is generally not the correct way to deal with them.

Exceptions are usually thrown (by something) to notify other parts of the program that some significant (i.e. "exceptional") event has occurred. Generally (though not always) an exception means that something has gone wrong. If you code your program to squash the exception, there is a fair chance that the problem will reappear in another form. To make things worse, when you squash the exception, you are throwing away the information in the exception object and its associated stack trace. That is likely to make it harder to figure out what the original source of the problem was.

In practice, exception squashing frequently happens when you use an IDE's auto-correction feature to "fix" a compilation error caused by an unhandled exception. For example, you might see code like this:

```
try {
    inputStream = new FileInputStream("someFile");
} catch (IOException e) {
    /* add exception handling code here */
}
```

Clearly, the programmer has accepted the IDE's suggestion to make the compilation error go away, but the suggestion was inappropriate. (If the file open has failed, the program should most likely do something about it. With the above "correction", the program is liable to fail later; e.g. with a `NullPointerException` because `inputStream` is now **`null`**.)

Having said that, here is an example of deliberately squashing an exception. (For the purposes of argument, assume that we have determined that an interrupt while showing the selfie is harmless.) The comment tells the reader that we squashed the exception deliberately, and why we did that.

```
try {
    selfie.show();
} catch (InterruptedException e) {
    // It doesn't matter if showing the selfie is interrupted.
```

```
}
```

Another conventional way to highlight that we are *deliberately* squashing an exception without saying why is to indicate this with the exception variable's name, like this:

```
try {
    selfie.show();
} catch (InterruptedException ignored) {  }
```

Some IDEs (like IntelliJ IDEA) won't display a warning about the empty catch block if the variable name is set to `ignored`.

Section 131.3: Pitfall - Throwing Throwable, Exception, Error or RuntimeException

While catching the `Throwable`, `Exception`, `Error` and `RuntimeException` exceptions is bad, throwing them is even worse.

The basic problem is that when your application needs to handle exceptions, the presence of the top level exceptions make it hard to discriminate between different error conditions. For example

```
try {
    InputStream is = new FileInputStream(someFile);  // could throw IOException
    ...
    if (somethingBad) {
        throw new Exception();  // WRONG
    }
} catch (IOException ex) {
    System.err.println("cannot open ...");
} catch (Exception ex) {
    System.err.println("something bad happened");  // WRONG
}
```

The problem is that because we threw an `Exception` instance, we are forced to catch it. However as described in another example, catching `Exception` is bad. In this situation, it becomes difficult to discriminate between the "expected" case of an `Exception` that gets thrown if `somethingBad` is **`true`**, and the unexpected case where we actually catch an unchecked exception such as `NullPointerException`.

If the top-level exception is allowed to propagate, we run into other problems:

- We now have to remember all of the different reasons that we threw the top-level, and discriminate / handle them.
- In the case of `Exception` and `Throwable` we also need to add these exceptions to the **`throws`** clause of methods if we want the exception to propagate. This is problematic, as described below.

In short, don't throw these exceptions. Throw a more specific exception that more closely describes the "exceptional event" that has happened. If you need to, define and use a custom exception class.

Declaring Throwable or Exception in a method's "throws" is problematic.

It is tempting to replace a long list of thrown exceptions in a method's **`throws`** clause with `Exception` or even `Throwable. This is a bad idea:

1. It forces the caller to handle (or propagate) `Exception`.
2. We can no longer rely on the compiler to tell us about specific checked exceptions that need to be handled.

3. Handling `Exception` properly is difficult. It is hard to know what actual exceptions may be caught, and if you don't know what could be caught, it is hard to know what recovery strategy is appropriate.
4. Handling `Throwable` is even harder, since now you also have to cope with potential failures that should never be recovered from.

This advice means that certain other patterns should be avoided. For example:

```
try {
    doSomething();
} catch (Exception ex) {
    report(ex);
    throw ex;
}
```

The above attempts to log all exceptions as they pass, without definitively handling them. Unfortunately, prior to Java 7, the `throw ex;` statement caused the compiler to think that any `Exception` could be thrown. That could force you to declare the enclosing method as `throws Exception`. From Java 7 onwards, the compiler knows that the set of exceptions that could be (re-thrown) there is smaller.

Section 131.4: Pitfall - Using exceptions for normal flowcontrol

There is a mantra that some Java experts are won't to recite:

> "Exceptions should only be used for exceptional cases."

(For example: http://programmers.stackexchange.com/questions/184654)

The essence of this is that is it is a bad idea (in Java) to use exceptions and exception handling to implement normal flow control. For example, compare these two ways of dealing with a parameter that could be null.

```
public String truncateWordOrNull(String word, int maxLength) {
    if (word == null) {
        return "";
    } else {
        return word.substring(0, Math.min(word.length(), maxLength));
    }
}

public String truncateWordOrNull(String word, int maxLength) {
    try {
        return word.substring(0, Math.min(word.length(), maxLength));
    } catch (NullPointerException ex) {
        return "";
    }
}
```

In this example, we are (by design) treating the case where `word` is `null` as if it is an empty word. The two versions deal with `null` either using conventional *if ... else* and or *try ... catch*. How should we decide which version is better?

The first criterion is readability. While readability is hard to quantify objectively, most programmers would agree that the essential meaning of the first version is easier to discern. Indeed, in order to truly understand the second form, you need to understand that a `NullPointerException` cannot be thrown by the `Math.min` or `String.substring` methods.

The second criterion is efficiency. In releases of Java prior to Java 8, the second version is significantly (orders of

magnitude) slower than the first version. In particular, the construction of an exception object entails capturing and recording the stackframes, just in case the stacktrace is required.

On the other hand, there are many situations where using exceptions is more readable, more efficient and (sometimes) more correct than using conditional code to deal with "exceptional" events. Indeed, there are rare situations where it is necessary to use them for "non-exceptional" events; i.e. events that occur relatively frequently. For the latter, it is worth looking at ways to reduce the overheads of creating exception objects.

Section 131.5: Pitfall - Directly subclassing `Throwable`

`Throwable` has two direct subclasses, `Exception` and `Error`. While it's possible to create a new class that extends `Throwable` directly, this is inadvisable as many applications assume only `Exception` and `Error` exist.

More to the point there is no practical benefit to directly subclassing `Throwable`, as the resulting class is, in effect, simply a checked exception. Subclassing `Exception` instead will result in the same behavior, but will more clearly convey your intent.

Section 131.6: Pitfall - Catching InterruptedException

As already pointed out in other pitfalls, catching all exceptions by using

```
try {
    // Some code
} catch (Exception) {
    // Some error handling
}
```

Comes with a lot of different problems. But one perticular problem is that it can lead to deadlocks as it breaks the interrupt system when writing multi-threaded applications.

If you start a thread you usually also need to be able to stop it abruptly for various reasons.

```
Thread t = new Thread(new Runnable() {
    public void run() {
         while (true) {
             //Do something indefinetely
         }
    }
}

t.start();

//Do something else

// The thread should be canceld if it is still active.
// A Better way to solve this is with a shared variable that is tested
// regularily by the thread for a clean exit, but for this example we try to
// forcibly interrupt this thread.
if (t.isAlive()) {
   t.interrupt();
   t.join();
}

//Continue with program
```

The `t.interrupt()` will raise an InterruptedException in that thread, than is intended to shut down the thread. But what if the Thread needs to clean up some resources before its completely stopped? For this it can catch the

InterruptedException and do some cleanup.

```
Thread t = new Thread(new Runnable() {
    public void run() {
        try {
            while (true) {
                //Do something indefinetely
            }
        } catch (InterruptedException ex) {
            //Do some quick cleanup

            // In this case a simple return would do.
            // But if you are not 100% sure that the thread ends after
            // catching the InterruptedException you will need to raise another
            // one for the layers surrounding this code.
            Thread.currentThread().interrupt();
        }
    }
}
```

But if you have a catch-all expression in your code, the InterruptedException will be caught by it as well and the interruption will not continue. Which in this case could lead to a deadlock as the parent thread waits indefinitely for this thead to stop with `t.join()`.

```
Thread t = new Thread(new Runnable() {
    public void run() {
        try {
            while (true) {
                try {
                    //Do something indefinetely
                }
                catch (Exception ex) {
                    ex.printStackTrace();
                }
            }
        } catch (InterruptedException ex) {
            // Dead code as the interrupt exception was already caught in
            // the inner try-catch
            Thread.currentThread().interrupt();
        }
    }
}
```

So it is better to catch Exceptions individually, but if you insist on using a catch-all, at least catch the InterruptedException individually beforehand.

```
Thread t = new Thread(new Runnable() {
    public void run() {
        try {
            while (true) {
                try {
                    //Do something indefinetely
                } catch (InterruptedException ex) {
                    throw ex; //Send it up in the chain
                } catch (Exception ex) {
                    ex.printStackTrace();
                }
            }
        } catch (InterruptedException ex) {
            // Some quick cleanup code
```

```
            Thread.currentThread().interrupt();
        }
    }
}
```

Section 131.7: Pitfall - Excessive or inappropriate stacktraces

One of the more annoying things that programmers can do is to scatter calls to `printStackTrace()` throughout their code.

The problem is that the `printStackTrace()` is going to write the stacktrace to standard output.

- For an application that is intended for end-users who are not Java programmers, a stacktrace is uninformative at best, and alarming at worst.
- For a server-side application, the chances are that nobody will look at the standard output.

A better idea is to not call `printStackTrace` directly, or if you do call it, do it in a way that the stack trace is written to a log file or error file rather than to the end-user's console.

One way to do this is to use a logging framework, and pass the exception object as a parameter of the log event. However, even logging the exception can be harmful if done injudiciously. Consider the following:

```
public void method1() throws SomeException {
    try {
        method2();
        // Do something
    } catch (SomeException ex) {
        Logger.getLogger().warn("Something bad in method1", ex);
        throw ex;
    }
}

public void method2() throws SomeException {
    try {
        // Do something else
    } catch (SomeException ex) {
        Logger.getLogger().warn("Something bad in method2", ex);
        throw ex;
    }
}
```

If the exception is thrown in `method2`, you are likely to see two copies of the same stacktrace in the logfile, corresponding to the same failure.

In short, either log the exception or re-throw it further (possibly wrapped with another exception). Don't do both.

Chapter 132: Java Pitfalls - Language syntax

Several Java programming language misusage might conduct a program to generate incorrect results despite being compiled correctly. This topic main purpose is to list common pitfalls with their causes, and to propose the correct way to avoid falling in such problems.

Section 132.1: Pitfall - Missing a ‘break’ in a 'switch' case

These Java issues can be very embarrassing, and sometimes remain undiscovered until run in production. Fallthrough behavior in switch statements is often useful; however, missing a “break” keyword when such behavior is not desired can lead to disastrous results. If you have forgotten to put a “break” in “case 0” in the code example below, the program will write “Zero” followed by “One”, since the control flow inside here will go through the entire “switch” statement until it reaches a “break”. For example:

```
public static void switchCasePrimer() {
        int caseIndex = 0;
        switch (caseIndex) {
            case 0:
                System.out.println("Zero");
            case 1:
                System.out.println("One");
                break;
            case 2:
                System.out.println("Two");
                break;
            default:
                System.out.println("Default");
        }
}
```

In most cases, the cleaner solution would be to use interfaces and move code with specific behaviour into separate implementations (*composition over inheritance*)

If a switch-statement is unavoidable it is recommended to document "expected" fallthroughs if they occur. That way you show fellow developers that you are aware of the missing break, and that this is expected behaviour.

```
switch(caseIndex) {
    [...]
    case 2:
        System.out.println("Two");
        // fallthrough
    default:
        System.out.println("Default");
```

Section 132.2: Pitfall - Declaring classes with the same names as standard classes

Sometimes, programmers who are new to Java make the mistake of defining a class with a name that is the same as a widely used class. For example:

```
package com.example;

/**
```

```
 * My string utilities
 */
public class String {
    ....
}
```

Then they wonder why they get unexpected errors. For example:

```
package com.example;

public class Test {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

If you compile and then attempt to run the above classes you will get an error:

```
$ javac com/example/*.java
$ java com.example.Test
Error: Main method not found in class test.Test, please define the main method as:
   public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
```

Someone looking at the code for the `Test` class would see the declaration of `main` and look at its signature and wonder what the `java` command is complaining about. But in fact, the `java` command is telling the truth.

When we declare a version of `String` in the same package as `Test`, this version takes precedence over the automatic import of `java.lang.String`. Thus, the signature of the `Test.main` method is actually

```
void main(com.example.String[] args)
```

instead of

```
void main(java.lang.String[] args)
```

and the `java` command will not recognize *that* as an entrypoint method.

Lesson: Do not define classes that have the same name as existing classes in `java.lang`, or other commonly used classes in the Java SE library. If you do that, you are setting yourself open for all sorts of obscure errors.

Section 132.3: Pitfall - Leaving out braces: the "dangling if" and "dangling else" problems

The latest version of the Oracle Java style guide mandates that the "then" and "else" statements in an `if` statement should always be enclosed in "braces" or "curly brackets". Similar rules apply to the bodies of various loop statements.

```
if (a) {           // <- open brace
    doSomething();
    doSomeMore();
}                  // <- close brace
```

This is not actually required by Java language syntax. Indeed, if the "then" part of an `if` statement is a single statement, it is legal to leave out the braces

```
if (a)
    doSomething();
```

or even

```
if (a) doSomething();
```

However there are dangers in ignoring Java style rules and leaving out the braces. Specifically, you significantly increase the risk that code with faulty indentation will be misread.

The "dangling if" problem:

Consider the example code from above, rewritten without braces.

```
if (a)
   doSomething();
   doSomeMore();
```

This code *seems to say* that the calls to `doSomething` and `doSomeMore` will both occur *if and only if* `a` is **`true`**. In fact, the code is incorrectly indented. The Java Language Specification that the `doSomeMore()` call is a separate statement following the `if` statement. The correct indentation is as follows:

```
if (a)
   doSomething();
doSomeMore();
```

The "dangling else" problem

A second problem appears when we add **`else`** to the mix. Consider the following example with missing braces.

```
if (a)
   if (b)
      doX();
   else if (c)
      doY();
else
   doZ();
```

The code above *seems to say* that `doZ` will be called when `a` is **`false`**. In fact, the indentation is incorrect once again. The correct indentation for the code is:

```
if (a)
   if (b)
      doX();
   else if (c)
      doY();
   else
      doZ();
```

If the code was written according to the Java style rules, it would actually look like this:

```
if (a) {
   if (b) {
      doX();
   } else if (c) {
      doY();
   } else {
```

```
      doZ();
   }
}
```

To illustrate why that is better, suppose that you had accidentally mis-indented the code. You might end up with something like this:

```
if (a) {
   if (b) {
      doX();
   } else if (c) {
      doY();
} else {
   doZ();
}
}
```

```
if (a) {
    if (b) {
       doX();
    } else if (c) {
       doY();
} else {
    doZ();
    }
}
```

But in both cases, the mis-indented code "looks wrong" to the eye of an experienced Java programmer.

Section 132.4: Pitfall - Octal literals

Consider the following code snippet:

```
// Print the sum of the numbers 1 to 10
int count = 0;
for (int i = 1; i < 010; i++) {    // Mistake here ....
    count = count + i;
}
System.out.println("The sum of 1 to 10 is " + count);
```

A Java beginner might be surprised to know that the above program prints the wrong answer. It actually prints the sum of the numbers 1 to 8.

The reason is that an integer literal that starts with the digit zero ('0') is interpreted by the Java compiler as an octal literal, not a decimal literal as you might expect. Thus, `010` is the octal number 10, which is 8 in decimal.

Section 132.5: Pitfall - Using '==' to test a boolean

Sometimes a new Java programmer will write code like this:

```
public void check(boolean ok) {
    if (ok == true) {           // Note 'ok == true'
        System.out.println("It is OK");
    }
}
```

An experienced programmer would spot that as being clumsy and want to rewrite it as:

```
public void check(boolean ok) {
    if (ok) {
       System.out.println("It is OK");
    }
}
```

However, there is more wrong with `ok == true` than simple clumsiness. Consider this variation:

```
public void check(boolean ok) {
```

```
    if (ok = true) {            // Oooops!
        System.out.println("It is OK");
    }
}
```

Here the programmer has mistyped `==` as `=` ... and now the code has a subtle bug. The expression `x = true` unconditionally assigns `true` to `x` and then evaluates to `true`. In other words, the `check` method will now print "It is OK" no matter what the parameter was.

The lesson here is to get out of the habit of using `== false` and `== true`. In addition to being verbose, they make your coding more error prone.

Note: A possible alternative to `ok == true` that avoids the pitfall is to use Yoda conditions; i.e. put the literal on the left side of the relational operator, as in `true == ok`. This works, but most programmers would probably agree that Yoda conditions look odd. Certainly `ok` (or `!ok`) is more concise and more natural.

Section 132.6: Pitfall - Ignoring method visibility

Even experienced Java developers tend to think that Java has only three protection modifiers. The language actually has four! The **package private** (a.k.a. default) level of visibility is often forgotten.

You should pay attention to what methods you make public. The public methods in an application are the application's visible API. This should be as small and compact as possible, especially if you are writing a reusable library (see also the SOLID principle). It is important to similarly consider the visibility of all methods, and to only use protected or package private access where appropriate.

When you declare methods that should be **private** as public, you expose the internal implementation details of the class.

A corollary to this is that you only unit test the public methods of your class - in fact you can **only** test public methods. It is bad practice to increase the visibility of private methods just to be able to run unit tests against those methods. Testing public methods that call the methods with more restrictive visibility should be sufficient to test an entire API. You should **never** expand your API with more public methods only to allow unit testing.

Section 132.7: Pitfall: Using 'assert' for argument or user input validation

A question that occasionally on StackOverflow is whether it is appropriate to use `assert` to validate arguments supplied to a method, or even inputs provided by the user.

The simple answer is that it is not appropriate.

Better alternatives include:

- Throwing an IllegalArgumentException using custom code.
- Using the `Preconditions` methods available in Google Guava library.
- Using the `Validate` methods available in Apache Commons Lang3 library.

This is what the Java Language Specification (JLS 14.10, for Java 8) advises on this matter:

> Typically, assertion checking is enabled during program development and testing, and disabled for deployment, to improve performance.
>
> Because assertions may be disabled, programs must not assume that the expressions contained in

> assertions will be evaluated. Thus, these boolean expressions should generally be free of side effects. Evaluating such a boolean expression should not affect any state that is visible after the evaluation is complete. It is not illegal for a boolean expression contained in an assertion to have a side effect, but it is generally inappropriate, as it could cause program behavior to vary depending on whether assertions were enabled or disabled.
>
> In light of this, assertions should not be used for argument checking in public methods. Argument checking is typically part of the contract of a method, and this contract must be upheld whether assertions are enabled or disabled.
>
> A secondary problem with using assertions for argument checking is that erroneous arguments should result in an appropriate run-time exception (such as `IllegalArgumentException`, `ArrayIndexOutOfBoundsException`, or `NullPointerException`). An assertion failure will not throw an appropriate exception. Again, it is not illegal to use assertions for argument checking on public methods, but it is generally inappropriate. It is intended that `AssertionError` never be caught, but it is possible to do so, thus the rules for try statements should treat assertions appearing in a try block similarly to the current treatment of throw statements.

Section 132.8: Pitfall - Wildcard imports can make your code fragile

Consider the following partial example:

```
import com.example.somelib.*;
import com.acme.otherlib.*;

public class Test {
    private Context x = new Context();   // from com.example.somelib
    ...
}
```

Suppose that when when you first developed the code against version 1.0 of `somelib` and version 1.0 of `otherlib`. Then at some later point, you need to upgrade your dependencies to a later versions, and you decide to use `otherlib` version 2.0. Also suppose that one of the changes that they made to `otherlib` between 1.0 and 2.0 was to add a `Context` class.

Now when you recompile `Test`, you will get a compilation error telling you that `Context` is an ambiguous import.

If you are familiar with the codebase, this probably is just a minor inconvenience. If not, then you have some work to do to address this problem, here and potentially elsewhere.

The problem here is the wildcard imports. On the one hand, using wildcards can make your classes a few lines shorter. On the other hand:

- Upwards compatible changes to other parts of your codebase, to Java standard libraries or to 3rd party libraries can lead to compilation errors.
- Readability suffers. Unless you are using an IDE, figuring out which of the wildcard imports is pulling in a named class can be difficult.

The lesson is that it is a bad idea to use wildcard imports in code that needs to be long lived. Specific (non-wildcard) imports are not much effort to maintain if you use an IDE, and the effort is worthwhile.

Section 132.9: Pitfall - Misplaced semicolons and missing braces

This is a mistake that causes real confusion for Java beginners, at least the first time that they do it. Instead of writing this:

```
if (feeling == HAPPY)
    System.out.println("Smile");
else
    System.out.println("Frown");
```

they accidentally write this:

```
if (feeling == HAPPY);
    System.out.println("Smile");
else
    System.out.println("Frown");
```

and are puzzled when the Java compiler tells them that the `else` is misplaced. The Java compiler with interpret the above as follows:

```
if (feeling == HAPPY)
    /*empty statement*/ ;
System.out.println("Smile");   // This is unconditional
else                           // This is misplaced.  A statement cannot
                               // start with 'else'
System.out.println("Frown");
```

In other cases, there will be no be compilation errors, but the code won't do what the programmer intends. For example:

```
for (int i = 0; i < 5; i++);
    System.out.println("Hello");
```

only prints "Hello" once. Once again, the spurious semicolon means that the body of the `for` loop is an empty statement. That means that the `println` call that follows is unconditional.

Another variation:

```
for (int i = 0; i < 5; i++);
    System.out.println("The number is " + i);
```

This will give a "Cannot find symbol" error for `i`. The presence of the spurious semicolon means that the `println` call is attempting to use `i` outside of its scope.

In those examples, there is a straight-forward solution: simply delete the spurious semicolon. However, there are some deeper lessons to be drawn from these examples:

1. The semicolon in Java is not "syntactic noise". The presence or absence of a semicolon can change the meaning of your program. Don't just add them at the end of every line.
2. Don't trust your code's indentation. In the Java language, extra whitespace at the beginning of a line is ignored by the compiler.
3. Use an automatic indenter. All IDEs and many simple text editors understand how to correctly indent Java

code.

4. This is the most important lesson. Follow the latest Java style guidelines, and put braces around the "then" and "else" statements and the body statement of a loop. The open brace ({) should not be on a new line.

If the programmer followed the style rules then the `if` example with a misplaced semicolons would look like this:

```
if (feeling == HAPPY); {
    System.out.println("Smile");
} else {
    System.out.println("Frown");
}
```

That looks odd to an experienced eye. If you auto-indented that code, it would probably look like this:

```
if (feeling == HAPPY); {
                           System.out.println("Smile");
                       } else {
                           System.out.println("Frown");
                       }
```

which should stand out as wrong to even a beginner.

Section 132.10: Pitfall - Overloading instead of overriding

Consider the following example:

```
public final class Person {
    private final String firstName;
    private final String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = (firstName == null) ? "" : firstName;
        this.lastName = (lastName == null) ? "" : lastName;
    }

    public boolean equals(String other) {
        if (!(other instanceof Person)) {
            return false;
        }
        Person p = (Person) other;
        return firstName.equals(p.firstName) &&
                lastName.equals(p.lastName);
    }

    public int hashcode() {
        return firstName.hashCode() + 31 * lastName.hashCode();
    }
}
```

This code is not going to behave as expected. The problem is that the `equals` and `hashcode` methods for `Person` do not override the standard methods defined by `Object`.

- The `equals` method has the wrong signature. It should be declared as `equals(Object)` not `equals(String)`.
- The `hashcode` method has the wrong name. It should be `hashCode()` (note the capital **C**).

These mistakes mean that we have declared accidental overloads, and these won't be used if `Person` is used in a

polymorphic context.

However, there is a simple way to deal with this (from Java 5 onwards). Use the `@Override` annotation whenever you *intend* your method to be an override:

Version ≥ Java SE 5

```
public final class Person {
    ...

    @Override
    public boolean equals(String other) {
        ....
    }

    @Override
    public hashcode() {
        ....
    }
}
```

When we add an `@Override` annotation to a method declaration, the compiler will check that the method *does* override (or implement) a method declared in a superclass or interface. So in the example above, the compiler will give us two compilation errors, which should be enough to alert us to the mistake.

Section 132.11: Pitfall of Auto-Unboxing Null Objects into Primitives

```
public class Foobar {
    public static void main(String[] args) {

        // example:
        Boolean ignore = null;
        if (ignore == false) {
            System.out.println("Do not ignore!");
        }
    }
}
```

The pitfall here is that `null` is compared to `false`. Since we're comparing a primitive `boolean` against a `Boolean`, Java attempts to *unbox* the the `Boolean Object` into a primitive equivalent, ready for comparison. However, since that value is `null`, a `NullPointerException` is thrown.

Java is incapable of comparing primitive types against `null` values, which causes a `NullPointerException` at runtime. Consider the primitive case of the condition `false == null`; this would generate a *compile time* error `incomparable types: int and <null>`.

Chapter 133: Java Pitfalls - Threads and Concurrency

Section 133.1: Pitfall - Extending 'java.lang.Thread'

The javadoc for the `Thread` class shows two ways to define and use a thread:

Using a custom thread class:

```
class PrimeThread extends Thread {
    long minPrime;
    PrimeThread(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
         . . .
    }
}

PrimeThread p = new PrimeThread(143);
p.start();
```

Using a `Runnable`:

```
class PrimeRun implements Runnable {
    long minPrime;
    PrimeRun(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
         . . .
    }
}

PrimeRun p = new PrimeRun(143);
new Thread(p).start();
```

(Source: `java.lang.Thread` javadoc.)

The custom thread class approach works, but it has a few problems:

1. It is awkward to use `PrimeThread` in a context that uses a classic thread pool, an executor, or the ForkJoin framework. (It is not impossible, because `PrimeThread` indirectly implements `Runnable`, but using a custom `Thread` class as a `Runnable` is certainly clumsy, and may not be viable ... depending on other aspects of the class.)
2. There is more opportunity for mistakes in other methods. For example, if you declared a `PrimeThread.start()` without delegating to `Thread.start()`, you would end up with a "thread" that ran on the current thread.

The approach of putting the thread logic into a `Runnable` avoids these problems. Indeed, if you use an anonymous class (Java 1.1 onwards) to implement the `Runnable` the result is more succinct, and more readable than the

examples above.

```
final long minPrime = ...
new Thread(new Runnable() {
    public void run() {
        // compute primes larger than minPrime
         . . .
    }
}.start();
```

With a lambda expression (Java 8 onwards), the above example would become even more elegant:

```
final long minPrime = ...
new Thread(() -> {
   // compute primes larger than minPrime
    . . .
}).start();
```

Section 133.2: Pitfall - Too many threads makes an application slower

A lot of people who are new to multi-threading think that using threads automatically make an application go faster. In fact, it is a lot more complicated than that. But one thing that we can state with certainty is that for any computer there is a limit on the number of threads that can be run at the same time:

- A computer has a fixed number of *cores* (or *hyperthreads*).
- A Java thread has to be *scheduled* to a core or hyperthread in order to run.
- If there are more runnable Java threads than (available) cores / hyperthreads, some of them must wait.

This tells us that simply creating more and more Java threads *cannot* make the application go faster and faster. But there are other considerations as well:

- Each thread requires an off-heap memory region for its thread stack. The typical (default) thread stack size is 512Kbytes or 1Mbytes. If you have a significant number of threads, the memory usage can be significant.
- Each active thread will refer to a number of objects in the heap. That increases the working set of *reachable* objects, which impacts on garbage collection and on physical memory usage.
- The overheads of switching between threads is non-trivial. It typically entails a switch into the OS kernel space to make a thread scheduling decision.
- The overheads of thread synchronization and inter-thread signaling (e.g. wait(), notify() / notifyAll) *can be* significant.

Depending on the details of your application, these factors generally mean that there is a "sweet spot" for the number of threads. Beyond that, adding more threads gives minimal performance improvement, and can make performance worse.

If your application create for each new task, then an unexpected increase in the workload (e.g. a high request rate) can lead to catastrophic behavior.

A better way to deal with this is to use bounded thread pool whose size you can control (statically or dynamically). When there is too much work to do, the application needs to queue the requests. If you use an `ExecutorService`, it will take care of the thread pool management and task queuing.

Section 133.3: Pitfall: incorrect use of wait() / notify()

The methods `object.wait()`, `object.notify()` and `object.notifyAll()` are meant to be used in a very specific way. (see http://stackoverflow.com/documentation/java/5409/wait-notify#t=20160811161648303307)

The "Lost Notification" problem

One common beginner mistake is to unconditionally call `object.wait()`

```
private final Object lock = new Object();

public void myConsumer() {
    synchronized (lock) {
        lock.wait();     // DON'T DO THIS!!
    }
    doSomething();
}
```

The reason this is wrong is that it depends on some other thread to call `lock.notify()` or `lock.notifyAll()`, but nothing guarantees that the other thread did not make that call *before* the consumer thread called `lock.wait()`.

`lock.notify()` and `lock.notifyAll()` do not do anything at all if some other thread is not *already* waiting for the notification. The thread that calls `myConsumer()` in this example will hang forever if it is too late to catch the notification.

The "Illegal Monitor State" bug

If you call `wait()` or `notify()` on an object without holding the lock, then the JVM will throw `IllegalMonitorStateException`.

```
public void myConsumer() {
    lock.wait();      // throws exception
    consume();
}

public void myProducer() {
    produce();
    lock.notify();    // throws exception
}
```

(The design for `wait()` / `notify()` requires that the lock is held because this is necessary to avoid systemic race conditions. If it was possible to call `wait()` or `notify()` without locking, then it would be impossible to implement the primary use-case for these primitives: waiting for a condition to occur.)

Wait / notify is too low-level

The *best* way to avoid problems with `wait()` and `notify()` is to not use them. Most synchronization problems can be solved by using the higher-level synchronization objects (queues, barriers, semaphores, etc.) that are available in the `java.utils.concurrent` package.

Section 133.4: Pitfall: Shared variables require proper synchronization

Consider this example:

```
public class ThreadTest implements Runnable {
```

```
    private boolean stop = false;

    public void run() {
        long counter = 0;
        while (!stop) {
            counter = counter + 1;
        }
        System.out.println("Counted " + counter);
    }

    public static void main(String[] args) {
        ThreadTest tt = new ThreadTest();
        new Thread(tt).start();    // Create and start child thread
        Thread.sleep(1000);
        tt.stop = true;            // Tell child thread to stop.
    }
}
```

The intent of this program is intended to start a thread, let it run for 1000 milliseconds, and then cause it to stop by setting the `stop` flag.

Will it work as intended?

Maybe yes, may be no.

An application does not necessarily stop when the `main` method returns. If another thread has been created, and that thread has not been marked as a daemon thread, then the application will continue to run after the main thread has ended. In this example, that means that the application will keep running until child thread ends. That should happens when `tt.stop` is set to **`true`**.

But that is actually not strictly true. In fact, the child thread will stop after it has *observed* `stop` with the value **`true`**. Will that happen? Maybe yes, maybe no.

The Java Language Specification *guarantees* that memory reads and writes made in a thread are visible to that thread, as per the order of the statements in the source code. However, in general, this is NOT guaranteed when one thread writes and another thread (subsequently) reads. To get guaranteed visibility, there needs to be a chain of *happens-before* relations between a write and a subsequent read. In the example above, there is no such chain for the update to the `stop` flag, and therefore it is not guaranteed that the child thread will see `stop` change to **`true`**.

(Note to authors: There should be a separate Topic on the Java Memory Model to go into the deep technical details.)

How do we fix the problem?

In this case, there are two simple ways to ensure that the `stop` update is visible:

1. Declare `stop` to be **`volatile`**; i.e.

   ```
   private volatile boolean stop = false;
   ```

   For a **`volatile`** variable, the JLS specifies that there is a *happens-before* relation between a write by one thread and a later read by a second thread.

2. Use a mutex to synchronize as follows:

```
public class ThreadTest implements Runnable {
```

```
    private boolean stop = false;

    public void run() {
        long counter = 0;
        while (true) {
            synchronize (this) {
                if (stop) {
                    break;
                }
            }
            counter = counter + 1;
        }
        System.out.println("Counted " + counter);
    }

    public static void main(String[] args) {
        ThreadTest tt = new ThreadTest();
        new Thread(tt).start();    // Create and start child thread
        Thread.sleep(1000);
        synchronize (tt) {
            tt.stop = true;        // Tell child thread to stop.
        }
    }
}
```

In addition to ensuring that there is mutual exclusion, the JLS specifies that there is a *happens-before* relation between the releasing a mutex in one thread and gaining the same mutex in a second thread.

But isn't assignment atomic?

Yes it is!

However, that fact does not mean that the effects of update will be visible simultaneously to all threads. Only a proper chain of *happens-before* relations will guarantee that.

Why did they do this?

Programmers doing multi-threaded programming in Java for the first time find the Memory Model is challenging. Programs behave in an unintuitive way because the natural expectation is that writes are visible uniformly. So why the Java designers design the Memory Model this way.

It actually comes down to a compromise between performance and ease of use (for the programmer).

A modern computer architecture consists of multiple processors (cores) with individual register sets. Main memory is accessible either to all processors or to groups of processors. Another property of modern computer hardware is that access to registers is typically orders of magnitude faster to access than access to main memory. As the number of cores scales up, it is easy to see that reading and writing to main memory can become a system's main performance bottleneck.

This mismatch is addressed by implementing one or more levels of memory caching between the processor cores and main memory. Each core access memory cells via its cache. Normally, a main memory read only happens when there is a cache miss, and a main memory write only happens when a cache line needs to be flushed. For an application where each core's working set of memory locations will fit into its cache, the core speed is no longer limited by main memory speed / bandwidth.

But that gives us a new problem when multiple cores are reading and writing shared variables. The latest version of a variable may sit in one core's cache. Unless the that core flushes the cache line to main memory, AND other cores

invalidate their cached copy of older versions, some of them are liable to see stale versions of the variable. But if the caches were flushed to memory each time there is a cache write ("just in case" there was a read by another core) that would consume main memory bandwidth unnecessarily.

The standard solution used at the hardware instruction set level is to provide instructions for cache invalidation and a cache write-through, and leave it to the compiler to decide when to use them.

Returning to Java. the Memory Model is designed so that the Java compilers are not required to issue cache invalidation and write-through instructions where they are not really needed. The assumption is that the programmer will use an appropriate synchronization mechanism (e.g. primitive mutexes, **`volatile`**, higher-level concurrency classes and so on) to indicate that it needs memory visibility. In the absence of a *happens-before* relation, the Java compilers are free to *assume* that no cache operations (or similar) are required.

This has significant performance advantages for multi-threaded applications, but the downside is that writing correct multi-threaded applications is not a simple matter. The programmer *does* have to understand what he or she is doing.

Why can't I reproduce this?

There are a number of reasons why problems like this are difficult to reproduce:

1. As explained above, the consequence of not dealing with memory visibility issues problems properly is *typically* that your compiled application does not handle the memory caches correctly. However, as we alluded to above, memory caches often get flushed anyway.
2. When you change the hardware platform, the characteristics of the memory caches may change. This can lead to different behavior if your application does not synchronize correctly.
3. You may be observing the effects of *serendipitous* synchronization. For example, if you add traceprints, their is typically some synchronization happening behind the scenes in the I/O streams that causes cache flushes. So adding traceprints *often* causes the application to behave differently.
4. Running an application under a debugger causes it to be compiled differently by the JIT compiler. Breakpoints and single stepping exacerbate this. These effects will often change the way an application behaves.

These things make bugs that are due to inadequate synchronization particularly difficult to solve.

Section 133.5: Pitfall - Thread creation is relatively expensive

Consider these two micro-benchmarks:

The first benchmark simply creates, starts and joins threads. The thread's `Runnable` does no work.

```
public class ThreadTest {
    public static void main(String[] args) throws Exception {
        while (true) {
            long start = System.nanoTime();
            for (int i = 0; i < 100_000; i++) {
                Thread t = new Thread(new Runnable() {
                        public void run() {
                });});
                t.start();
                t.join();
            }
            long end = System.nanoTime();
```

```
            System.out.println((end - start) / 100_000.0);
        }
    }
}

$ java ThreadTest
34627.91355
33596.66021
33661.19084
33699.44895
33603.097
33759.3928
33671.5719
33619.46809
33679.92508
33500.32862
33409.70188
33475.70541
33925.87848
33672.89529
^C
```

On a typical modern PC running Linux with 64bit Java 8 u101, this benchmark shows an average time taken to create, start and join thread of between 33.6 and 33.9 microseconds.

The second benchmark does the equivalent to the first but using an `ExecutorService` to submit tasks and a `Future` to rendezvous with the end of the task.

```
import java.util.concurrent.*;

public class ExecutorTest {
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        while (true) {
            long start = System.nanoTime();
            for (int i = 0; i < 100_000; i++) {
                Future<?> future = exec.submit(new Runnable() {
                    public void run() {
                    }
                });
                future.get();
            }
            long end = System.nanoTime();
            System.out.println((end - start) / 100_000.0);
        }
    }
}

$ java ExecutorTest
6714.66053
5418.24901
5571.65213
5307.83651
5294.44132
5370.69978
5291.83493
5386.23932
5384.06842
5293.14126
5445.17405
5389.70685
```

```
^C
```

As you can see, the averages are between 5.3 and 5.6 microseconds.

While the actual times will depend on a variety of factors, the difference between these two results is significant. It is clearly faster to use a thread pool to recycle threads than it is to create new threads.

Chapter 134: Java Pitfalls - Nulls and NullPointerException

Section 134.1: Pitfall - "Making good" unexpected nulls

On StackOverflow, we often see code like this in Answers:

```
public String joinStrings(String a, String b) {
    if (a == null) {
        a = "";
    }
    if (b == null) {
        b = "";
    }
    return a + ": " + b;
}
```

Often, this is accompanied with an assertion that is "best practice" to test for `null` like this to avoid `NullPointerException`.

Is it best practice? In short: No.

There are some underlying assumptions that need to be questioned before we can say if it is a good idea to do this in our `joinStrings`:

What does it mean for "a" or "b" to be null?

A `String` value can be zero or more characters, so we already have a way of representing an empty string. Does `null` mean something different to `""`? If no, then it is problematic to have two ways to represent an empty string.

Did the null come from an uninitialized variable?

A `null` can come from an uninitialized field, or an uninitialized array element. The value could be uninitialized by design, or by accident. If it was by accident then this is a bug.

Does the null represent a "don't know" or "missing value"?

Sometimes a `null` can have a genuine meaning; e.g. that the real value of a variable is unknown or unavailable or "optional". In Java 8, the `Optional` class provides a better way of expressing that.

If this is a bug (or a design error) should we "make good"?

One interpretation of the code is that we are "making good" an unexpected `null` by using an empty string in its place. Is the correct strategy? Would it be better to let the `NullPointerException` happen, and then catch the exception further up the stack and log it as a bug?

The problem with "making good" is that it is liable to either hide the problem, or make it harder to diagnose.

Is this efficient / good for code quality?

If the "make good" approach is used consistently, your code is going to contain a lot of "defensive" null tests. This is going to make it longer and harder to read. Furthermore, all of this testing and "making good" is liable to impact on the performance of your application.

In summary

If `null` is a meaningful value, then testing for the `null` case is the correct approach. The corollary is that if a `null` value is meaningful, then this should be clearly documented in the javadocs of any methods that accept the `null` value or return it.

Otherwise, it is a better idea to treat an unexpected `null` as a programming error, and let the `NullPointerException` happen so that the developer gets to know there is a problem in the code.

Section 134.2: Pitfall - Using null to represent an empty array or collection

Some programmers think that it is a good idea to save space by using a `null` to represent an empty array or collection. While it is true that you can save a small amount of space, the flipside is that it makes your code more complicated, and more fragile. Compare these two versions of a method for summing an array:

The first version is how you would normally code the method:

```
/**
 * Sum the values in an array of integers.
 * @arg values the array to be summed
 * @return the sum
 **/
public int sum(int[] values) {
    int sum = 0;
    for (int value : values) {
        sum += value;
    }
    return sum;
}
```

The second version is how you need to code the method if you are in the habit of using `null` to represent an empty array.

```
/**
 * Sum the values in an array of integers.
 * @arg values the array to be summed, or null.
 * @return the sum, or zero if the array is null.
 **/
public int sum(int[] values) {
    int sum = 0;
    if (values != null) {
        for (int value : values) {
            sum += value;
        }
    }
    return sum;
}
```

As you can see, the code is a bit more complicated. This is directly attributable to the decision to use `null` in this way.

Now consider if this array that might be a `null` is used in lots of places. At each place where you use it, you need to consider whether you need to test for `null`. If you miss a `null` test that needs to be there, you risk a `NullPointerException`. Hence, the strategy of using `null` in this way leads to your application being more fragile; i.e. more vulnerable to the consequences of programmer errors.

The lesson here is to use empty arrays and empty lists when that is what you mean.

```
int[] values = new int[0];                    // always empty
List<Integer> list = new ArrayList();         // initially empty
List<Integer> list = Collections.emptyList(); // always empty
```

The space overhead is small, and there are other ways to minimize it if this this is a worthwhile thing to do.

Section 134.3: Pitfall - Not checking if an I/O stream isn't even initialized when closing it

To prevent memory leaks, one should not forget to close an input stream or an output stream whose job is done. This is usually done with a **`try-catch-finally`** statement without the **`catch`** part:

```
void writeNullBytesToAFile(int count, String filename) throws IOException {
    FileOutputStream out = null;
    try {
        out = new FileOutputStream(filename);
        for(; count > 0; count--)
            out.write(0);
    } finally {
        out.close();
    }
}
```

While the above code might look innocent, it has a flaw that can make debugging impossible. If the line where `out` is initialized (`out = new FileOutputStream(filename)`) throws an exception, then `out` will be **`null`** when `out.close()` is executed, resulting in a nasty `NullPointerException`!

To prevent this, simply make sure the stream isn't **`null`** before trying to close it.

```
void writeNullBytesToAFile(int count, String filename) throws IOException {
    FileOutputStream out = null;
    try {
        out = new FileOutputStream(filename);
        for(; count > 0; count--)
            out.write(0);
    } finally {
        if (out != null)
            out.close();
    }
}
```

An even better approach is to **`try`**-with-resources, since it'll automatically close the stream with a probability of 0 to throw an NPE without the need of a **`finally`** block.

```
void writeNullBytesToAFile(int count, String filename) throws IOException {
    try (FileOutputStream out = new FileOutputStream(filename)) {
        for(; count > 0; count--)
            out.write(0);
    }
}
```

Section 134.4: Pitfall - Returning null instead of throwing an exception

Some Java programmers have a general aversion to throwing or propagating exceptions. This leads to code like the following:

```
public Reader getReader(String pathname) {
    try {
        return new BufferedReader(FileReader(pathname));
    } catch (IOException ex) {
        System.out.println("Open failed: " + ex.getMessage());
        return null;
    }
```

}

So what is the problem with that?

The problem is that the `getReader` is returning a **`null`** as a special value to indicate that the `Reader` could not be opened. Now the returned value needs to be tested to see if it is **`null`** before it is used. If the test is left out, the result will be a `NullPointerException`.

There are actually three problems here:

1. The `IOException` was caught too soon.
2. The structure of this code means that there is a risk of leaking a resource.
3. A **`null`** was used then returned because no "real" `Reader` was available to return.

In fact, assuming that the exception did need to be caught early like this, there were a couple of alternatives to returning **`null`**:

1. It would be possible to implement a `NullReader` class; e.g. one where API's operations behaves as if the reader was already at the "end of file" position.
2. With Java 8, it would be possible to declare `getReader` as returning an `Optional<Reader>`.

Section 134.5: Pitfall - Unnecessary use of Primitive Wrappers can lead to NullPointerExceptions

Sometimes, programmers who are new Java will use primitive types and wrappers interchangeably. This can lead to problems. Consider this example:

```
public class MyRecord {
    public int a, b;
    public Integer c, d;
}

...
MyRecord record = new MyRecord();
record.a = 1;               // OK
record.b = record.b + 1;    // OK
record.c = 1;               // OK
record.d = record.d + 1;    // throws a NullPointerException
```

Our `MyRecord` class1 relies on default initialization to initialize the values on its fields. Thus, when we **`new`** a record, the `a` and `b` fields will be set to zero, and the `c` and `d` fields will be set to **`null`**.

When we try to use the default initialized fields, we see that the **`int`** fields works all of the time, but the `Integer` fields work in some cases and not others. Specifically, in the case that fails (with `d`), what happens is that the expression on the right-hand side attempts to unbox a **`null`** reference, and that is what causes the `NullPointerException` to be thrown.

There are a couple of ways to look at this:

- If the fields `c` and `d` need to be primitive wrappers, then either we should not be relying on default initialization, or we should be testing for **`null`**. For former is the correct approach *unless* there is a definite meaning for the fields in the **`null`** state.
- If the fields don't need to be primitive wrappers, then it is a mistake to make them primitive wrappers. In addition to this problem, the primitive wrappers have extra overheads relative to primitive types.

The lesson here is to not use primitive wrapper types unless you really need to.

1 - This class is not an example of good coding practice. For instance, a well-designed class would not have public fields. However, that is not the point of this example.

Section 134.6: Pitfall - Using "Yoda notation" to avoid NullPointerException

A lot of example code posted on StackOverflow includes snippets like this:

```
if ("A".equals(someString)) {
    // do something
}
```

This does "prevent" or "avoid" a possible `NullPointerException` in the case that `someString` is **`null`**. Furthermore, it is arguable that

```
    "A".equals(someString)
```

is better than:

```
    someString != null && someString.equals("A")
```

(It is more concise, and in some circumstances it might be more efficient. However, as we argue below, conciseness could be a negative.)

However, the real pitfall is using the Yoda test **to avoid `NullPointerExceptions`** as a matter of habit.

When you write `"A".equals(someString)` you are actually "making good" the case where `someString` happens to be **`null`**. But as another example (Pitfall - "Making good" unexpected nulls) explains, "making good" **`null`** values can be harmful for a variety of reasons.

This means that Yoda conditions are not "best practice"1. Unless the **`null`** is expected, it is better to let the `NullPointerException` happen so that you can get a unit test failure (or a bug report). That allows you to find and fix the bug that caused the unexpected / unwanted **`null`** to appear.

Yoda conditions should only be used in cases where the **`null`** is *expected* because the object you are testing has come from an API that is *documented* as returning a **`null`**. And arguably, it could be better to use one of the less pretty ways expressing the test because that helps to highlight the **`null`** test to someone who is reviewing your code.

1 - According to Wikipedia: *"Best coding practices are a set of informal rules that the software development community has learned over time which can help improve the quality of software."*. Using Yoda notation does not achieve this. In a lot of situations, it makes the code worse.

Chapter 135: Java Pitfalls - Performance Issues

This topic describes a number of "pitfalls" (i.e. mistakes that novice java programmers make) that relate to Java application performance.

Section 135.1: Pitfall - String concatenation in a loop does not scale

Consider the following code as an illustration:

```
public String joinWords(List<String> words) {
    String message = "";
    for (String word : words) {
        message = message + " " + word;
    }
    return message;
}
```

Unfortunate this code is inefficient if the `words` list is long. The root of the problem is this statement:

```
message = message + " " + word;
```

For each loop iteration, this statement creates a new `message` string containing a copy of all characters in the original `message` string with extra characters appended to it. This generates a lot of temporary strings, and does a lot of copying.

When we analyse `joinWords`, assuming that there are N words with an average length of M, we find that O(N) temporary strings are created and O(M.N2) characters will be copied in the process. The N2 component is particularly troubling.

The recommended approach for this kind of problem1 is to use a `StringBuilder` instead of string concatenation as follows:

```
public String joinWords2(List<String> words) {
    StringBuilder message = new StringBuilder();
    for (String word : words) {
        message.append(" ").append(word);
    }
    return message.toString();
}
```

The analysis of `joinWords2` needs to take account of the overheads of "growing" the `StringBuilder` backing array that holds the builder's characters. However, it turns out that the number of new objects created is O(logN) and that the number of characters copied is O(M.N) characters. The latter includes characters copied in the final `toString()` call.

(It may be possible to tune this further, by creating the `StringBuilder` with the correct capacity to start with. However, the overall complexity remains the same.)

Returning to the original `joinWords` method, it turns out that the critical statement will be optimized by a typical Java compiler to something like this:

```
  StringBuilder tmp = new StringBuilder();
```

```
    tmp.append(message).append(" ").append(word);
    message = tmp.toString();
```

However, the Java compiler will not "hoist" the `StringBuilder` out of the loop, as we did by hand in the code for `joinWords2`.

Reference:

- "Is Java's String '+' operator in a loop slow?"

1 - In Java 8 and later, the `Joiner` class can be used to solve this particular problem. However, that is not what this example is *really supposed to be about*.

Section 135.2: Pitfall - Using size() to test if a collection is empty is inefficient

The Java Collections Framework provides two related methods for all `Collection` objects:

- `size()` returns the number of entries in a `Collection`, and
- `isEmpty()` method returns true if (and only if) the `Collection` is empty.

Both methods can be used to test for collection emptiness. For example:

```
Collection<String> strings = new ArrayList<>();
boolean isEmpty_wrong = strings.size() == 0; // Avoid this
boolean isEmpty = strings.isEmpty();         // Best
```

While these approaches look the same, some collection implementations do not store the size. For such a collection, the implementation of `size()` needs to calculate the size each time it is called. For instance:

- A simple linked list class (but not the `java.util.LinkedList`) might need to traverse the list to count the elements.
- The `ConcurrentHashMap` class needs to sum the entries in all of the map's "segments".
- A lazy implementation of a collection might need to realize the entire collection in memory in order to count the elements.

By contrast, an `isEmpty()` method only needs to test if there is *at least one* element in the collection. This does not entail counting the elements.

While `size() == 0` is not always less efficient that `isEmpty()`, it is inconceivable for a properly implemented `isEmpty()` to be less efficient than `size() == 0`. Hence `isEmpty()` is preferred.

Section 135.3: Pitfall - Interning strings so that you can use == is a bad idea

When some programmers see this advice:

> "Testing strings using `==` is incorrect (unless the strings are interned)"

their initial reaction is to intern strings so that they can use `==`. (After all `==` is faster than calling `String.equals(...)`, isn't it.)

This is the wrong approach, from a number of perspectives:

Fragility

First of all, you can only safely use `==` if you know that *all* of the `String` objects you are testing have been interned. The JLS guarantees that String literals in your source code will have been interned. However, none of the standard Java SE APIs guarantee to return interned strings, apart from `String.intern(String)` itself. If you miss just one source of `String` objects that haven't been interned, your application will be unreliable. That unreliability will manifest itself as false negatives rather than exceptions which is liable to make it harder to detect.

Costs of using 'intern()'

Under the hood, interning works by maintaining a hash table that contains previously interned `String` objects. Some kind of weak reference mechanism is used so that the interning hash table does not become a storage leak. While the hash table is implemented in native code (unlike `HashMap`, `HashTable` and so on), the `intern` calls are still relatively costly in terms of CPU and memory used.

This cost has to be compared with the saving of we are going to get by using `==` instead of `equals`. In fact, we are not going to break even unless each interned string is compared with other strings "a few" times.

(Aside: the few situations where interning is worthwhile tend to be about reducing the memory foot print of an application where the same strings recur many times, *and* those strings have a long lifetime.)

The impact on garbage collection

In addition to the direct CPU and memory costs described above, interned Strings impact on the garbage collector performance.

For versions of Java prior to Java 7, interned strings are held in the "PermGen" space which is collected infrequently. If PermGen needs to be collected, this (typically) triggers a full garbage collection. If the PermGen space fills completely, the JVM crashes, even if there was free space in the regular heap spaces.

In Java 7, the string pool was moved out of "PermGen" into the normal heap. However, the hash table is still going to be a long-lived data structure, which is going to cause any interned strings to be long-lived. (Even if the interned string objects were allocated in Eden space they would most likely be promoted before they were collected.)

Thus in all cases, interning a string is going to prolong its lifetime relative to an ordinary string. That will increase the garbage collection overheads over the lifetime of the JVM.

The second issue is that the hash table needs to use a weak reference mechanism of some kind to prevent string interning leaking memory. But such a mechanism is more work for the garbage collector.

These garbage collection overheads are difficult to quantify, but there is little doubt that they do exist. If you use `intern` extensively, they could be significant.

The string pool hashtable size

According to this source, from Java 6 onwards, the string pool is implemented as fixed sized hash table with chains to deal with strings that hash to the same bucket. In early releases of Java 6, the hash table had a (hard-wired) constant size. A tuning parameter (`-XX:StringTableSize`) was added as a mid-life update to Java 6. Then in a mid-life update to Java 7, the default size of the pool was changed from `1009` to `60013`.

The bottom line is that if you do intend to use `intern` intensively in your code, it is *advisable* to pick a version of Java where the hashtable size is tunable and make sure that you tune the size it appropriately. Otherwise, the performance of `intern` is liable to degrade as the pool gets larger.

Interning as a potential denial of service vector

The hashcode algorithm for strings is well-known. If you intern strings supplied by malicious users or applications, this could be used as part of a denial of service (DoS) attack. If the malicious agent arranges that all of the strings it provides have the same hash code, this could lead to an unbalanced hash table and `O(N)` performance for `intern` ... where `N` is the number of collided strings.

(There are simpler / more effective ways to launch a DoS attack against a service. However, this vector could be used if the goal of the DoS attack is to break security, or to evade first-line DoS defences.)

Section 135.4: Pitfall - Using 'new' to create primitive wrapper instances is inefficient

The Java language allows you to use **`new`** to create instances `Integer`, `Boolean` and so on, but it is generally a bad idea. It is better to either use autoboxing (Java 5 and later) or the `valueOf` method.

```
Integer i1 = new Integer(1);      // BAD
Integer i2 = 2;                   // BEST (autoboxing)
Integer i3 = Integer.valueOf(3);  // OK
```

The reason that using **`new`** `Integer(`**`int`**`)` explicitly is a bad idea is that it creates a new object (unless optimized out by JIT compiler). By contrast, when autoboxing or an explicit `valueOf` call are used, the Java runtime will try to reuse an `Integer` object from a cache of pre-existing objects. Each time the runtime has a cache "hit", it avoids creating an object. This also saves heap memory and reduces GC overheads caused by object churn.

Notes:

1. In recent Java implementations, autoboxing is implemented by calling `valueOf`, and there are caches for `Boolean`, `Byte`, `Short`, `Integer`, `Long` and `Character`.
2. The caching behavior for the integral types is mandated by the Java Language Specification.

Section 135.5: Pitfall - Efficiency concerns with regular expressions

Regular expression matching is a powerful tool (in Java, and in other contexts) but it does have some drawbacks. One of these that regular expressions tends to be rather expensive.

Pattern and Matcher instances should be reused

Consider the following example:

```
/**
 * Test if all strings in a list consist of English letters and numbers.
 * @param strings the list to be checked
 * @return 'true' if an only if all strings satisfy the criteria
 * @throws NullPointerException if 'strings' is 'null' or a 'null' element.
 */
public boolean allAlphanumeric(List<String> strings) {
    for (String s : strings) {
        if (!s.matches("[A-Za-z0-9]*")) {
            return false;
        }
    }
    return true;
}
```

This code is correct, but it is inefficient. The problem is in the `matches(...)` call. Under the hood, `s.matches("[A-`

Za-z0-9]*") is equivalent to this:

```
Pattern.matches(s, "[A-Za-z0-9]*")
```

which is in turn equivalent to

```
Pattern.compile("[A-Za-z0-9]*").matcher(s).matches()
```

The `Pattern.compile("[A-Za-z0-9]*")` call parses the regular expression, analyze it, and construct a `Pattern` object that holds the data structure that will be used by the regex engine. This is a non-trivial computation. Then a `Matcher` object is created to wrap the `s` argument. Finally we call `match()` to do the actual pattern matching.

The problem is that this work is all repeated for each loop iteration. The solution is to restructure the code as follows:

```
private static Pattern ALPHA_NUMERIC = Pattern.compile("[A-Za-z0-9]*");

public boolean allAlphanumeric(List<String> strings) {
    Matcher matcher = ALPHA_NUMERIC.matcher("");
    for (String s : strings) {
        matcher.reset(s);
        if (!matcher.matches()) {
            return false;
        }
    }
    return true;
}
```

Note that the javadoc for `Pattern` states:

> Instances of this class are immutable and are safe for use by multiple concurrent threads. Instances of the `Matcher` class are not safe for such use.

Don't use match() when you should use find()

Suppose you want to test if a string `s` contains three or more digits in a row. You cn express this in various ways including:

```
  if (s.matches(".*[0-9]{3}.*")) {
      System.out.println("matches");
  }
```

or

```
  if (Pattern.compile("[0-9]{3}").matcher(s).find()) {
      System.out.println("matches");
  }
```

The first one is more concise, but it is also likely to be less efficient. On the face of it, the first version is going to try to match the entire string against the pattern. Furthermore, since ".*" is a "greedy" pattern, the pattern matcher is likely to advance "eagerly" try to the end of the string, and backtrack until it finds a match.

By contrast, the second version will search from left to right and will stop searching as soon as it finds the 3 digits in a row.

Use more efficient alternatives to regular expressions

Regular expressions are a powerful tool, but they should not be your only tool. A lot of tasks can be done more efficiently in other ways. For example:

```
Pattern.compile("ABC").matcher(s).find()
```

does the same thing as:

```
s.contains("ABC")
```

except that the latter is a lot more efficient. (Even if you can amortize the cost of compiling the regular expression.)

Often, the non-regex form is more complicated. For example, the test performed by the `matches()` call the earlier `allAlplanumeric` method can be rewritten as:

```
public boolean matches(String s) {
    for (char c : s) {
        if ((c >= 'A' && c <= 'Z') ||
            (c >= 'a' && c <= 'z') ||
            (c >= '0' && c <= '9')) {
             return false;
        }
    }
    return true;
}
```

Now that is more code than using a `Matcher`, but it is also going to be significantly faster.

Catastrophic Backtracking

(This is potentially a problem with all implementations of regular expressions, but we will mention it here because it is a pitfall for `Pattern` usage.)

Consider this (contrived) example:

```
Pattern pat = Pattern.compile("(A+)+B");
System.out.println(pat.matcher("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAB").matches());
System.out.println(pat.matcher("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAC").matches());
```

The first `println` call will quickly print **`true`**. The second one will print **`false`**. Eventually. Indeed, if you experiment with the code above, you will see that each time you add an `A` before the `C`, the time take will double.

This is behavior is an example of *catastrophic backtracking*. The pattern matching engine that implements the regex matching is fruitlessly trying all of the *possible* ways that the pattern *might* match.

Let us look at what `(A+)+B` actually means. Superficially, it seems to say "one or more `A` characters followed by a `B` value", but in reality it says one or more groups, each of which consists of one or more `A` characters. So, for example:

- 'AB' matches one way only: '(A)B'
- 'AAB' matches two ways: '(AA)B' or '(A)(A)B`
- 'AAAB' matches four ways: '(AAA)B' or '(AA)(A)Bor `'(A)(AA)B` or '(A)(A)(A)B`
- and so on

In other words, the number of possible matches is 2N where N is the number of `A` characters.

The above example is clearly contrived, but patterns that exhibit this kind of performance characteristics (i.e. `O(2^N)` or `O(N^K)` for a large `K`) arise frequently when ill-considered regular expressions are used. There are some standard remedies:

- Avoid nesting repeating patterns within other repeating patterns.
- Avoid using too many repeating patterns.
- Use non-backtracking repetition as appropriate.
- Don't use regexes for complicated parsing tasks. (Write a proper parser instead.)

Finally, beware of situations where a user or an API client can supply a regex string with pathological characteristics. That can lead to accidental or deliberate "denial of service".

References:

- The Regular Expressions tag, particularly http://stackoverflow.com/documentation/regex/977/backtracking#t=20161001033913136116 3 and http://stackoverflow.com/documentation/regex/4527/when-you-should-not-use-regular-expressions#t=20161001033959356491 3
- "Regex Performance" by Jeff Atwood.
- "How to kill Java with a Regular Expression" by Andreas Haufler.

Section 135.6: Pitfall - Small reads / writes on unbuffered streams are inefficient

Consider the following code to copy one file to another:

```
import java.io.*;

public class FileCopy {

    public static void main(String[] args) throws Exception {
        try (InputStream is = new FileInputStream(args[0]);
             OutputStream os = new FileOutputStream(args[1])) {
            int octet;
            while ((octet = is.read()) != -1) {
                os.write(octet);
            }
        }
    }
}
```

(We have deliberated omitted normal argument checking, error reporting and so on because they are not relevant to *point* of this example.)

If you compile the above code and use it to copy a huge file, you will notice that it is very slow. In fact, it will be at least a couple of orders of magnitude slower than the standard OS file copy utilities.

(*Add actual performance measurements here!*)

The primary reason that the example above is slow (in the large file case) is that it is performing one-byte reads and one-byte writes on unbuffered byte streams. The simple way to improve performance is to wrap the streams with buffered streams. For example:

```
import java.io.*;

public class FileCopy {
```

```
    public static void main(String[] args) throws Exception {
        try (InputStream is = new BufferedInputStream(
                    new FileInputStream(args[0]));
             OutputStream os = new BufferedOutputStream(
                    new FileOutputStream(args[1]))) {
            int octet;
            while ((octet = is.read()) != -1) {
                os.write(octet);
            }
        }
    }
}
```

These small changes will improve data copy rate by *at least* a couple of orders of magnitude, depending on various platform-related factors. The buffered stream wrappers cause the data to be read and written in larger chunks. The instances both have buffers implemented as byte arrays.

- With `is`, data is read from the file into the buffer a few kilobytes at a time. When `read()` is called, the implementation will typically return a byte from the buffer. It will only read from the underlying input stream if the buffer has been emptied.
- The behavior for `os` is analogous. Calls to `os.write(int)` write single bytes into the buffer. Data is only written to the output stream when the buffer is full, or when `os` is flushed or closed.

What about character-based streams?

As you should be aware, Java I/O provides different APIs for reading and writing binary and text data.

- `InputStream` and `OutputStream` are the base APIs for stream-based binary I/O
- `Reader` and `Writer` are the base APIs for stream-based text I/O.

For text I/O, `BufferedReader` and `BufferedWriter` are the equivalents for `BufferedInputStream` and `BufferedOutputStream`.

Why do buffered streams make this much difference?

The real reason that buffered streams help performance is to do with the way that an application talks to the operating system:

- Java method in a Java application, or native procedure calls in the JVM's native runtime libraries are fast. They typically take a couple of machine instructions and have minimal performance impact.
- By contrast, JVM runtime calls to the operating system are not fast. They involve something known as a "syscall". The typical pattern for a syscall is as follows:

    1. Put the syscall arguments into registers.
    2. Execute a SYSENTER trap instruction.
    3. The trap handler switched to privileged state and changes the virtual memory mappings. Then it dispatches to the code to handle the specific syscall.
    4. The syscall handler checks the arguments, taking care that it isn't being told to access memory that the user process should not see.
    5. The syscall specific work is performed. In the case of a `read` syscall, this may involve:
        1. checking that there is data to be read at the file descriptor's current position
        2. calling the file system handler to fetch the required data from disk (or wherever it is stored) into the buffer cache,
        3. copying data from the buffer cache to the JVM-supplied address

4. adjusting thstream pointerse file descriptor position
6. Return from the syscall. This entails changing VM mappings again and switching out of privileged state.

As you can imagine, performing a single syscall can thousands of machine instructions. Conservatively, *at least* two orders of magnitude longer than a regular method call. (Probably three or more.)

Given this, the reason that buffered streams make a big difference is that they drastically reduce the number of syscalls. Instead of doing a syscall for each `read()` call, the buffered input stream reads a large amount of data into a buffer as required. Most `read()` calls on the buffered stream do some simple bounds checking and return a `byte` that was read previously. Similar reasoning applies in the output stream case, and also the character stream cases.

(Some people think that buffered I/O performance comes from the mismatch between the read request size and the size of a disk block, disk rotational latency and things like that. In fact, a modern OS uses a number of strategies to ensure that the application *typically* doesn't need to wait for the disk. This is not the real explanation.)

Are buffered streams always a win?

Not always. Buffered streams are definitely a win if your application is going to do lots of "small" reads or writes. However, if your application only needs to perform large reads or writes to / from a large `byte[]` or `char[]`, then buffered streams will give you no real benefits. Indeed there might even be a (tiny) performance penalty.

Is this the fastest way to copy a file in Java?

No it isn't. When you use Java's stream-based APIs to copy a file, you incur the cost of at least one extra memory-to-memory copy of the data. It is possible to avoid this if your use the NIO `ByteBuffer` and `Channel` APIs. (*Add a link to a separate example here.*)

Section 135.7: Pitfall - Over-use of primitive wrapper types is inefficient

Consider these two pieces of code:

```
int a = 1000;
int b = a + 1;
```

and

```
Integer a = 1000;
Integer b = a + 1;
```

Question: Which version is more efficient?

Answer: The two versions look almost the identical, but the first version is a lot more efficient than the second one.

The second version is using a representation for the numbers that uses more space, and is relying on auto-boxing and auto-unboxing behind the scenes. In fact the second version is directly equivalent to the following code:

```
Integer a = Integer.valueOf(1000);               // box 1000
Integer b = Integer.valueOf(a.intValue() + 1);   // unbox 1000, add 1, box 1001
```

Comparing this to the other version that uses `int`, there are clearly three extra method calls when `Integer` is used. In the case of `valueOf`, the calls are each going to create and initialize a new `Integer` object. All of this extra boxing and unboxing work is likely to make the second version an order of magnitude slower than the first one.

In addition to that, the second version is allocating objects on the heap in each `valueOf` call. While the space utilization is platform specific, it is likely to be in the region of 16 bytes for each `Integer` object. By contrast, the **`int`** version needs zero extra heap space, assuming that `a` and `b` are local variables.

Another big reason why primitives are faster then their boxed equivalent is how their respective array types are laid out in memory.

If you take **`int`**`[]` and `Integer[]` as an example, in the case of an **`int`**`[]` the **`int`** *values* are contiguously laid out in memory. But in the case of an `Integer[]` it's not the values that are laid out, but references (pointers) to `Integer` objects, which in turn contain the actual **`int`** values.

Besides being an extra level of indirection, this can be a big tank when it comes to cache locality when iterating over the values. In the case of an **`int`**`[]` the CPU could fetch all the values in the array, into it's cache at once, because they are contiguous in memory. But in the case of an `Integer[]` the CPU potentially has to do an additional memory fetch for each element, since the array only contains references to the actual values.

In short, using primitive wrapper types is relatively expensive in both CPU and memory resources. Using them unnecessarily is in efficient.

Section 135.8: Pitfall - The overheads of creating log messages

`TRACE` and `DEBUG` log levels are there to be able to convey high detail about the operation of the given code at runtime. Setting the log level above these is usually recommended, however some care must be taken for these statements to not affect performance even when seemingly "turned off".

Consider this log statement:

```
// Processing a request of some kind, logging the parameters
LOG.debug("Request coming from " + myInetAddress.toString()
          + " parameters: " + Arrays.toString(veryLongParamArray));
```

Even when the log level is set to `INFO`, arguments passed to `debug()` will be evaluated on each execution of the line. This makes it unnecessarily consuming on several counts:

- `String` concatenation: multiple `String` instances will be created
- `InetAddress` might even do a DNS lookup.
- the `veryLongParamArray` might be very long - creating a String out of it consumes memory, takes time

Solution

Most logging framework provide means to create log messages using fix strings and object references. The log message will be evaluated only if the message is actually logged. Example:

```
// No toString() evaluation, no string concatenation if debug is disabled
LOG.debug("Request coming from {} parameters: {}", myInetAddress, parameters));
```

This works very well as long as all parameters can be converted to strings using String.valueOf(Object). If the log message compuation is more complex, the log level can be checked before logging:

```
if (LOG.isDebugEnabled()) {
    // Argument expression evaluated only when DEBUG is enabled
    LOG.debug("Request coming from {}, parameters: {}", myInetAddress,
              Arrays.toString(veryLongParamArray);
```

```
}
```

Here, `LOG.debug()` with the costly `Arrays.toString(Obect[])` computation is processed only when `DEBUG` is actually enabled.

Section 135.9: Pitfall - Iterating a Map's keys can be inefficient

The following example code is slower than it needs to be :

```
Map<String, String> map = new HashMap<>();
for (String key : map.keySet()) {
    String value = map.get(key);
    // Do something with key and value
}
```

That is because it requires a map lookup (the `get()` method) for each key in the map. This lookup may not be efficient (in a HashMap, it entails calling `hashCode` on the key, then looking up the correct bucket in internal data structures, and sometimes even calling `equals`). On a large map, this may not be a trivial overhead.

The correct way of avoiding this is to iterate on the map's entries, which is detailed in the Collections topic

Section 135.10: Pitfall - Calling System.gc() is inefficient

It is (almost always) a bad idea to call `System.gc()`.

The javadoc for the `gc()` method specifies the following:

> "Calling the `gc` method suggests that the Java Virtual Machine expend effort toward recycling unused objects in order to make the memory they currently occupy available for quick reuse. When control returns from the method call, the Java Virtual Machine has made a best effort to reclaim space from all discarded objects."

There are a couple of important points that can be drawn from this:

1. The use of the word "suggests" rather than (say) "tells" means that the JVM is free to ignore the suggestion. The default JVM behavior (recent releases) is to follow the suggestion, but this can be overridden by setting `-XX:+DisableExplicitGC` when when launching the JVM.
2. The phrase "a best effort to reclaim space from all discarded objects" implies that calling `gc` will trigger a "full" garbage collection.

So why is calling `System.gc()` a bad idea?

First, running a full garbage collection is expensive. A full GC involves visiting and "marking" every object that is still reachable; i.e. every object that is not garbage. If you trigger this when there isn't much garbage to be collected, then the GC does a lot of work for relatively little benefit.

Second, a full garbage collection is liable to disturb the "locality" properties of the objects that are not collected. Objects that are allocated by the same thread at roughly the same time tend to be allocated close together in memory. This is good. Objects that are allocated at the same time are likely to be related; i.e. reference each other. If your application uses those references, then the chances are that memory access will be faster because of various memory and page caching effects. Unfortunately, a full garbage collection tend to move objects around so that objects that were once close are now further apart.

Third, running a full garbage collection is liable to make your application pause until the collection is complete. While this is happening, your application will be non-responsive.

In fact, the best strategy is to let the JVM decide when to run the GC, and what kind of collection to run. If you don't interfere, the JVM will choose a time and collection type that optimizes throughput or minimizes GC pause times.

At the beginning we said "... (almost always) a bad idea ...". In fact there are a couple of scenarios where it *might* be a good idea:

1. If you are implementing a unit test for some code that is garbage collection sensitive (e.g. something involving finalizers or weak / soft / phantom references) then calling `System.gc()` may be necessary.
2. In some interactive applications, there can be particular points in time where the user won't care if there is a garbage collection pause. One example is a game where there are natural pauses in the "play"; e.g. when loading a new level.

Section 135.11: Pitfall - Calling 'new String(String)' is inefficient

Using `new String(String)` to duplicate a string is inefficient and almost always unnecessary.

- String objects are immutable, so there is no need to copy them to protect against changes.
- In some older versions of Java, `String` objects can share backing arrays with other `String` objects. In those versions, it is possible to leak memory by creating a (small) substring of a (large) string and retaining it. However, from Java 7 onwards, `String` backing arrays are not shared.

In the absence of any tangible benefit, calling `new String(String)` is simply wasteful:

- Making the copy takes CPU time.
- The copy uses more memory which increases the application's memoru footprint and / or increases GC overheads.
- Operations like `equals(Object)` and `hashCode()` can be slower if String objects are copied.

Chapter 136: ServiceLoader

Section 136.1: Simple ServiceLoader Example

The ServiceLoader is a simple and easy to use built-in mechanism for dynamic loading of interface implementations. With the service loader - providing means for instantation (but not the wiring) - a simple dependency injection mechanism can be built in Java SE. With the ServiceLoader interface and implementation separation becomes natural and programs can be conveniently extended. Actually a lot of Java API are implented based on the ServiceLoader

The basic concepts are

- Operating on *interfaces* of services
- Obtaining implementation(s) of the service via `ServiceLoader`
- Providing implementation of servics

Lets start with the interface and put it in a jar, named for example `accounting-api.jar`

```
package example;

public interface AccountingService {

  long getBalance();
}
```

Now we provide an implementation of that service in a jar named `accounting-impl.jar`, containing an implementation of the service

```
package example.impl;
import example.AccountingService;

public interface DefaultAccountingService implements AccouningService {

  public long getBalance() {
    return balanceFromDB();
  }

  private long balanceFromDB(){
    ...
  }
}
```

further, the `accounting-impl.jar` contains a file declaring that this jar provides an implementation of `AccountingService`. The file has to have a path starting with `META-INF/services/` and must have the same name as the *fully-qualified* name of the interface:

- `META-INF/services/example.AccountingService`

The content of the file is the *fully-qualfified* name of the implementation:

```
example.impl.DefaultAccountingService
```

Given both jars are in the classpath of the program, that consumes the `AccountingService`, an instance of the Service can be obtained by using the ServiceLauncher

```
ServiceLoader<AccountingService> loader = ServiceLoader.load(AccountingService.class)
AccountingService service = loader.next();
long balance = service.getBalance();
```

As the `ServiceLoader` is an `Iterable`, it supports multiple implementation providers, where the program may choose from:

```
ServiceLoader<AccountingService> loader = ServiceLoader.load(AccountingService.class)
for(AccountingService service : loader) {
   //...
}
```

Note that when invoking `next()` a new instance will allways be created. If you want to re-use an instance, you have to use the `iterator()` method of the ServiceLoader or the for-each loop as shown above.

Section 136.2: Logger Service

The following example shows how to instantiate a class for logging via the `ServiceLoader`.

Service

```
package servicetest;

import java.io.IOException;

public interface Logger extends AutoCloseable {

    void log(String message) throws IOException;
}
```

Implementations of the service

The following implementation simply writes the message to `System.err`

```
package servicetest.logger;

import servicetest.Logger;

public class ConsoleLogger implements Logger {

    @Override
    public void log(String message) {
        System.err.println(message);
    }

    @Override
    public void close() {
    }

}
```

The following implementation writes the messages to a text file:

```
package servicetest.logger;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
```

```
import servicetest.Logger;

public class FileLogger implements Logger {

    private final BufferedWriter writer;

    public FileLogger() throws IOException {
        writer = new BufferedWriter(new FileWriter("log.txt"));
    }

    @Override
    public void log(String message) throws IOException {
        writer.append(message);
        writer.newLine();
    }

    @Override
    public void close() throws IOException {
        writer.close();
    }

}
```

META-INF/services/servicetest.Logger

The `META-INF/services/servicetest.Logger` file lists the names of the `Logger` implementations.

```
servicetest.logger.ConsoleLogger
servicetest.logger.FileLogger
```

Usage

The following `main` method writes a message to all available loggers. The loggers are instantiated using `ServiceLoader`.

```
public static void main(String[] args) throws Exception {
    final String message = "Hello World!";

    // get ServiceLoader for Logger
    ServiceLoader<Logger> loader = ServiceLoader.load(servicetest.Logger.class);

    // iterate through instances of available loggers, writing the message to each one
    Iterator<Logger> iterator = loader.iterator();
    while (iterator.hasNext()) {
        try (Logger logger = iterator.next()) {
            logger.log(message);
        }
    }
}
```

Chapter 137: Classloaders

Section 137.1: Implementing a custom classLoader

Every custom loader must directly or indirectly extend the `java.lang.ClassLoader` class. The main *extension points* are the following methods:

- `findClass(String)` - overload this method if your classloader follows the standard delegation model for class loading.
- `loadClass(String, boolean)` - overload this method to implement an alternative delegation model.
- `findResource` and `findResources` - overload these methods to customize resource loading.

The `defineClass` methods which are responsible for actually loading the class from a byte array are **`final`** to prevent overloading. Any custom behavior needs to be performed prior to calling `defineClass`.

Here is a simple that loads a specific class from a byte array:

```
public class ByteArrayClassLoader extends ClassLoader {
    private String classname;
    private byte[] classfile;

    public ByteArrayClassLoader(String classname, byte[] classfile) {
        this.classname = classname;
        this.classfile = classfile.clone();
    }

    @Override
    protected Class findClass(String classname) throws ClassNotFoundException {
        if (classname.equals(this.classname)) {
            return defineClass(classname, classfile, 0, classfile.length);
        } else {
            throw new ClassNotFoundException(classname);
        }
    }
}
```

Since we have only overridden the `findClass` method, this custom class loader is going to behave as follows when `loadClass` is called.

1. The classloader's `loadClass` method calls `findLoadedClass` to see if a class with this name has already been loaded by this classloader. If that succeeds, the resulting **`Class`** object is returned to the requestor.
2. The `loadClass` method then delegates to the parent classloader by calling its `loadClass` call. If the parent can deal with the request, it will return a **`Class`** object which is then returned to the requestor.
3. If the parent classloader cannot load the class, `findClass` then calls our override `findClass` method, passing the name of the class to be loaded.
4. If the requested name matches **`this`**`.classname`, we call `defineClass` to load the actual class from the **`this`**`.classfile` byte array. The resulting **`Class`** object is then returned.
5. If the name did not match, we throw `ClassNotFoundException`.

Section 137.2: Loading an external .class file

To load a class we first need to define it. The class is defined by the `ClassLoader`. There's just one problem, Oracle didn't write the `ClassLoader`'s code with this feature available. To define the class we will need to access a method named `defineClass()` which is a private method of the `ClassLoader`.

To access it, what we will do is create a new class, `ByteClassLoader`, and extend it to `ClassLoader`. Now that we have extended our class to `ClassLoader`, we can access the `ClassLoader`'s private methods. To make `defineClass()` available, we will create a new method that will act like a mirror for the private `defineClass()` method. To call the private method we will need the class name, `name`, the class bytes, `classBytes`, the first byte's offset, which will be `0` because `classBytes`' data starts at `classBytes[0]`, and the last byte's offset, which will be `classBytes.lenght` because it represents the size of the data, which will be the last offset.

```
public class ByteClassLoader extends ClassLoader {

    public Class<?> defineClass(String name, byte[] classBytes) {
        return defineClass(name, classBytes, 0, classBytes.length);
    }

}
```

Now, we have a public `defineClass()` method. It can be called by passing the name of the class and the class bytes as arguments.

Let's say we have class named `MyClass` in the package `stackoverflow`...

To call the method we need the class bytes so we create a `Path` object representing our class' path by using the `Paths.get()` method and passing the path of the binary class as an argument. Now, we can get the class bytes with `Files.readAllBytes(path)`. So we create a `ByteClassLoader` instance and use the method we created, `defineClass()`. We already have the class bytes but to call our method we also need the class name which is given by the package name (dot) the class canonical name, in this case `stackoverflow.MyClass`.

```
Path path = Paths.get("MyClass.class");

ByteClassLoader loader = new ByteClassLoader();
loader.defineClass("stackoverflow.MyClass", Files.readAllBytes(path);
```

Note: The `defineClass()` method returns a `Class<?>` object. You can save it if you want.

To load the class, we just call `loadClass()` and pass the class name. This method can throw an `ClassNotFoundException` so we need to use a try cath block

```
try{
    loader.loadClass("stackoverflow.MyClass");
} catch(ClassNotFoundException e){
    e.printStackTrace();
}
```

Section 137.3: Instantiating and using a classloader

This basic example shows how an application can instantiate a classloader and use it to dynamically load a class.

```
URL[] urls = new URL[] {new URL("file:/home/me/extras.jar")};
Classloader loader = new URLClassLoader(urls);
Class<?> myObjectClass = loader.findClass("com.example.MyObject");
```

The classloader created in this example will have the default classloader as its parent, and will first try to find any class in the parent classloader before looking in "extra.jar". If the requested class has already been loaded, the `findClass` call will return the reference to the previously loaded class.

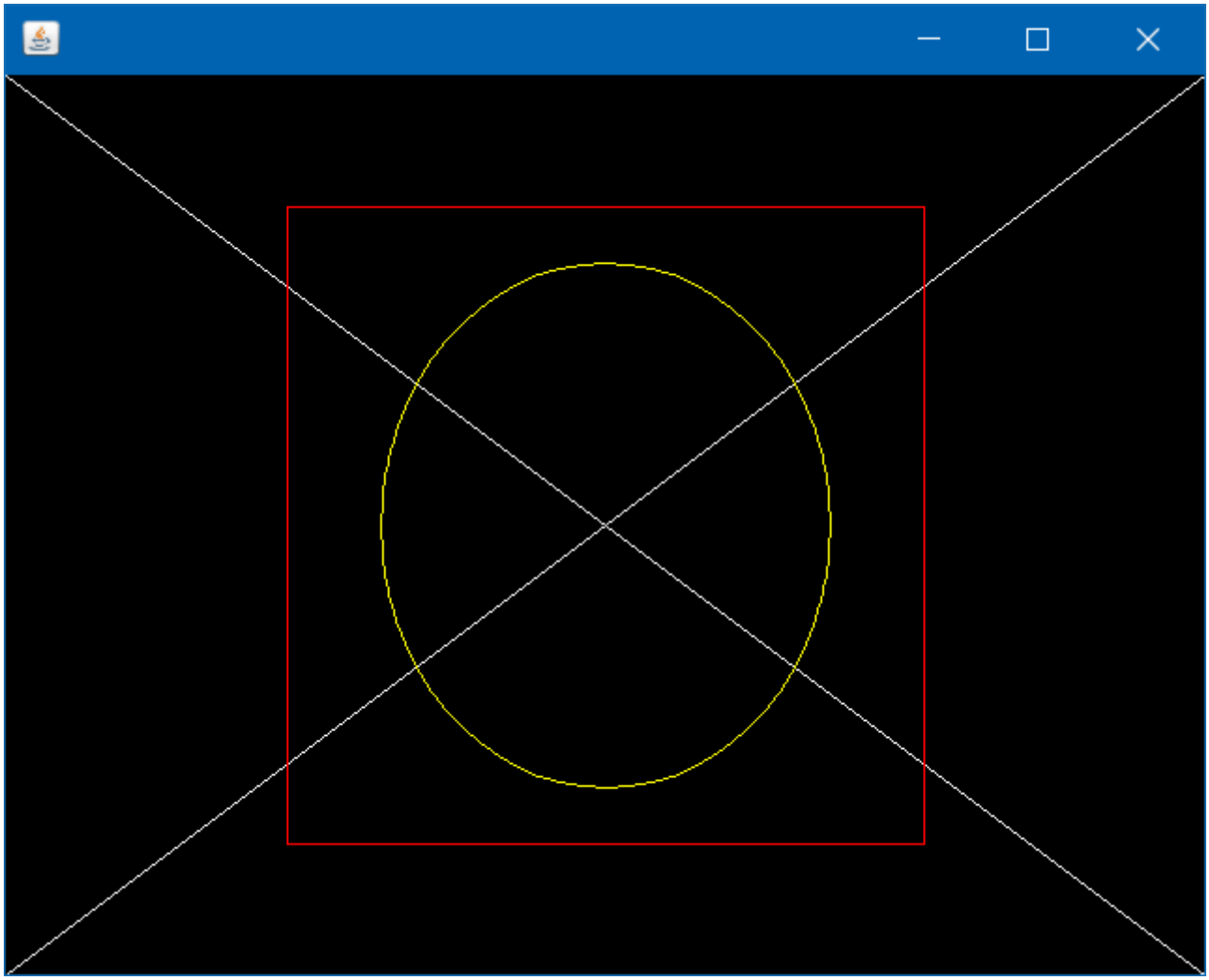
The `findClass` call can fail in a variety of ways. The most common are:

- If the named class cannot be found, the call with throw `ClassNotFoundException`.
- If the named class depends on some other class that cannot be found, the call will throw `NoClassDefFoundError`.

Chapter 138: Creating Images Programmatically

Section 138.1: Creating a simple image programmatically and displaying it

```
class ImageCreationExample {

    static Image createSampleImage() {
        // instantiate a new BufferedImage (subclass of Image) instance
        BufferedImage img = new BufferedImage(640, 480, BufferedImage.TYPE_INT_ARGB);

        //draw something on the image
        paintOnImage(img);

        return img;
    }

    static void paintOnImage(BufferedImage img) {
        // get a drawable Graphics2D (subclass of Graphics) object
        Graphics2D g2d = (Graphics2D) img.getGraphics();

        // some sample drawing
        g2d.setColor(Color.BLACK);
        g2d.fillRect(0, 0, 640, 480);
        g2d.setColor(Color.WHITE);
        g2d.drawLine(0, 0, 640, 480);
        g2d.drawLine(0, 480, 640, 0);
        g2d.setColor(Color.YELLOW);
        g2d.drawOval(200, 100, 240, 280);
        g2d.setColor(Color.RED);
        g2d.drawRect(150, 70, 340, 340);

        // drawing on images can be very memory-consuming
        // so it's better to free resources early
        // it's not necessary, though
        g2d.dispose();
    }

    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Image img = createSampleImage();
        ImageIcon icon = new ImageIcon(img);
        frame.add(new JLabel(icon));
        frame.pack();
        frame.setVisible(true);
    }
}
```



Section 138.2: Save an Image to disk

```
public static void saveImage(String destination) throws IOException {
    // method implemented in "Creating a simple image Programmatically and displaying it" example
    BufferedImage img = createSampleImage();

    // ImageIO provides several write methods with different outputs
    ImageIO.write(img, "png", new File(destination));
}
```

Section 138.3: Setting individual pixel's color in BufferedImage

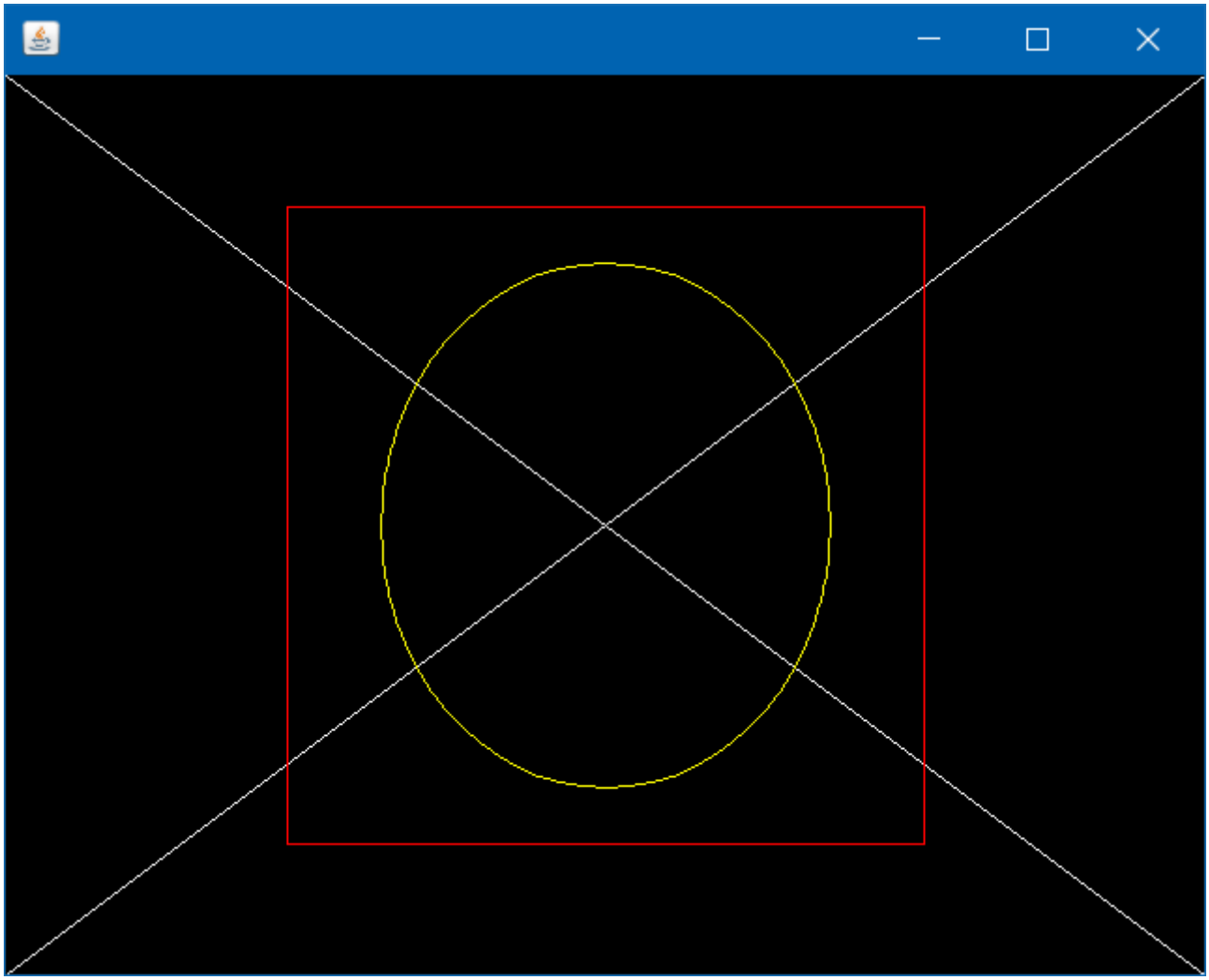
```
BufferedImage image = new BufferedImage(256, 256, BufferedImage.TYPE_INT_ARGB);

//you don't have to use the Graphics object, you can read and set pixel color individually
for (int i = 0; i < 256; i++) {
    for (int j = 0; j < 256; j++) {
        int alpha = 255; //don't forget this, or use BufferedImage.TYPE_INT_RGB instead
        int red = i; //or any formula you like
        int green = j; //or any formula you like
        int blue = 50; //or any formula you like
        int color = (alpha << 24) | (red << 16) | (green << 8) | blue;
        image.setRGB(i, j, color);
    }
}

ImageIO.write(image, "png", new File("computed.png"));
```

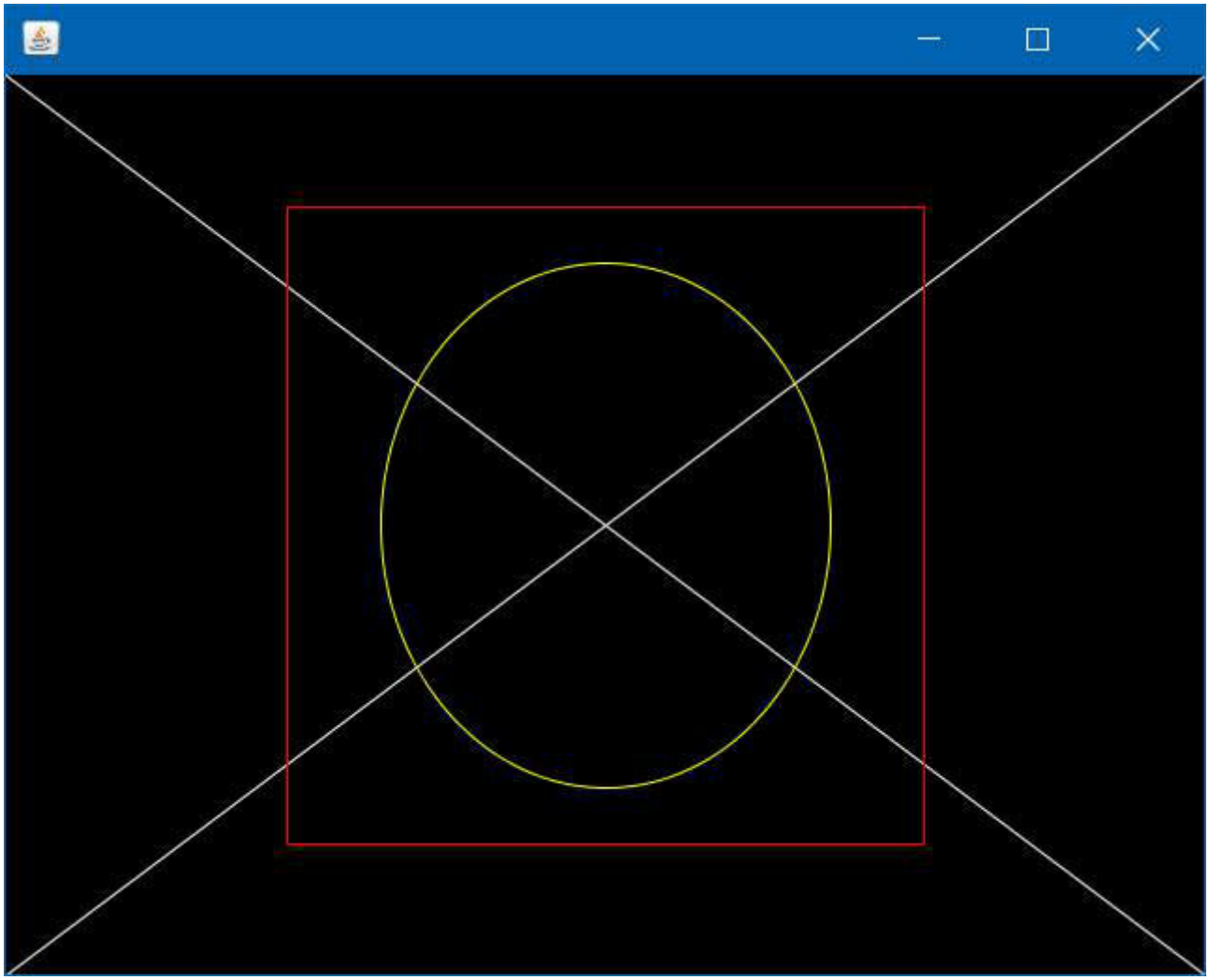
Output:

Section 138.4: Specifying image rendering quality

```
static void setupQualityHigh(Graphics2D g2d) {
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
    g2d.setRenderingHint(RenderingHints.KEY_RENDERING, RenderingHints.VALUE_RENDER_QUALITY);
    // many other RenderingHints KEY/VALUE pairs to specify
}


static void setupQualityLow(Graphics2D g2d) {
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_OFF);
    g2d.setRenderingHint(RenderingHints.KEY_RENDERING, RenderingHints.VALUE_RENDER_SPEED);
}
```

A comparison of QUALITY and SPEED rendering of the sample image:





Section 138.5: Creating an image with BufferedImage class

```
int width = 256; //in pixels
int height = 256; //in pixels
BufferedImage image = new BufferedImage(width, height, BufferedImage.TYPE_4BYTE_ABGR);
//BufferedImage.TYPE_4BYTE_ABGR - store RGB color and visibility (alpha), see javadoc for more info

Graphics g = image.createGraphics();

//draw whatever you like, like you would in a drawComponent(Graphics g) method in an UI application
g.setColor(Color.RED);
g.fillRect(20, 30, 50, 50);

g.setColor(Color.BLUE);
g.drawOval(120, 120, 80, 40);

g.dispose(); //dispose graphics objects when they are no longer needed

//now image has programmatically generated content, you can use it in graphics.drawImage() to draw it
somewhere else
//or just simply save it to a file
ImageIO.write(image, "png", new File("myimage.png"));
```

Output:

Section 138.6: Editing and re-using image with BufferedImage

```
BufferedImage cat = ImageIO.read(new File("cat.jpg")); //read existing file

//modify it
Graphics g = cat.createGraphics();
g.setColor(Color.RED);
g.drawString("Cat", 10, 10);
g.dispose();

//now create a new image
BufferedImage cats = new BufferedImage(256, 256, BufferedImage.TYPE_4BYTE_ABGR);

//and draw the old one on it, 16 times
g = cats.createGraphics();
for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 4; j++) {
        g.drawImage(cat, i * 64, j * 64, null);
    }
}

g.setColor(Color.BLUE);
g.drawRect(0, 0, 255, 255); //add some nice border
g.dispose(); //and done

ImageIO.write(cats, "png", new File("cats.png"));
```

Original cat file:

Produced file:

Section 138.7: How to scale a BufferedImage

```
 /**
 * Resizes an image using a Graphics2D object backed by a BufferedImage.
 * @param srcImg - source image to scale
 * @param w - desired width
 * @param h - desired height
 * @return - the new resized image
 */
private BufferedImage getScaledImage(Image srcImg, int w, int h){

    //Create a new image with good size that contains or might contain arbitrary alpha values between
and including 0.0 and 1.0.
    BufferedImage resizedImg = new BufferedImage(w, h, BufferedImage.TRANSLUCENT);

    //Create a device-independant object to draw the resized image
    Graphics2D g2 = resizedImg.createGraphics();

    //This could be changed, Cf.
http://stackoverflow.com/documentation/java/5482/creating-images-programmatically/19498/specifying-im
age-rendering-quality
    g2.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
RenderingHints.VALUE_INTERPOLATION_BILINEAR);

    //Finally draw the source image in the Graphics2D with the desired size.
    g2.drawImage(srcImg, 0, 0, w, h, null);

    //Disposes of this graphics context and releases any system resources that it is using
    g2.dispose();

    //Return the image used to create the Graphics2D
    return resizedImg;
}
```

Chapter 139: Atomic Types

Parameter	Description
set	Volatile set of the field
get	Volatile read of the field
lazySet	This is a store ordered operation of the field
compareAndSet	If the value is the expeed value then sent it to the new value
getAndSet	get the current value and update

Java Atomic Types are simple mutable types that provide basic operations that are thread-safe and atomic without resorting to locking. They are intended for use in cases where locking would be a concurrency bottleneck, or where there is risk of deadlock or livelock.

Section 139.1: Creating Atomic Types

For simple multi-threaded code, using synchronization is acceptable. However, using synchronization does have a liveness impact, and as a codebase becomes more complex, the likelihood goes up that you will end up with Deadlock, Starvation, or Livelock.

In cases of more complex concurrency, using Atomic Variables is often a better alternative, as it allows an individual variable to be accessed in a thread-safe manner without the overhead of using synchronized methods or code blocks.

Creating an `AtomicInteger` type:

```
AtomicInteger aInt = new AtomicInteger() // Create with default value 0

AtomicInteger aInt = new AtomicInteger(1) // Create with initial value 1
```

Similarly for other instance types.

```
AtomicIntegerArray aIntArray = new AtomicIntegerArray(10) // Create array of specific length
AtomicIntegerArray aIntArray = new AtomicIntegerArray(new int[] {1, 2, 3}) // Initialize array with
another array
```

Similarly for other atomic types.

There is a notable exception that there is no **`float`** and **`double`** types. These can be simulated through the use of `Float.floatToIntBits(float)` and `Float.intBitsToFloat(int)` for **`float`** as well as `Double.doubleToLongBits(double)` and `Double.longBitsToDouble(long)` for doubles.

If you are willing to use `sun.misc.Unsafe` you can use any primitive variable as atomic by using the atomic operation in `sun.misc.Unsafe`. All primitive types should be converted or encoded in int or longs to so use it in this way. For more on this see: sun.misc.Unsafe.

Section 139.2: Motivation for Atomic Types

The simple way to implement multi-threaded applications is to use Java's built-in synchronization and locking primitives; e.g. the **`synchronized`** keyword. The following example shows how we might use **`synchronized`** to accumulate counts.

```
public class Counters {
    private final int[] counters;
```

```
    public Counters(int nosCounters) {
        counters = new int[nosCounters];
    }

    /**
     * Increments the integer at the given index
     */
    public synchronized void count(int number) {
        if (number >= 0 && number < counters.length) {
            counters[number]++;
        }
    }

    /**
     * Obtains the current count of the number at the given index,
     * or if there is no number at that index, returns 0.
     */
    public synchronized int getCount(int number) {
        return (number >= 0 && number < counters.length) ? counters[number] : 0;
    }
}
```

This implementation will work correctly. However, if you have a large number of threads making lots of simultaneous calls on the same `Counters` object, the synchronization is liable to be a bottleneck. Specifically:

1. Each **`synchronized`** method call will start with the current thread acquiring the lock for the `Counters` instance.
2. The thread will hold the lock while it checks `number` value and updates the counter.
3. Finally, the it will release the lock, allowing other threads access.

If one thread attempts to acquire the lock while another one holds it, the attempting thread will be blocked (stopped) at step 1 until the lock is released. If multiple threads are waiting, one of them will get it, and the others will continue to be blocked.

This can lead to a couple of problems:

- If there is a lot of *contention* for the lock (i.e. lots of thread try to acquire it), then some threads can be blocked for a long time.
- When a thread is blocked waiting for the lock, the operating system will typically try switch execution to a different thread. This *context switching* incurs a relatively large performance impact on the processor.
- When there are multiple threads blocked on the same lock, there are no guarantees that any one of them will be treated "fairly" (i.e. each thread is guaranteed to be scheduled to run). This can lead to *thread starvation*.

How does one implement Atomic Types?

Let us start by rewriting the example above using `AtomicInteger` counters:

```
public class Counters {
    private final AtomicInteger[] counters;

    public Counters(int nosCounters) {
        counters = new AtomicInteger[nosCounters];
        for (int i = 0; i < nosCounters; i++) {
            counters[i] = new AtomicInteger();
        }
```

```
    }

    /**
     * Increments the integer at the given index
     */
    public void count(int number) {
        if (number >= 0 && number < counters.length) {
            counters[number].incrementAndGet();
        }
    }

    /**
     * Obtains the current count of the object at the given index,
     * or if there is no number at that index, returns 0.
     */
    public int getCount(int number) {
        return (number >= 0 && number < counters.length) ?
                counters[number].get() : 0;
    }
}
```

We have replaced the `int[]` with an `AtomicInteger[]`, and initialized it with an instance in each element. We have also added calls to `incrementAndGet()` and `get()` in place of operations on `int` values.

But the most important thing is that we can remove the `synchronized` keyword because locking is no longer required. This works because the `incrementAndGet()` and `get()` operations are *atomic* and *thread-safe*. In this context, it means that:

- Each counter in the array will only be *observable* in the either the "before" state for an operation (like an "increment") or in the "after" state.
- Assuming that the operation occurs at time `T`, no thread will be able to see the "before" state after time `T`.

Furthermore, while two threads might actually attempt to update the same `AtomicInteger` instance at the same time, the implementations of the operations ensure that only one increment happens at a time on the given instance. This is done without locking, often resulting in better performance.

How do Atomic Types work?

Atomic types typically rely on specialized hardware instructions in the instruction set of the target machine. For example, Intel-based instruction sets provide a `CAS` (Compare and Swap) instruction that will perform a specific sequence of memory operations atomically.

These low-level instructions are are used to implement higher-level operations in the APIs of the respective `AtomicXxx` classes. For example, (again, in C-like pseudocode):

```
private volatile num;

int increment() {
  while (TRUE) {
    int old = num;
    int new = old + 1;
    if (old == compare_and_swap(&num, old, new)) {
      return new;
    }
  }
}
```

If there is no contention on the `AtomicXxxx`, the `if` test will succeed and the loop will end immediately. If there is contention, then the `if` will fail for all but one of the threads, and they will "spin" in the loop for a small number of cycles of the loop. In practice, the spinning is orders of magnitude faster (except at *unrealistically high* levels of contention, where synchronized performs better than atomic classes because when the CAS operation fails, then the retry will only add more contention) than suspending the thread and switching to another one.

Incidentally, CAS instructions are typically used by the JVM to implement *uncontended locking*. If the JVM can see that a lock is not currently locked, it will attempt to use a CAS to acquire the lock. If the CAS succeeds, then there is no need to do the expensive thread scheduling, context switching and so on. For more information on the techniques used, see Biased Locking in HotSpot.

Chapter 140: RSA Encryption

Section 140.1: An example using a hybrid cryptosystem consisting of OAEP and GCM

The following example encrypts data by using a hybrid cryptosystem consisting of AES GCM and OAEP, using their default parameter sizes and an AES key size of 128 bits.

OAEP is less vulnerable to padding oracle attacks than PKCS#1 v1.5 padding. GCM is also protected against padding oracle attacks.

Decryption can be performed by first retrieving the length of the encapsulated key and then by retrieving the encapsulated key. The encapsulated key can then be decrypted using the RSA private key that forms a key pair with the public key. After that the AES/GCM encrypted ciphertext can be decrypted to the original plaintext.

The protocol consists of:

1. a length field for the wrapped key (`RSAPrivateKey` misses a `getKeySize()` method);
2. the wrapped/encapsulated key, of the same size as the RSA key size in bytes;
3. the GCM ciphertext and 128 bit authentication tag (automatically added by Java).

Notes:

- To correctly use this code you should supply an RSA key of at least 2048 bits, bigger is better (but slower, especially during decryption);
- To use AES-256 you should install the unlimited cryptography policy files first;
- Instead creating your own protocol you might want to use a container format such as the Cryptographic Message Syntax (CMS / PKCS#7) or PGP instead.

So here's the example:

```
/**
 * Encrypts the data using a hybrid crypto-system which uses GCM to encrypt the data and OAEP to
encrypt the AES key.
 * The key size of the AES encryption will be 128 bit.
 * All the default parameter choices are used for OAEP and GCM.
 *
 * @param publicKey the RSA public key used to wrap the AES key
 * @param plaintext the plaintext to be encrypted, not altered
 * @return the ciphertext
 * @throws InvalidKeyException if the key is not an RSA public key
 * @throws NullPointerException if the plaintext is null
 */
public static byte[] encryptData(PublicKey publicKey, byte[] plaintext)
        throws InvalidKeyException, NullPointerException {

    // --- create the RSA OAEP cipher ---

    Cipher oaep;
    try {
        // SHA-1 is the default and not vulnerable in this setting
        // use OAEPParameterSpec to configure more than just the hash
        oaep = Cipher.getInstance("RSA/ECB/OAEPwithSHA1andMGF1Padding");
    } catch (NoSuchAlgorithmException e) {
        throw new RuntimeException(
                "Runtime doesn't have support for RSA cipher (mandatory algorithm for runtimes)",
e);
```

```
    } catch (NoSuchPaddingException e) {
        throw new RuntimeException(
                "Runtime doesn't have support for OAEP padding (present in the standard Java
runtime sinze XX)", e);
    }
    oaep.init(Cipher.WRAP_MODE, publicKey);

    // --- wrap the plaintext in a buffer

    // will throw NullPointerException if plaintext is null
    ByteBuffer plaintextBuffer = ByteBuffer.wrap(plaintext);

    // --- generate a new AES secret key ---

    KeyGenerator aesKeyGenerator;
    try {
        aesKeyGenerator = KeyGenerator.getInstance("AES");
    } catch (NoSuchAlgorithmException e) {
        throw new RuntimeException(
                "Runtime doesn't have support for AES key generator (mandatory algorithm for
runtimes)", e);
    }
    // for AES-192 and 256 make sure you've got the rights (install the
    // Unlimited Crypto Policy files)
    aesKeyGenerator.init(128);
    SecretKey aesKey = aesKeyGenerator.generateKey();

    // --- wrap the new AES secret key ---

    byte[] wrappedKey;
    try {
        wrappedKey = oaep.wrap(aesKey);
    } catch (IllegalBlockSizeException e) {
        throw new RuntimeException(
                "AES key should always fit OAEP with normal sized RSA key", e);
    }

    // --- setup the AES GCM cipher mode ---

    Cipher aesGCM;
    try {
        aesGCM = Cipher.getInstance("AES/GCM/Nopadding");
        // we can get away with a zero nonce since the key is randomly generated
        // 128 bits is the recommended (maximum) value for the tag size
        // 12 bytes (96 bits) is the default nonce size for GCM mode encryption
        GCMParameterSpec staticParameterSpec = new GCMParameterSpec(128, new byte[12]);
        aesGCM.init(Cipher.ENCRYPT_MODE, aesKey, staticParameterSpec);
    } catch (NoSuchAlgorithmException e) {
        throw new RuntimeException(
                "Runtime doesn't have support for AES cipher (mandatory algorithm for runtimes)",
e);
    } catch (NoSuchPaddingException e) {
        throw new RuntimeException(
                "Runtime doesn't have support for GCM (present in the standard Java runtime sinze
XX)", e);
    } catch (InvalidAlgorithmParameterException e) {
        throw new RuntimeException(
                "IvParameterSpec not accepted by this implementation of GCM", e);
    }

    // --- create a buffer of the right size for our own protocol ---
```

```
    ByteBuffer ciphertextBuffer = ByteBuffer.allocate(
            Short.BYTES
            + oaep.getOutputSize(128 / Byte.SIZE)
            + aesGCM.getOutputSize(plaintext.length));

    // - element 1: make sure that we know the size of the wrapped key
    ciphertextBuffer.putShort((short) wrappedKey.length);

    // - element 2: put in the wrapped key
    ciphertextBuffer.put(wrappedKey);

    // - element 3: GCM encrypt into buffer
    try {
        aesGCM.doFinal(plaintextBuffer, ciphertextBuffer);
    } catch (ShortBufferException | IllegalBlockSizeException | BadPaddingException e) {
        throw new RuntimeException("Cryptographic exception, AES/GCM encryption should not fail
here", e);
    }

    return ciphertextBuffer.array();
}
```

Of course, encryption is not very useful without decryption. Note that this will return minimal information if decryption fails.

```
/**
 * Decrypts the data using a hybrid crypto-system which uses GCM to encrypt
 * the data and OAEP to encrypt the AES key. All the default parameter
 * choices are used for OAEP and GCM.
 *
 * @param privateKey
 *            the RSA private key used to unwrap the AES key
 * @param ciphertext
 *            the ciphertext to be encrypted, not altered
 * @return the plaintext
 * @throws InvalidKeyException
 *             if the key is not an RSA private key
 * @throws NullPointerException
 *             if the ciphertext is null
 * @throws IllegalArgumentException
 *             with the message "Invalid ciphertext" if the ciphertext is invalid (minimize
information leakage)
 */
public static byte[] decryptData(PrivateKey privateKey, byte[] ciphertext)
        throws InvalidKeyException, NullPointerException {

    // --- create the RSA OAEP cipher ---

    Cipher oaep;
    try {
        // SHA-1 is the default and not vulnerable in this setting
        // use OAEPParameterSpec to configure more than just the hash
        oaep = Cipher.getInstance("RSA/ECB/OAEPwithSHA1andMGF1Padding");
    } catch (NoSuchAlgorithmException e) {
        throw new RuntimeException(
                "Runtime doesn't have support for RSA cipher (mandatory algorithm for runtimes)",
                e);
    } catch (NoSuchPaddingException e) {
        throw new RuntimeException(
                "Runtime doesn't have support for OAEP padding (present in the standard Java
runtime sinze XX)",
```

```
                e);
        }
        oaep.init(Cipher.UNWRAP_MODE, privateKey);

        // --- wrap the ciphertext in a buffer

        // will throw NullPointerException if ciphertext is null
        ByteBuffer ciphertextBuffer = ByteBuffer.wrap(ciphertext);

        // sanity check #1
        if (ciphertextBuffer.remaining() < 2) {
            throw new IllegalArgumentException("Invalid ciphertext");
        }
        // - element 1: the length of the encapsulated key
        int wrappedKeySize = ciphertextBuffer.getShort() & 0xFFFF;
        // sanity check #2
        if (ciphertextBuffer.remaining() < wrappedKeySize + 128 / Byte.SIZE) {
            throw new IllegalArgumentException("Invalid ciphertext");
        }

        // --- unwrap the AES secret key ---

        byte[] wrappedKey = new byte[wrappedKeySize];
        // - element 2: the encapsulated key
        ciphertextBuffer.get(wrappedKey);
        SecretKey aesKey;
        try {
            aesKey = (SecretKey) oaep.unwrap(wrappedKey, "AES",
                    Cipher.SECRET_KEY);
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(
                    "Runtime doesn't have support for AES cipher (mandatory algorithm for runtimes)",
                    e);
        } catch (InvalidKeyException e) {
            throw new RuntimeException(
                    "Invalid ciphertext");
        }

        // --- setup the AES GCM cipher mode ---

        Cipher aesGCM;
        try {
            aesGCM = Cipher.getInstance("AES/GCM/Nopadding");
            // we can get away with a zero nonce since the key is randomly
            // generated
            // 128 bits is the recommended (maximum) value for the tag size
            // 12 bytes (96 bits) is the default nonce size for GCM mode
            // encryption
            GCMParameterSpec staticParameterSpec = new GCMParameterSpec(128,
                    new byte[12]);
            aesGCM.init(Cipher.DECRYPT_MODE, aesKey, staticParameterSpec);
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(
                    "Runtime doesn't have support for AES cipher (mandatory algorithm for runtimes)",
                    e);
        } catch (NoSuchPaddingException e) {
            throw new RuntimeException(
                    "Runtime doesn't have support for GCM (present in the standard Java runtime sinze
XX)",
                    e);
        } catch (InvalidAlgorithmParameterException e) {
            throw new RuntimeException(
```

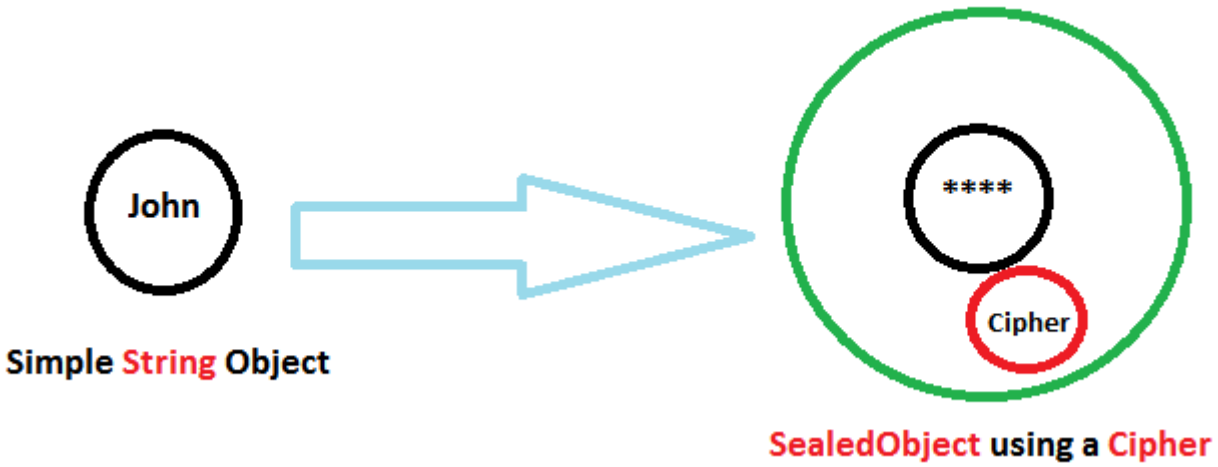
```
                "IvParameterSpec not accepted by this implementation of GCM",
                e);
    }

    // --- create a buffer of the right size for our own protocol ---

    ByteBuffer plaintextBuffer = ByteBuffer.allocate(aesGCM
            .getOutputSize(ciphertextBuffer.remaining()));

    // - element 3: GCM ciphertext
    try {
        aesGCM.doFinal(ciphertextBuffer, plaintextBuffer);
    } catch (ShortBufferException | IllegalBlockSizeException
            | BadPaddingException e) {
        throw new RuntimeException(
                "Invalid ciphertext");
    }

    return plaintextBuffer.array();
}
```

Chapter 141: Secure objects

Section 141.1: SealedObject (javax.crypto.SealedObject)

This class enables a programmer to create an object and protect its confidentiality with a cryptographic algorithm.

Given any Serializable object, one can create a **SealedObject** that encapsulates the original object, in serialized format (i.e., a "deep copy"), and seals (encrypts) its serialized contents, using a cryptographic algorithm such as AES, DES, to protect its confidentiality. The encrypted content can later be decrypted (with the corresponding algorithm using the correct decryption key) and de-serialized, yielding the original object.

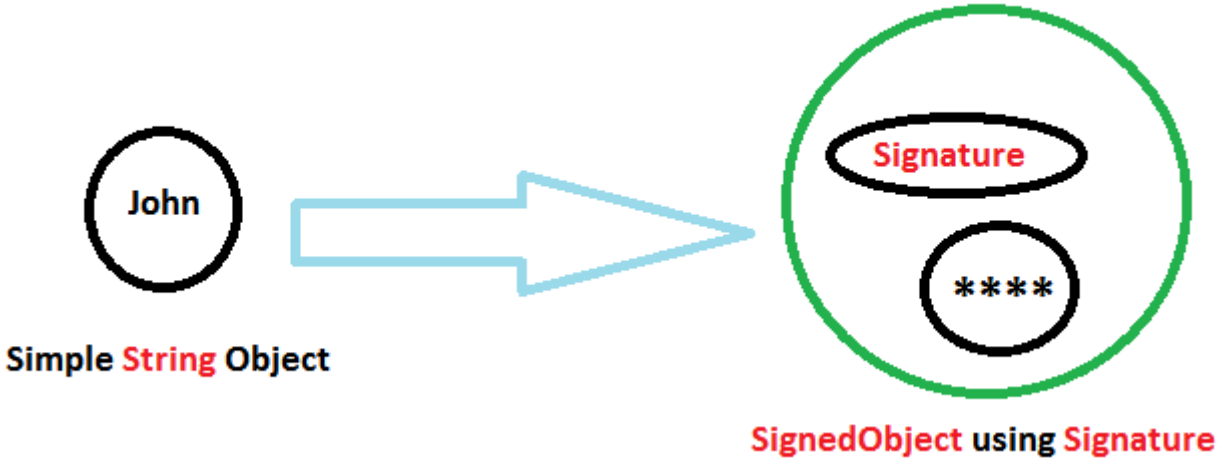



```
Serializable obj = new String("John");
// Generate key
KeyGenerator kgen = KeyGenerator.getInstance("AES");
kgen.init(128);
SecretKey aesKey = kgen.generateKey();
Cipher cipher = Cipher.getInstance("AES");
cipher.init(Cipher.ENCRYPT_MODE, aesKey);
SealedObject sealedObject = new SealedObject(obj, cipher);
System.out.println("sealedObject-" + sealedObject);
System.out.println("sealedObject Data-" + sealedObject.getObject(aesKey));
```

Section 141.2: SignedObject (java.security.SignedObject)

SignedObject is a class for the purpose of creating authentic runtime objects whose integrity cannot be compromised without being detected.

More specifically, a SignedObject contains another Serializable object, the (to-be-)signed object and its signature.




```
//Create a key
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA", "SUN");
SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");
keyGen.initialize(1024, random);
// create a private key
PrivateKey signingKey = keyGen.generateKeyPair().getPrivate();
// create a Signature
Signature signingEngine = Signature.getInstance("DSA");
signingEngine.initSign(signingKey);
// create a simple object
Serializable obj = new String("John");
// sign our object
SignedObject signedObject = new SignedObject(obj, signingKey, signingEngine);

System.out.println("signedObject-" + signedObject);
System.out.println("signedObject Data-" + signedObject.getObject());
```

Chapter 142: Security & Cryptography

Section 142.1: Compute Cryptographic Hashes

To compute the hashes of relatively small blocks of data using different algorithms:

```
final MessageDigest md5 = MessageDigest.getInstance("MD5");
final MessageDigest sha1 = MessageDigest.getInstance("SHA-1");
final MessageDigest sha256 = MessageDigest.getInstance("SHA-256");

final byte[] data = "FOO BAR".getBytes();

System.out.println("MD5    hash: " + DatatypeConverter.printHexBinary(md5.digest(data)));
System.out.println("SHA1   hash: " + DatatypeConverter.printHexBinary(sha1.digest(data)));
System.out.println("SHA256 hash: " + DatatypeConverter.printHexBinary(sha256.digest(data)));
```

Produces this output:

```
MD5    hash: E99E768582F6DD5A3BA2D9C849DF736E
SHA1   hash: 0135FAA6323685BA8A8FF8D3F955F0C36949D8FB
SHA256 hash: 8D35C97BCD902B96D1B551741BBE8A7F50BB5A690B4D0225482EAA63DBFB9DED
```

Additional algorithms may be available depending on your implementation of the Java platform.

Section 142.2: Encrypt and Decrypt Data with Public / Private Keys

To encrypt data with a public key:

```
final Cipher rsa = Cipher.getInstance("RSA");

rsa.init(Cipher.ENCRYPT_MODE, keyPair.getPublic());
rsa.update(message.getBytes());
final byte[] result = rsa.doFinal();

System.out.println("Message: " + message);
System.out.println("Encrypted: " + DatatypeConverter.printHexBinary(result));
```

Produces output similar to:

```
Message: Hello
Encrypted: 5641FBB9558ECFA9ED...
```

Note that when creating the `Cipher` object, you have to specify a transformation that is compatible with the type of key being used. (See JCA Standard Algorithm Names for a list of supported transformations.). For RSA encryption data `message.getBytes()` length must be smaller than the key size. See this SO Answer for detail.

To decrypt the data:

```
final Cipher rsa = Cipher.getInstance("RSA");

rsa.init(Cipher.DECRYPT_MODE, keyPair.getPrivate());
rsa.update(cipherText);
final String result = new String(rsa.doFinal());
```

```
System.out.println("Decrypted: " + result);
```

Produces the following output:

```
Decrypted: Hello
```

Section 142.3: Generate Cryptographically Random Data

To generate samples of cryptographically random data:

```
final byte[] sample = new byte[16];

new SecureRandom().nextBytes(sample);

System.out.println("Sample: " + DatatypeConverter.printHexBinary(sample));
```

Produces output similar to:

```
Sample: E4F14CEA2384F70B706B53A6DF8C5EFE
```

Note that the call to `nextBytes()` may block while entropy is gathered depending on the algorithm being used.

To specify the algorithm and provider:

```
final byte[] sample = new byte[16];
final SecureRandom randomness = SecureRandom.getInstance("SHA1PRNG", "SUN");

randomness.nextBytes(sample);

System.out.println("Provider: " + randomness.getProvider());
System.out.println("Algorithm: " + randomness.getAlgorithm());
System.out.println("Sample: " + DatatypeConverter.printHexBinary(sample));
```

Produces output similar to:

```
Provider: SUN version 1.8
Algorithm: SHA1PRNG
Sample: C80C44BAEB352FD29FBBE20489E4C0B9
```

Section 142.4: Generate Public / Private Key Pairs

To generate key pairs using different algorithms and key sizes:

```
final KeyPairGenerator dhGenerator = KeyPairGenerator.getInstance("DiffieHellman");
final KeyPairGenerator dsaGenerator = KeyPairGenerator.getInstance("DSA");
final KeyPairGenerator rsaGenerator = KeyPairGenerator.getInstance("RSA");

dhGenerator.initialize(1024);
dsaGenerator.initialize(1024);
rsaGenerator.initialize(2048);

final KeyPair dhPair = dhGenerator.generateKeyPair();
final KeyPair dsaPair = dsaGenerator.generateKeyPair();
final KeyPair rsaPair = rsaGenerator.generateKeyPair();
```

Additional algorithms and key sizes may be available on your implementation of the Java platform.

To specify a source of randomness to use when generating the keys:

```
final KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA");

generator.initialize(2048, SecureRandom.getInstance("SHA1PRNG", "SUN"));

final KeyPair pair = generator.generateKeyPair();
```

Section 142.5: Compute and Verify Digital Signatures

To compute a signature:

```
final PrivateKey privateKey = keyPair.getPrivate();
final byte[] data = "FOO BAR".getBytes();
final Signature signer = Signature.getInstance("SHA1withRSA");

signer.initSign(privateKey);
signer.update(data);

final byte[] signature = signer.sign();
```

Note that the signature algorithm must be compatible with the algorithm used to generate the key pair.

To verify a signature:

```
final PublicKey publicKey = keyPair.getPublic();
final Signature verifier = Signature.getInstance("SHA1withRSA");

verifier.initVerify(publicKey);
verifier.update(data);

System.out.println("Signature: " + verifier.verify(signature));
```

Produces this output:

```
Signature: true
```

Chapter 143: Security & Cryptography

Security practices in Java can be separated into two broad, vaguely defined categories; Java platform security, and secure Java programming.

Java platform security practices deal with managing the security and integrity of the JVM. It includes such topics as managing JCE providers and security policies.

Secure Java programming practices concern the best ways to write secure Java programs. It includes such topics as using random numbers and cryptography, and preventing vulnerabilities.

Section 143.1: The JCE

The Java Cryptography Extension (JCE) is a framework built into the JVM to allow developers to easily and securely use cryptography in their programs. It does this by providing a simple, portable interface to programmers, while using a system of JCE Providers to securely implement the underlying cryptographic operations.

Section 143.2: Keys and Key Management

While the JCE secures cryptographic operations and key generation, it is up to the developer to actually manage their keys. More information needs to be provided here.

One commonly-accepted best practice for handling keys at runtime is to store them only as `byte` arrays, and never as strings. This is because Java strings are immutable, and cannot be manually "cleared" or "zeroed out" in memory; while a reference to a string can be removed, the exact string will remain in memory until its segment of memory is garbage-collected and reused. An attacker would have a large window in which they could dump the program's memory and easily find the key. Contrarily, `byte` arrays are mutable, and can have their contents overwritten in place; it is a good idea to 'zero-out' your keys as soon as you no longer need them.

Section 143.3: Common Java vulnerabilities

Section 143.4: Networking Concerns

Section 143.5: Randomness and You

For most applications, the `java.utils.Random` class is a perfectly fine source of "random" data. If you need to choose a random element from an array, or generate a random string, or create a temporary "unique" identifier, you should probably use `Random`.

However, many cryptographic systems rely on randomness for their security, and the randomness provided by `Random` simply isn't of high enough quality. For any cryptographic operation that requires a random input, you should use `SecureRandom` instead.

Section 143.6: Hashing and Validation

A cryptographic hash function is a member of a class of functions with three vital properties; consistency, uniqueness, and irreversibility.

Consistency: Given the same data, a hash function will always return the same value. That is, if X = Y, f(x) will always equal f(y) for hash function f.

Uniqueness: No two inputs to a hash function will ever result in the same output. That is, if X != Y, f(x) != f(y), for any

values of X and Y.

Irreversibility: It is impractically difficult, if not impossible, to "reverse" a hash function. That is, given only f(X), there should be no way of finding the original X short of putting every possible value of X through the function f (brute-force). There should be no function f1 such that f1(f(X)) = X.

Many functions lack at least one of these attributes. For example, MD5 and SHA1 are known to have collisions, i.e. two inputs that have the same output, so they lack uniqueness. Some functions that are currently believed to be secure are SHA-256 and SHA-512.

Chapter 144: SecurityManager

Section 144.1: Sandboxing classes loaded by a ClassLoader

The ClassLoader needs to provide a `ProtectionDomain` identifying the source of the code:

```
public class PluginClassLoader extends ClassLoader {
    private final ClassProvider provider;

    private final ProtectionDomain pd;


    public PluginClassLoader(ClassProvider provider) {
        this.provider = provider;
        Permissions permissions = new Permissions();

        this.pd = new ProtectionDomain(provider.getCodeSource(), permissions, this, null);
    }


    @Override
    protected Class<?> findClass(String name) throws ClassNotFoundException {
        byte[] classDef = provider.getClass(name);
        Class<?> clazz = defineClass(name, classDef, 0, classDef.length, pd);
        return clazz;
    }
}
```

By overriding `findClass` rather than `loadClass` the delegational model is preserved, and the PluginClassLoader will first query the system and parent classloader for class definitions.

Create a Policy:

```
public class PluginSecurityPolicy extends Policy {
    private final Permissions appPermissions = new Permissions();
    private final Permissions pluginPermissions = new Permissions();

    public PluginSecurityPolicy() {
        // amend this as appropriate
        appPermissions.add(new AllPermission());
        // add any permissions plugins should have to pluginPermissions
    }

    @Override
    public Provider getProvider() {
        return super.getProvider();
    }

    @Override
    public String getType() {
        return super.getType();
    }

    @Override
    public Parameters getParameters() {
        return super.getParameters();
    }

    @Override
```

```
    public PermissionCollection getPermissions(CodeSource codesource) {
        return new Permissions();
    }

    @Override
    public PermissionCollection getPermissions(ProtectionDomain domain) {
        return isPlugin(domain)?pluginPermissions:appPermissions;
    }

    private boolean isPlugin(ProtectionDomain pd){
        return pd.getClassLoader() instanceof PluginClassLoader;
    }

}
```

Finally, set the policy and a SecurityManager (default implementation is fine):

```
    Policy.setPolicy(new PluginSecurityPolicy());
    System.setSecurityManager(new SecurityManager());
```

Section 144.2: Enabling the SecurityManager

Java Virtual Machines (JVMs) can be run with a SecurityManager installed. The SecurityManager governs what the code running in the JVM is allowed to do, based on factors such as where the code was loaded from and what certificates were used to sign the code.

The SecurityManager can be installed by setting the java.security.manager system property on the command line when starting the JVM:

```
java -Djava.security.manager <main class name>
```

or programmatically from within Java code:

```
System.setSecurityManager(new SecurityManager())
```

The standard Java SecurityManager grants permissions on the basis of a Policy, which is defined in a policy file. If no policy file is specified, the default policy file under `$JAVA_HOME/lib/security/java.policy` will be used.

Section 144.3: Implementing policy deny rules

It is occasionally desirable to *deny* a certain `Permission` to some `ProtectionDomain`, *regardless* of any other permissions that domain accrues. This example demonstrates just one of all the possible approaches for satisfying this kind of requirement. It introduces a "negative" permission class, along with a wrapper that enables the default `Policy` to be reused as a repository of such permissions.

Notes:

- The standard policy file syntax and mechanism for permission assignment in general remain unaffected. This means that *deny* rules within policy files are still expressed as *grants*.
- The policy wrapper is meant to specifically encapsulate the default file-backed `Policy` (assumed to be `com.sun.security.provider.PolicyFile`).
- Denied permissions are only processed as such at the policy level. If statically assigned to a domain, they will by default be treated by that domain as ordinary "positive" permissions.

The `DeniedPermission` class

```
package com.example;
```

```
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Modifier;
import java.security.BasicPermission;
import java.security.Permission;
import java.security.UnresolvedPermission;
import java.text.MessageFormat;

/**
 * A representation of a "negative" privilege.
 * <p>
 * A <code>DeniedPermission</code>, when "granted" (to some <code>ProtectionDomain</code> and/or
 * <code>Principal</code>), represents a privilege which <em>cannot</em> be exercised, regardless of
 * any positive permissions (<code>AllPermission</code> included) possessed. In other words, if a
 * set of granted permissions, <em>P</em>, contains a permission of this class, <em>D</em>, then the
 * set of effectively granted permissions is<br/>
 * <br/>
 *     <em>{ P<sub>implied</sub> - D<sub>implied</sub> }</em>.
 * </p>
 * <p>
 * Each instance of this class encapsulates a <em>target permission</em>, representing the
 * "positive" permission being denied.
 * </p>
 * Denied permissions employ the following naming scheme:<br/>
 * <br/>
 *
    <em>&lt;target_class_name&gt;:&lt;target_name&gt;(:&lt;target_actions&gt;)</em><br/>
 * <br/>
 * where:
 * <ul>
 * <li><em>target_class_name</em> is the name of the target permission's class,</li>
 * <li><em>target_name</em> is the name of the target permission, and</li>
 * <li><em>target_actions</em> is, optionally, the actions string of the target permission.</li>
 * </ul>
 * A denied permission, having a target permission <em>t</em>, is said to <em>imply</em> another
 * permission <em>p</em>, if:
 * <ul>
 * <li>p <em>is not</em> itself a denied permission, and <code>(t.implies(p) == true)</code>,
 * or</li>
 * <li>p <em>is</em> a denied permission, with a target <em>t1</em>, and
 * <code>(t.implies(t1) == true)</code>.
 * </ul>
 * <p>
 * It is the responsibility of the policy decision point (e.g., the <code>Policy</code> provider) to
 * take denied permission semantics into account when issuing authorization statements.
 * </p>
 */
public final class DeniedPermission extends BasicPermission {

    private final Permission target;
    private static final long serialVersionUID = 473625163869800679L;

    /**
     * Instantiates a <code>DeniedPermission</code> that encapsulates a target permission of the
     * indicated class, specified name and, optionally, actions.
     *
     * @throws IllegalArgumentException
     *             if:
     *             <ul>
     *             <li><code>targetClassName</code> is <code>null</code>, the empty string, does not
     *             refer to a concrete <code>Permission</code> descendant, or refers to
```

```
     *             <code>DeniedPermission.class</code> or
<code>UnresolvedPermission.class</code>.</li>
     *             <li><code>targetName</code> is <code>null</code>.</li>
     *             <li><code>targetClassName</code> cannot be instantiated, and it's the caller's
fault;
     *             e.g., because <code>targetName</code> and/or <code>targetActions</code> do not
adhere
     *             to the naming constraints of the target class; or due to the target class not
     *             exposing a <code>(String name)</code>, or <code>(String name, String
actions)</code>
     *             constructor, depending on whether <code>targetActions</code> is <code>null</code>
or
     *             not.</li>
     *             </ul>
     */
    public static DeniedPermission newDeniedPermission(String targetClassName, String targetName,
            String targetActions) {
        if (targetClassName == null || targetClassName.trim().isEmpty() || targetName == null) {
            throw new IllegalArgumentException(
                    "Null or empty [targetClassName], or null [targetName] argument was
supplied.");
        }
        StringBuilder sb = new StringBuilder(targetClassName).append(":").append(targetName);
        if (targetName != null) {
            sb.append(":").append(targetName);
        }
        return new DeniedPermission(sb.toString());
    }

    /**
     * Instantiates a <code>DeniedPermission</code> that encapsulates a target permission of the
class,
     * name and, optionally, actions, collectively provided as the <code>name</code> argument.
     *
     * @throws IllegalArgumentException
     *             if:
     *             <ul>
     *             <li><code>name</code>'s target permission class name component is empty, does not
     *             refer to a concrete <code>Permission</code> descendant, or refers to
     *             <code>DeniedPermission.class</code> or
<code>UnresolvedPermission.class</code>.</li>
     *             <li><code>name</code>'s target name component is <code>empty</code></li>
     *             <li>the target permission class cannot be instantiated, and it's the caller's
fault;
     *             e.g., because <code>name</code>'s target name and/or target actions component(s)
do
     *             not adhere to the naming constraints of the target class; or due to the target
class
     *             not exposing a <code>(String name)</code>, or
     *             <code>(String name, String actions)</code> constructor, depending on whether the
     *             target actions component is empty or not.</li>
     *             </ul>
     */
    public DeniedPermission(String name) {
        super(name);
        String[] comps = name.split(":");
        if (comps.length < 2) {
            throw new IllegalArgumentException(MessageFormat.format("Malformed name [{0}]
argument.", name));
        }
        this.target = initTarget(comps[0], comps[1], ((comps.length < 3) ? null : comps[2]));
    }
```

```
    /**
     * Instantiates a <code>DeniedPermission</code> that encapsulates the given target permission.
     *
     * @throws IllegalArgumentException
     *             if <code>target</code> is <code>null</code>, a <code>DeniedPermission</code>, or
an
     *             <code>UnresolvedPermission</code>.
     */
    public static DeniedPermission newDeniedPermission(Permission target) {
        if (target == null) {
            throw new IllegalArgumentException("Null [target] argument.");
        }
        if (target instanceof DeniedPermission || target instanceof UnresolvedPermission) {
            throw new IllegalArgumentException("[target] must not be a DeniedPermission or an
UnresolvedPermission.");
        }
        StringBuilder sb = new
StringBuilder(target.getClass().getName()).append(":").append(target.getName());
        String targetActions = target.getActions();
        if (targetActions != null) {
            sb.append(":").append(targetActions);
        }
        return new DeniedPermission(sb.toString(), target);
    }

    private DeniedPermission(String name, Permission target) {
        super(name);
        this.target = target;
    }

    private Permission initTarget(String targetClassName, String targetName, String targetActions)
{
        Class<?> targetClass;
        try {
            targetClass = Class.forName(targetClassName);
        }
        catch (ClassNotFoundException cnfe) {
            if (targetClassName.trim().isEmpty()) {
                targetClassName = "<empty>";
            }
            throw new IllegalArgumentException(
                    MessageFormat.format("Target Permission class [{0}] not found.",
targetClassName));
        }
        if (!Permission.class.isAssignableFrom(targetClass) ||
Modifier.isAbstract(targetClass.getModifiers())) {
            throw new IllegalArgumentException(MessageFormat
                    .format("Target Permission class [{0}] is not a (concrete) Permission.",
targetClassName));
        }
        if (targetClass == DeniedPermission.class || targetClass == UnresolvedPermission.class) {
            throw new IllegalArgumentException("Target Permission class cannot be a
DeniedPermission itself.");
        }
        Constructor<?> targetCtor;
        try {
            if (targetActions == null) {
                targetCtor = targetClass.getConstructor(String.class);
            }
            else {
                targetCtor = targetClass.getConstructor(String.class, String.class);
            }
```

```
        }
        catch (NoSuchMethodException nsme) {
            throw new IllegalArgumentException(MessageFormat.format(
                    "Target Permission class [{0}] does not provide or expose a (String name) or
(String name, String actions) constructor.",
                    targetClassName));
        }
        try {
            return (Permission) targetCtor
                    .newInstance(((targetCtor.getParameterCount() == 1) ? new Object[] { targetName
}
                            : new Object[] { targetName, targetActions }));
        }
        catch (ReflectiveOperationException roe) {
            if (roe instanceof InvocationTargetException) {
                if (targetName == null) {
                    targetName = "<null>";
                }
                else if (targetName.trim().isEmpty()) {
                    targetName = "<empty>";
                }
                if (targetActions == null) {
                    targetActions = "<null>";
                }
                else if (targetActions.trim().isEmpty()) {
                    targetActions = "<empty>";
                }
                throw new IllegalArgumentException(MessageFormat.format(
                        "Could not instantiate target Permission class [{0}]; provided target name
[{1}] and/or target actions [{2}] potentially erroneous.",
                        targetClassName, targetName, targetActions), roe);
            }
            throw new RuntimeException(
                    "Could not instantiate target Permission class [{0}]; an unforeseen error
occurred - see attached cause for details",
                    roe);
        }
    }

    /**
     * Checks whether the given permission is implied by this one, as per the {@link DeniedPermission
     * overview}.
     */
    @Override
    public boolean implies(Permission p) {
        if (p instanceof DeniedPermission) {
            return target.implies(((DeniedPermission) p).target);
        }
        return target.implies(p);
    }

    /**
     * Returns this denied permission's target permission (the actual positive permission which is
not
     * to be granted).
     */
    public Permission getTargetPermission() {
        return target;
    }

}
```

The `DenyingPolicy` class

```java
package com.example;

import java.security.CodeSource;
import java.security.NoSuchAlgorithmException;
import java.security.Permission;
import java.security.PermissionCollection;
import java.security.Policy;
import java.security.ProtectionDomain;
import java.security.UnresolvedPermission;
import java.util.Enumeration;

/**
 * Wrapper that adds rudimentary {@link DeniedPermission} processing capabilities to the standard
 * file-backed <code>Policy</code>.
 */
public final class DenyingPolicy extends Policy {

    {
        try {
            defaultPolicy = Policy.getInstance("javaPolicy", null);
        }
        catch (NoSuchAlgorithmException nsae) {
            throw new RuntimeException("Could not acquire default Policy.", nsae);
        }
    }

    private final Policy defaultPolicy;

    @Override
    public PermissionCollection getPermissions(CodeSource codesource) {
        return defaultPolicy.getPermissions(codesource);
    }

    @Override
    public PermissionCollection getPermissions(ProtectionDomain domain) {
        return defaultPolicy.getPermissions(domain);
    }

    /**
     * @return
     *         <ul>
     *         <li><code>true</code> if:</li>
     *         <ul>
     *         <li><code>permission</code> <em>is not</em> an instance of
     *         <code>DeniedPermission</code>,</li>
     *         <li>an <code>implies(domain, permission)</code> invocation on the system-default
     *         <code>Policy</code> yields <code>true</code>, and</li>
     *         <li><code>permission</code> <em>is not</em> implied by any
<code>DeniedPermission</code>s
     *         having potentially been assigned to <code>domain</code>.</li>
     *         </ul>
     *         <li><code>false</code>, otherwise.
     *         </ul>
     */
    @Override
    public boolean implies(ProtectionDomain domain, Permission permission) {
        if (permission instanceof DeniedPermission) {
            /*
             * At the policy decision level, DeniedPermissions can only themselves imply, not be
implied (as
             * they take away, rather than grant, privileges). Furthermore, clients aren't supposed
to use this
```

```
             * method for checking whether some domain _does not_ have a permission (which is what
             * DeniedPermissions express after all).
             */
            return false;
        }

        if (!defaultPolicy.implies(domain, permission)) {
            // permission not granted, so no need to check whether denied
            return false;
        }

        /*
         * Permission granted--now check whether there's an overriding DeniedPermission. The
following
         * assumes that previousPolicy is a sun.security.provider.PolicyFile (different
implementations
         * might not support #getPermissions(ProtectionDomain) and/or handle UnresolvedPermissions
         * differently).
         */

        Enumeration<Permission> perms = defaultPolicy.getPermissions(domain).elements();
        while (perms.hasMoreElements()) {
            Permission p = perms.nextElement();
            /*
             * DeniedPermissions will generally remain unresolved, as no code is expected to check
whether other
             * code has been "granted" such a permission.
             */
            if (p instanceof UnresolvedPermission) {
                UnresolvedPermission up = (UnresolvedPermission) p;
                if (up.getUnresolvedType().equals(DeniedPermission.class.getName())) {
                    // force resolution
                    defaultPolicy.implies(domain, up);
                    // evaluate right away, to avoid reiterating over the collection
                    p = new DeniedPermission(up.getUnresolvedName());
                }
            }
            if (p instanceof DeniedPermission && p.implies(permission)) {
                // permission denied
                return false;
            }
        }
        // permission granted
        return true;
    }

    @Override
    public void refresh() {
        defaultPolicy.refresh();
    }

}
```

Demo

```
package com.example;

import java.security.Policy;

public class Main {

    public static void main(String... args) {
        Policy.setPolicy(new DenyingPolicy());
```

```
        System.setSecurityManager(new SecurityManager());
        // should fail
        System.getProperty("foo.bar");
    }

}
```

Assign some permissions:

```
grant codeBase "file:///path/to/classes/bin/-"
    permission java.util.PropertyPermission "*", "read,write";
    permission com.example.DeniedPermission "java.util.PropertyPermission:foo.bar:read";
};
```

Lastly, run the `Main` and watch it fail, due to the "deny" rule (the `DeniedPermission`) overriding the `grant` (its `PropertyPermission`). Note that a `setProperty("foo.baz", "xyz")` invocation would instead have succeeded, since the denied permission only covers the "read" action, and solely for the "foo.bar" property.

Chapter 145: JNDI

Section 145.1: RMI through JNDI

This example shows how JNDI works in RMI. It has two roles:

- to provide the server with a bind/unbind/rebind API to the RMI Registry
- to provide the client with a lookup/list API to the RMI Registry.

The RMI Registry is part of RMI, not JNDI.

To make this simple, we will use `java.rmi.registry.CreateRegistry()` to create the RMI Registry.

1. Server.java(the JNDI server)

```
package com.neohope.jndi.test;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.io.IOException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.util.Hashtable;

/**
 * JNDI Server
 * 1.create a registry on port 1234
 * 2.bind JNDI
 * 3.wait for connection
 * 4.clean up and end
 */
public class Server {
    private static Registry registry;
    private static InitialContext ctx;

    public static void initJNDI() {
        try {
            registry = LocateRegistry.createRegistry(1234);
            final Hashtable jndiProperties = new Hashtable();
            jndiProperties.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.rmi.registry.RegistryContextFactory");
            jndiProperties.put(Context.PROVIDER_URL, "rmi://localhost:1234");
            ctx = new InitialContext(jndiProperties);
        } catch (NamingException e) {
            e.printStackTrace();
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }

    public static void bindJNDI(String name, Object obj) throws NamingException {
        ctx.bind(name, obj);
    }

    public static void unbindJNDI(String name) throws NamingException {
        ctx.unbind(name);
    }
```

```
    public static void unInitJNDI() throws NamingException {
        ctx.close();
    }

    public static void main(String[] args) throws NamingException, IOException {
        initJNDI();
        NMessage msg = new NMessage("Just A Message");
        bindJNDI("/neohope/jndi/test01", msg);
        System.in.read();
        unbindJNDI("/neohope/jndi/test01");
        unInitJNDI();
    }
}
```

2. Client.java(the JNDI client)

```
package com.neohope.jndi.test;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Hashtable;

/**
 * 1.init context
 * 2.lookup registry for the service
 * 3.use the service
 * 4.end
 */
public class Client {
    public static void main(String[] args) throws NamingException {
        final Hashtable jndiProperties = new Hashtable();
        jndiProperties.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.rmi.registry.RegistryContextFactory");
        jndiProperties.put(Context.PROVIDER_URL, "rmi://localhost:1234");

        InitialContext ctx = new InitialContext(jndiProperties);
        NMessage msg = (NeoMessage) ctx.lookup("/neohope/jndi/test01");
        System.out.println(msg.message);
        ctx.close();
    }
}
```

3. NMessage.java (RMI server class)

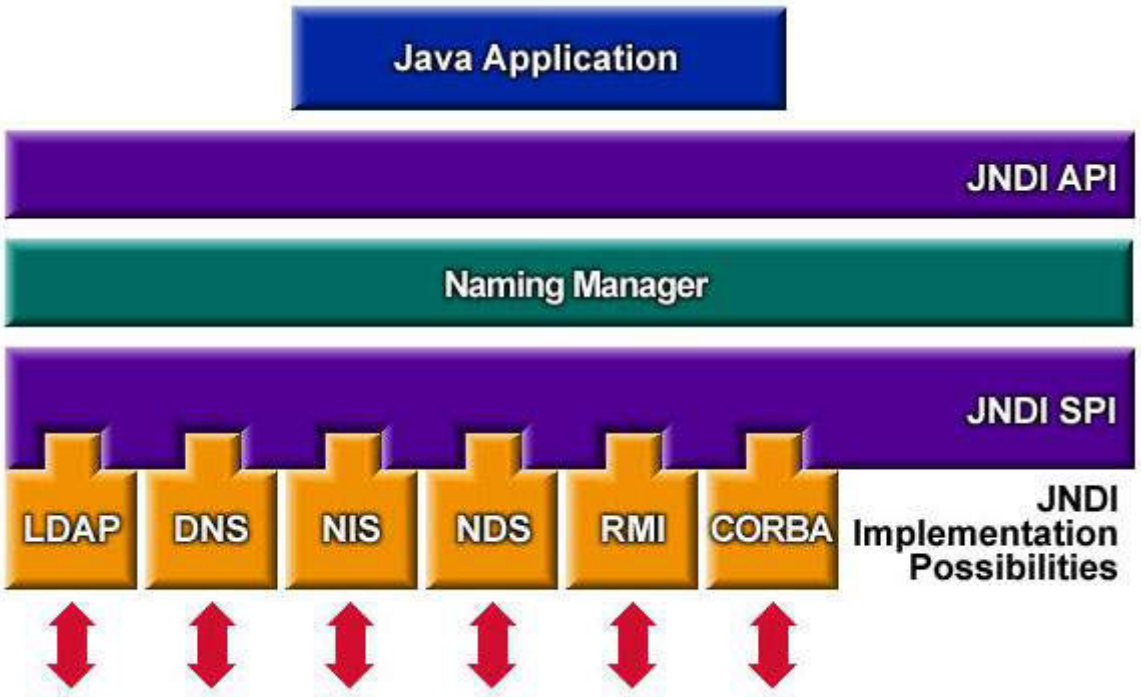
```
package com.neohope.jndi.test;

import java.io.Serializable;
import java.rmi.Remote;

/**
 * NMessage
 * RMI server class
 * must implements Remote and Serializable
 */
public class NMessage implements Remote, Serializable {
    public String message = "";

    public NMessage(String message)
    {
        this.message = message;
```

```
    }
}
```

How to run the eaxmple:

1. build and start the server
2. build and start the client

Introduce



The **Java Naming and Directory Interface (JNDI)** is a Java API for a directory service that allows Java software clients to discover and look up data and objects via a name. It is designed to be independent of any specific naming or directory service implementation.

The JNDI architecture consists of an **API** (Application Programming Interface) and an **SPI** (Service Provider Interface). Java applications use this API to access a variety of naming and directory services. The SPI enables a variety of naming and directory services to be plugged in transparently, allowing the Java application using the API of the JNDI technology to access their services.

As you can see form the picture above, JNDI supports LDAP, DNS, NIS, NDS, RMI and CORBA. Of course, you can extend it.

How it works

In this example, the Java RMI use the JNDI API to look up objects in a network. If you want to look up a object, you need at least two pieces of information:

- Where to find the object

The RMI Registry manages the name bindings, it tells you where to find the object.

- The name of the object

What is a object's name? It is usually a string, it can also be a object that implements the Name interface.

Step by step

1. First you need a registry, which manage the name binding. In this example, we use `java.rmi.registry.LocateRegistry`.

```
//This will start a registry on localhost, port 1234
registry = LocateRegistry.createRegistry(1234);
```

2. Both client and server need a Context. Server use the Context to bind the name and object. Client use the Context to lookup the name and get the object.

```
//We use com.sun.jndi.rmi.registry.RegistryContextFactory as the InitialContextFactory
final Hashtable jndiProperties = new Hashtable();
jndiProperties.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.rmi.registry.RegistryContextFactory");
//the registry usrl is "rmi://localhost:1234"
jndiProperties.put(Context.PROVIDER_URL, "rmi://localhost:1234");
InitialContext ctx = new InitialContext(jndiProperties);
```

3. The server bind the name and object

```
//The jndi name is "/neohope/jndi/test01"
bindJNDI("/neohope/jndi/test01", msg);
```

4. The client look up the object by the name "/neohope/jndi/test01"

```
//look up the object by name "java:com/neohope/jndi/test01"
NeoMessage msg = (NeoMessage) ctx.lookup("/neohope/jndi/test01");
```

5. Now the client can use the object
6. When the server is ending, need to clean up.

```
ctx.unbind("/neohope/jndi/test01");
ctx.close();
```

Chapter 146: sun.misc.Unsafe

Section 146.1: Instantiating sun.misc.Unsafe via reflection

```
public static Unsafe getUnsafe() {
    try {
        Field unsafe = Unsafe.class.getDeclaredField("theUnsafe");
        unsafe.setAccessible(true);
        return (Unsafe) unsafe.get(null);
    } catch (IllegalAccessException e) {
        // Handle
    } catch (IllegalArgumentException e) {
        // Handle
    } catch (NoSuchFieldException e) {
        // Handle
    } catch (SecurityException e) {
        // Handle
    }
}
```

`sun.misc.Unsafe` has a Private constructor, and the static `getUnsafe()` method is guarded with a check of the classloader to ensure that the code was loaded with the primary classloader. Therefore, one method of loading the instance is to use reflection to get the static field.

Section 146.2: Instantiating sun.misc.Unsafe via bootclasspath

```
public class UnsafeLoader {
    public static Unsafe loadUnsafe() {
        return Unsafe.getUnsafe();
    }
}
```

While this example will compile, it is likely to fail at runtime unless the Unsafe class was loaded with the primary classloader. To ensure that happens the JVM should be loaded with the appropriate arguments, like:

```
java -Xbootclasspath:$JAVA_HOME/jre/lib/rt.jar:./UnsafeLoader.jar foo.bar.MyApp
```

The `foo.bar.MyApp` class can then use `UnsafeLoader.loadUnsafe()`.

Section 146.3: Getting Instance of Unsafe

Unsafe is stored as a private field that cannot be accessed directly. The constructor is private and the only method to access `public static Unsafe getUnsafe()` has privileged access. By use of reflection, there is a work-around to make private fields accessible:

```
public static final Unsafe UNSAFE;

static {
    Unsafe unsafe = null;

    try {
        final PrivilegedExceptionAction<Unsafe> action = () -> {
            final Field f = Unsafe.class.getDeclaredField("theUnsafe");
            f.setAccessible(true);

            return (Unsafe) f.get(null);
```

```
        };

        unsafe = AccessController.doPrivileged(action);
    } catch (final Throwable t) {
        throw new RuntimeException("Exception accessing Unsafe", t);
    }

    UNSAFE = unsafe;
}
```

Section 146.4: Uses of Unsafe

Some uses of unsafe is s follows:

Use	API
Off heap / direct memory allocation, reallocation and deallocation	`allocateMemory(bytes)`, `reallocateMemory(address, bytes)` and `freeMemory(address)`
Memory fences	`loadFence()`, `storeFence()`, `fullFence()`
Parking current thread	`park(isAbsolute, time)`, `unpark(thread)`
Direct field and or memory access	`get*` and `put*` family of methods
Throwing unchecked exceptions	`throwException(e)`
CAS and Atomic Operations	`compareAndSwap*` family of methods
Setting out memory	`setMemory`
Volatile or concurrent operations	`get*`**`Volatile`**, `put*`**`Volatile`**, `putOrdered*`

The get and put family of methods are relative to a given object. If the object is null then it is treated as an absolute address.

```
// Putting a value to a field
protected static long fieldOffset = UNSAFE.objectFieldOffset(getClass().getField("theField"));
UNSAFE.putLong(this, fieldOffset , newValue);

// Puting an absolute value
 UNSAFE.putLong(null, address, newValue);
 UNSAFE.putLong(address, newValue);
```

Some methods are only defined for int and longs. You can use these methods on floats and doubles using `floatToRawIntBits`, `intBitsToFloat`, doubleToRawLongBits, longBitsToDouble`

Chapter 147: Java Memory Model

Section 147.1: Motivation for the Memory Model

Consider the following example:

```
public class Example {
    public int a, b, c, d;

    public void doIt() {
       a = b + 1;
       c = d + 1;
    }
}
```

If this class is used is a single-threaded application, then the observable behavior will be exactly as you would expect. For instance:

```
public class SingleThreaded {
    public static void main(String[] args) {
        Example eg = new Example();
        System.out.println(eg.a + ", " + eg.c);
        eg.doIt();
        System.out.println(eg.a + ", " + eg.c);
    }
}
```

will output:

```
0, 0
1, 1
```

As far as the "main" thread can tell, the statements in the `main()` method and the `doIt()` method will be executed in the order that they are written in the source code. This is a clear requirement of the Java Language Specification (JLS).

Now consider the same class used in a multi-threaded application.

```
public class MultiThreaded {
    public static void main(String[] args) {
        final Example eg = new Example();
        new Thread(new Runnable() {
            public void run() {
                while (true) {
                    eg.doIt();
                }
            }
        }).start();
        while (true) {
            System.out.println(eg.a + ", " + eg.c);
        }
    }
}
```

What will this print?

In fact, according to the JLS it is not possible to predict that this will print:

- You will probably see a few lines of `0, 0` to start with.
- Then you probably see lines like `N, N` or `N, N + 1`.
- You might see lines like `N + 1, N`.
- In theory, you might even see that the `0, 0` lines continue forever1.

1 - In practice the presence of the `println` statements is liable to cause some serendipitous synchronization and memory cache flushing. That is likely to hide some of the effects that would cause the above behavior.

So how can we explain these?

Reordering of assignments

One possible explanation for unexpected results is that the JIT compiler has changed the order of the assignments in the `doIt()` method. The JLS requires that statements *appear to* execute in order *from the perspective of the current* thread. In this case, nothing in the code of the `doIt()` method can observe the effect of a (hypothetical) reordering of those two statement. This means that the JIT compiler would be permitted to do that.

Why would it do that?

On typical modern hardware, machine instructions are executed using a instruction pipeline which allows a sequence of instructions to be in different stages. Some phases of instruction execution take longer than others, and memory operations tend to take a longer time. A smart compiler can optimize the instruction throughput of the pipeline by ordering the instructions to maximize the amount of overlap. This may lead to executing parts of statements out of order. The JLS permits this provided that not affect the result of the computation *from the perspective of the current thread*.

Effects of memory caches

A second possible explanation is effect of memory caching. In a classical computer architecture, each processor has a small set of registers, and a larger amount of memory. Access to registers is much faster than access to main memory. In modern architectures, there are memory caches that are slower than registers, but faster than main memory.

A compiler will exploit this by trying to keep copies of variables in registers, or in the memory caches. If a variable does not *need* to be flushed to main memory, or does not *need* to be read from memory, there are significant performance benefits in not doing this. In cases where the JLS does not require memory operations to be visible to another thread, the Java JIT compiler is likely to not add the "read barrier" and "write barrier" instructions that will force main memory reads and writes. Once again, the performance benefits of doing this are significant.

Proper synchronization

So far, we have seen that the JLS allows the JIT compiler to generate code that makes single-threaded code faster by reordering or avoiding memory operations. But what happens when other threads can observe the state of the (shared) variables in main memory?

The answer is, that the other threads are liable to observe variable states which would appear to be impossible ... based on the code order of the Java statements. The solution to this is to use appropriate synchronization. The three main approaches are:

- Using primitive mutexes and the **`synchronized`** constructs.
- Using **`volatile`** variables.
- Using higher level concurrency support; e.g. classes in the `java.util.concurrent` packages.

But even with this, it is important to understand where synchronization is needed, and what effects that you can rely on. This is where the Java Memory Model comes in.

The Memory Model

The Java Memory Model is the section of the JLS that specifies the conditions under which one thread is guaranteed to see the effects of memory writes made by another thread. The Memory Model is specified with a fair degree of *formal rigor*, and (as a result) requires detailed and careful reading to understand. But the basic principle is that certain constructs create a "happens-before" relation between write of a variable by one thread, and a subsequent read of the same variable by another thread. If the "happens before" relation exists, the JIT compiler is *obliged* to generate code that will ensure that the read operation sees the value written by the write.

Armed with this, it is possible to reason about memory coherency in a Java program, and decide whether this will be predictable and consistent for *all* execution platforms.

Section 147.2: Happens-before relationships

(The following is a simplified version of what the Java Language Specification says. For a deeper understanding, you need to read the specification itself.)

Happens-before relationships are the part of the Memory Model that allow us to understand and reason about memory visibility. As the JLS says (JLS 17.4.5):

> "Two *actions* can be ordered by a *happens-before* relationship. If one action *happens-before* another, then the first is visible to and ordered before the second."

What does this mean?

Actions

The actions that the above quote refers to are specified in JLS 17.4.2. There are 5 kinds of action listed defined by the spec:

- Read: Reading a non-volatile variable.
- Write: Writing a non-volatile variable.
- Synchronization actions:
    - Volatile read: Reading a volatile variable.
    - Volatile write: Writing a volatile variable.
    - Lock. Locking a monitor
    - Unlock. Unlocking a monitor.
    - The (synthetic) first and last actions of a thread.
    - Actions that start a thread or detect that a thread has terminated.
- External Actions. An action that has a result that depends on the environment in which the program.
- Thread divergence actions. These model the behavior of certain kinds of infinite loop.

Program Order and Synchronization Order

These two orderings (JLS 17.4.3 and JLS 17.4.4) govern the execution of statements in a Java

Program order describes the order of statement execution within a single thread.

Synchronization order describes the order of statement execution for two statements connected by a synchronization:

- An unlock action on monitor *synchronizes-with* all subsequent lock actions on that monitor.
- A write to a volatile variable *synchronizes-with* all subsequent reads of the same variable by any thread.
- An action that starts a thread (i.e. the call to `Thread.start()`) *synchronizes-with* the first action in the thread it starts (i.e. the call to the thread's `run()` method).
- The default initialization of fields *synchronizes-with* the first action in every thread. (See the JLS for an explanation of this.)
- The final action in a thread *synchronizes-with* any action in another thread that detects the termination; e.g. the return of a `join()` call or `isTerminated()` call that returns `true`.
- If one thread interrupts another thread, the interrupt call in the first thread *synchronizes-with* the point where another thread detects that the thread was interrupted.

Happens-before Order

This ordering (JLS 17.4.5) is what determines whether a memory write is guaranteed to be visible to a subsequent memory read.

More specifically, a read of a variable `v` is guaranteed to observe a write to `v` if and only if `write(v)` *happens-before* `read(v)` AND there is no intervening write to `v`. If there are intervening writes, then the `read(v)` may see the results of them rather than the earlier one.

The rules that define the *happens-before* ordering are as follows:

- **Happens-Before Rule #1** - If x and y are actions of the same thread and x comes before y in *program order*, then x *happens-before* y.
- **Happens-Before Rule #2** - There is a happens-before edge from the end of a constructor of an object to the start of a finalizer for that object.
- **Happens-Before Rule #3** - If an action x *synchronizes-with* a subsequent action y, then x *happens-before* y.
- **Happens-Before Rule #4** - If x *happens-before* y and y *happens-before* z then x *happens-before* z.

In addition, various classes in the Java standard libraries are specified as defining *happens-before* relationships. You can interpret this as meaning that it happens *somehow*, without needing to know exactly how the guarantee is going to be met.

Section 147.3: How to avoid needing to understand the Memory Model

The Memory Model is difficult to understand, and difficult to apply. It is useful if you need to reason about the correctness of multi-threaded code, but you do not want to have to do this reasoning for every multi-threaded application that you write.

If you adopt the following principals when writing concurrent code in Java, you can *largely* avoid the need to resort to *happens-before* reasoning.

- Use immutable data structures where possible. A properly implemented immutable class will be thread-safe, and will not introduce thread-safety issues when you use it with other classes.
- Understand and avoid "unsafe publication".
- Use primitive mutexes or `Lock` objects to synchronize access to state in mutable objects that need to be thread-safe1.
- Use `Executor` / `ExecutorService` or the fork join framework rather than attempting to create manage threads directly.
- Use the `java.util.concurrent classes that provide advanced locks, semaphores, latches and barriers, instead of using wait/notify/notifyAll directly.
- Use the `java.util.concurrent` versions of maps, sets, lists, queues and deques rather than external synchonization of non-concurrent collections.

The general principle is to try to use Java's built-in concurrency libraries rather than "rolling your own" concurrency. You can rely on them working, if you use them properly.

1 - Not all objects need to be thread safe. For example, if an object or objects is *thread-confined* (i.e. it is only accessible to one thread), then its thread-safety is not relevant.

Section 147.4: Happens-before reasoning applied to some examples

We will present some examples to show how to apply *happens-before* reasoning to check that writes are visible to subsequent reads.

Single-threaded code

As you would expect, writes are always visible to subsequent reads in a single-threaded program.

```
public class SingleThreadExample {
    public int a, b;

    public int add() {
       a = 1;          // write(a)
       b = 2;          // write(b)
       return a + b;   // read(a) followed by read(b)
    }
}
```

By Happens-Before Rule #1:

1. The `write(a)` action *happens-before* the `write(b)` action.
2. The `write(b)` action *happens-before* the `read(a)` action.
3. The `read(a)` action *happens-before* the `read(a)` action.

By Happens-Before Rule #4:

4. `write(a)` *happens-before* `write(b)` AND `write(b)` *happens-before* `read(a)` IMPLIES `write(a)` *happens-before* `read(a)`.

5. `write(b)` *happens-before* `read(a)` AND `read(a)` *happens-before* `read(b)` IMPLIES `write(b)` *happens-before* `read(b)`.

Summing up:

6. The `write(a)` *happens-before* `read(a)` relation means that the `a + b` statement is guaranteed to see the correct value of `a`.
7. The `write(b)` *happens-before* `read(b)` relation means that the `a + b` statement is guaranteed to see the correct value of `b`.

Behavior of 'volatile' in an example with 2 threads

We will use the following example code to explore some implications of the Memory Model for `volatile.

```
public class VolatileExample {
    private volatile int a;
    private int b;         // NOT volatile

    public void update(int first, int second) {
       b = first;         // write(b)
       a = second;        // write-volatile(a)
    }

    public int observe() {
       return a + b;      // read-volatile(a) followed by read(b)
    }
}
```

First, consider the following sequence of statements involving 2 threads:

1. A single instance of `VolatileExample` is created; call it `ve`,
2. `ve.update(1, 2)` is called in one thread, and
3. `ve.observe()` is called in another thread.

By Happens-Before Rule #1:

1. The `write(a)` action *happens-before* the `volatile-write(a)` action.
2. The `volatile-read(a)` action *happens-before* the `read(b)` action.

By Happens-Before Rule #2:

3. The `volatile-write(a)` action in the first thread *happens-before* the `volatile-read(a)` action in the second thread.

By Happens-Before Rule #4:

4. The `write(b)` action in the first thread *happens-before* the `read(b)` action in the second thread.

In other words, for this particular sequence, we are guaranteed that the 2nd thread will see the update to the non-volatile variable `b` made by the first thread. However, it is should also be clear that if the assignments in the `update` method were the other way around, or the `observe()` method read the variable `b` before `a`, then the *happens-before* chain would be broken. The chain would also be broken if `volatile-read(a)` in the second thread was not subsequent to the `volatile-write(a)` in the first thread.

When the chain is broken, there is no *guarantee* that `observe()` will see the correct value of `b`.

Volatile with three threads

Suppose we to add a third thread into the previous example:

1. A single instance of `VolatileExample` is created; call it `ve`,
2. Two threads call `update`:
    - `ve.update(1, 2)` is called in one thread,
    - `ve.update(3, 4)` is called in the second thread,
3. `ve.observe()` is subsequently called in a third thread.

To analyse this completely, we need to consider all of the possible interleavings of the statements in thread one and thread two. Instead, we will consider just two of them.

Scenario #1 - suppose that `update(1, 2)` precedes `update(3,4)` we get this sequence:

```
write(b, 1), write-volatile(a, 2)     // first thread
write(b, 3), write-volatile(a, 4)     // second thread
read-volatile(a), read(b)             // third thread
```

In this case, it is easy to see that there is an unbroken *happens-before* chain from `write(b, 3)` to `read(b)`. Furthermore there is no intervening write to `b`. So, for this scenario, the third thread is guaranteed to see `b` as having value `3`.

Scenario #2 - suppose that `update(1, 2)` and `update(3,4)` overlap and the ations are interleaved as follows:

```
write(b, 3)                           // second thread
write(b, 1)                           // first thread
write-volatile(a, 2)                  // first thread
write-volatile(a, 4)                  // second thread
read-volatile(a), read(b)             // third thread
```

Now, while there is a *happens-before* chain from `write(b, 3)` to `read(b)`, there is an intervening `write(b, 1)` action performed by the other thread. This means we cannot be certain which value `read(b)` will see.

(Aside: This demonstrates that we cannot rely on **`volatile`** for ensuring visibility of non-volatile variables, except in very limited situations.)

Chapter 148: Java deployment

There are a variety of technologies for "packaging" Java applications, webapps and so forth, for deployment to the platform on which they will run. They range from simple library or executable `JAR` files, `WAR` and `EAR` files, through to installers and self-contained executables.

Section 148.1: Making an executable JAR from the command line

To make a jar, you need one or more class files. This should have a main method if it is to be run by a double click.

For this example, we will use:

```
import javax.swing.*;
import java.awt.Container;

public class HelloWorld {

    public static void main(String[] args) {
        JFrame f = new JFrame("Hello, World");
        JLabel label = new JLabel("Hello, World");
        Container cont = f.getContentPane();
        cont.add(label);
        f.setSize(400,100);
        f.setVisible(true);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

}
```

It has been named HelloWorld.java

Next, we want to compile this program.

You may use any program you want to do this. To run from the command line, see the documentation on compiling and running your first java program.

Once you have HelloWorld.class, make a new folder and call it whatever you want.

Make another file called manifest.txt and paste into it

```
Main-Class: HelloWorld
Class-Path: HelloWorld.jar
```

Put it in the same folder with HelloWorld.class
Use the command line to make your current directory (`cd C:\Your\Folder\Path\Here` on windows) your folder.

Use Terminal and change directory to the directory (`cd /Users/user/Documents/Java/jarfolder` on Mac) your folder

When that is done, type in `jar -cvfm HelloWorld.jar manifest.txt HelloWorld.class` and press enter. This makes a jar file (in the folder with your manifest and HelloWorld.class) using the .class files specified and named HelloWorld.jar. See the Syntax section for information about the options (like -m and -v).
After these steps, go to your directory with the manifest file and you should find HelloWorld.jar
Clicking on it should display `Hello, World` in a text box.

Section 148.2: Creating an UberJAR for an application and its dependencies

A common requirement for a Java application is that can be deployed by copying a single file. For simple applications that depend only on the standard Java SE class libraries, this requirement is satisfied by creating a JAR file containing all of the (compiled) application classes.

Things are not so straightforward if the application depends on third-party libraries. If you simply put dependency JAR files inside an application JAR, the standard Java class loader will not be able to find the library classes, and your application will not start. Instead, you need to create a single JAR file that contains the application classes and associated resources together with the dependency classes and resources. These need to be organized as a single namespace for the classloader to search.

The such a JAR file is often referred to as an UberJAR.

Creating an UberJAR using the "jar" command

The procedure for creating an UberJAR is straight-forward. (I will use Linux commands for simplicity. The commands should be identical for Mac OS, and similar for Windows.)

1. Create a temporary directory, and change directory to it.

   ```
   $ mkdir tempDir
   $ cd tempDir
   ```

2. For each dependent JAR file, *in the reverse order* that they need to appear on the application's classpath, used the `jar` command to unpack the JAR into the temporary directory.

   ```
   $ jar -xf <path/to/file.jar>
   ```

   Doing this for multiple JAR files will *overlay* contents of the JARs.

3. Copy the application classes from the build tree into the temporary directory

   ```
   $ cp -r path/to/classes .
   ```

4. Create the UberJAR from the contents of the temporary directory:

   ```
   $ jar -cf ../myApplication.jar
   ```

   If you are creating an executable JAR file, include an appropriate MANIFEST.MF as described here.

Creating an UberJAR using Maven

If your project is built using Maven, you can get it to create an UberJAR using either the "maven-assembly" or "maven-shade" plugins. See the Maven Assembly topic (in the Maven documentation) for details.

The advantages and drawbacks of UberJARs

Some of advantages of UberJARs are self-evident:

- An UberJAR is easy to distribute.

- You cannot break the library dependencies for an UberJAR, since the libraries are self-contained.

In addition, if you use an appropriate tooling to create the UberJAR, you will have the option of excluding library classes that are not used from the JAR file. However, that this is typically done by static analysis of the classes. If your application uses reflection, annotation processing and similar techniques, you need to be careful that classes are not excluded incorrectly.

UberJARs also have some disadvantages:

- If you have lots of UberJARs with the same dependencies, then each one will contain a copy of the dependencies.
- Some open source libraries have licenses which *may* preclude 1 their use in an UberJAR.

1 - Some open source library licenses allow you to use the library only of the end-user is able to replace one version of the library with another. UberJARs can make replacement of version dependencies difficult.

Section 148.3: Creating JAR, WAR and EAR files

The JAR, WAR and EAR files types are fundamentally ZIP files with a "manifest" file and (for WAR and EAR files) a particular internal directory / file structure.

The recommended way to create these files is to use a Java-specific build tool which "understands" the requirements for the respective file types. If you don't use a build tool, then IDE "export" is the next option to try.

(*Editorial note: the descriptions of how to create these files are best placed in the documentation for the respective tools. Put them there. Please show some self-restraint and DON'T shoe-horn them into this example!*)

Creating JAR and WAR files using Maven

Creating a JAR or WAR using Maven is simply a matter of putting the correct **`<packaging>`** element into the POM file; e,g,

```
<packaging>jar</packaging>
```

or

```
<packaging>war</packaging>
```

For more details. Maven can be configured to create "executable" JAR files by adding the requisite information about the entry-point class and external dependencies as plugin properties for the maven jar plugin. There is even a plugin for creating "uberJAR" files that combine an application and its dependencies into a single JAR file.

Please refer to the Maven documentation (http://stackoverflow.com/documentation/maven/topics)for more information.

Creating JAR, WAR and EAR files using Ant

The Ant build tool has separate "tasks" for building JAR, WAR and EAR. Please refer to the Ant documentation (http://stackoverflow.com/documentation/ant/topics) for more information.

Creating JAR, WAR and EAR files using an IDE

The three most popular Java IDEs all have built-in support for creating deployment files. The functionality is often described as "exporting".

- Eclipse - http://stackoverflow.com/documentation/eclipse/topics
- NetBeans - http://stackoverflow.com/documentation/netbeans/topics
- Intellij-IDEA - Exporting

Creating JAR, WAR and EAR files using the `jar` command.

It is also possible to create these files "by hand" using the `jar` command. It is simply a matter of assembling a file tree with the correct component files in the correct place, creating a manifest file, and running `jar` to create the JAR file.

Please refer to the `jar` command Topic (Creating and modifying JAR files) for more information

Section 148.4: Introduction to Java Web Start

The Oracle Java Tutorials summarize Web Start as follows:

> Java Web Start software provides the power to launch full-featured applications with a single click. Users can download and launch applications, such as a complete spreadsheet program or an Internet chat client, without going through lengthy installation procedures.

Other advantages of Java Web Start are support for signed code and explicit declaration of platform dependencies, and support for code caching and deployment of application updates.

Java Web Start is also referred to as JavaWS and JAWS. The primary sources of information are:

- The Java Tutorials - Lesson: Java Web Start
- Java Web Start Guide
- Java Web Start FAQ
- JNLP Specification
- `javax.jnlp` API Documentation
- Java Web Start Developers Site

Prerequisites

At a high level, Web Start works by distributing Java applications packed as JAR files from a remote webserver. The prerequisites are:

- A pre-existing Java installation (JRE or JDK) on the target machine where the application is to run. Java 1.2.2 or higher is required:
    - From Java 5.0 onwards, Web Start support is included in the JRE / JDK.
    - For earlier releases, Web Start support is installed separately.
    - The Web Start infrastructure includes some Javascript that can be included in a web page to assist the user to install the necessary software.
- The webserver that hosts the software must be accessible to the target machine.
- If the user is going to launch a Web Start application using a link in a web page, then:
    - they need a compatible web browser, and
    - for modern (secure) browsers, they need to be told how to tell the browser to allow Java to run ... without compromising web browser security.

An example JNLP file

The following example is intended to illustrate the basic functionality of JNLP.

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec="1.0+" codebase="https://www.example.com/demo"
   href="demo_webstart.jnlp">
    <information>
        <title>Demo</title>
        <vendor>The Example.com Team</vendor>
    </information>
    <resources>
        <!-- Application Resources -->
        <j2se version="1.7+" href="http://java.sun.com/products/autodl/j2se"/>
        <jar href="Demo.jar" main="true"/>
    </resources>
    <application-desc
        name="Demo Application"
        main-class="com.example.jwsdemo.Main"
        width="300"
        height="300">
     </application-desc>
     <update check="background"/>
</jnlp>
```

As you can see, a JNLP file XML-based, and the information is all contained in the `<jnlp>` element.

- The `spec` attribute gives the version of the JNPL spec that this file conforms to.
- The `codebase` attribute gives the base URL for resolving relative `href` URLs in the rest of the file.
- The `href` attribute gives the definitive URL for this JNLP file.
- The `<information>` element contains metadata the application including its title, authors, description and help website.
- The `<resources>` element describes the dependencies for the application including the required Java version, OS platform and JAR files.
- The `<application-desc>` (or `<applet-desc>`) element provides information needed to launch the application.

Setting up the web server

The webserver must be configured to use `application/x-java-jnlp-file` as the MIMEtype for `.jnlp` files.

The JNLP file and the application's JAR files must be installed on the webserver so that they are available using the URLs indicated by the JNLP file.

Enabling launch via a web page

If the application is to be launched via a web link, the page that contains the link must be created on the webserver.

- If you can assume that Java Web Start is already installed on the user's machine, then the web page simply needs to contain a link for launching the application. For example.

```
<a href="https://www.example.com/demo_webstart.jnlp">Launch the application</a>
```

- Otherwise, the page should also include some scripting to detect the kind of browser the user is using and request to download and install the required version of Java.

NOTE: It is a bad idea to encourage users to encourage to install Java this way, or even to enable Java in their web browsers so that JNLP web page launch will work.

Launching Web Start applications from the command line

The instructions for launching an Web Start application from the command line are simple. Assuming that the user has a Java 5.0 JRE or JDK installed, the simply need to run this:

```
$ javaws <url>
```

where **`<url>`** is the URL for the JNLP file on the remote server.

Chapter 149: Java plugin system implementations

Section 149.1: Using URLClassLoader

There are several ways to implement a plugin system for a Java application. One of the simplest is to use *URLClassLoader*. The following example will involve a bit of JavaFX code.

Suppose we have a module of a main application. This module is supposed to load plugins in form of Jars from 'plugins' folder. Initial code:

```
package main;

public class MainApplication extends Application
{
    @Override
    public void start(Stage primaryStage) throws Exception
    {
        File pluginDirectory=new File("plugins"); //arbitrary directory
        if(!pluginDirectory.exists())pluginDirectory.mkdir();
        VBox loadedPlugins=new VBox(6); //a container to show the visual info later
        Rectangle2D screenbounds=Screen.getPrimary().getVisualBounds();
        Scene scene=new Scene(loadedPlugins,screenbounds.getWidth()/2,screenbounds.getHeight()/2);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    public static void main(String[] a)
    {
            launch(a);
    }
}
```

Then, we create an interface which will represent a plugin.

```
package main;

public interface Plugin
{
    default void initialize()
    {
        System.out.println("Initialized "+this.getClass().getName());
    }
    default String name(){return getClass().getSimpleName();}
}
```

We want to load classes which implement this interface, so first we need to filter files which have a '.jar' extension:

```
File[] files=pluginDirectory.listFiles((dir, name) -> name.endsWith(".jar"));
```

If there are any files, we need to create collections of URLs and class names:

```
    if(files!=null && files.length>0)
    {
        ArrayList<String> classes=new ArrayList<>();
        ArrayList<URL> urls=new ArrayList<>(files.length);
        for(File file:files)
        {
```

```
            JarFile jar=new JarFile(file);
            jar.stream().forEach(jarEntry -> {
                if(jarEntry.getName().endsWith(".class"))
                {
                    classes.add(jarEntry.getName());
                }
            });
            URL url=file.toURI().toURL();
            urls.add(url);
        }

    }
```

Let's add a static HashSet to *MainApplication* which will hold loaded plugins:

```
static HashSet<Plugin> plugins=new HashSet<>();
```

Next, we instantiate a *URLClassLoader*, and iterate over class names, instantiating classes which implement *Plugin* interface:

```
URLClassLoader urlClassLoader=new URLClassLoader(urls.toArray(new URL[urls.size()]));
classes.forEach(className->{
    try
    {
        Class cls=urlClassLoader.loadClass(className.replaceAll("/",".").replace(".class",""));
//transforming to binary name
        Class[] interfaces=cls.getInterfaces();
        for(Class intface:interfaces)
        {
            if(intface.equals(Plugin.class)) //checking presence of Plugin interface
            {
                Plugin plugin=(Plugin) cls.newInstance(); //instantiating the Plugin
                plugins.add(plugin);
                break;
            }
        }
    }
    catch (Exception e){e.printStackTrace();}
});
```

Then, we can call plugin's methods, for example, to initialize them:

```
if(!plugins.isEmpty())loadedPlugins.getChildren().add(new Label("Loaded plugins:"));
    plugins.forEach(plugin -> {
        plugin.initialize();
        loadedPlugins.getChildren().add(new Label(plugin.name()));
});
```

The final code of *MainApplication*:

```
package main;
public class MainApplication extends Application
{
    static HashSet<Plugin> plugins=new HashSet<>();
    @Override
    public void start(Stage primaryStage) throws Exception
    {
        File pluginDirectory=new File("plugins");
        if(!pluginDirectory.exists())pluginDirectory.mkdir();
```

```
        File[] files=pluginDirectory.listFiles((dir, name) -> name.endsWith(".jar"));
        VBox loadedPlugins=new VBox(6);
        loadedPlugins.setAlignment(Pos.CENTER);
        if(files!=null && files.length>0)
        {
            ArrayList<String> classes=new ArrayList<>();
            ArrayList<URL> urls=new ArrayList<>(files.length);
            for(File file:files)
            {
                JarFile jar=new JarFile(file);
                jar.stream().forEach(jarEntry -> {
                    if(jarEntry.getName().endsWith(".class"))
                    {
                        classes.add(jarEntry.getName());
                    }
                });
                URL url=file.toURI().toURL();
                urls.add(url);
            }
            URLClassLoader urlClassLoader=new URLClassLoader(urls.toArray(new URL[urls.size()]));
            classes.forEach(className->{
                try
                {
                    Class
cls=urlClassLoader.loadClass(className.replaceAll("/",".").replace(".class",""));
                    Class[] interfaces=cls.getInterfaces();
                    for(Class intface:interfaces)
                    {
                        if(intface.equals(Plugin.class))
                        {
                            Plugin plugin=(Plugin) cls.newInstance();
                            plugins.add(plugin);
                            break;
                        }
                    }
                }
                catch (Exception e){e.printStackTrace();}
            });
            if(!plugins.isEmpty())loadedPlugins.getChildren().add(new Label("Loaded plugins:"));
            plugins.forEach(plugin -> {
                plugin.initialize();
                loadedPlugins.getChildren().add(new Label(plugin.name()));
            });
        }
        Rectangle2D screenbounds=Screen.getPrimary().getVisualBounds();
        Scene scene=new Scene(loadedPlugins,screenbounds.getWidth()/2,screenbounds.getHeight()/2);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    public static void main(String[] a)
    {
            launch(a);
    }
}
```

Let's create two plugins. Obviously, the plugin's source should be in a separate module.

```
package plugins;

import main.Plugin;
```

```
public class FirstPlugin implements Plugin
{
    //this plugin has default behaviour
}
```

Second plugin:

```
package plugins;

import main.Plugin;

public class AnotherPlugin implements Plugin
{
    @Override
    public void initialize() //overrided to show user's home directory
    {
        System.out.println("User home directory: "+System.getProperty("user.home"));
    }
}
```

These plugins have to be packaged into standard Jars - this process depends on your IDE or other tools.

When Jars will be put into 'plugins' directly, *MainApplication* will detect them and instantiate appropriate classes.

Chapter 150: JavaBean

JavaBeans (TM) is a pattern for designing Java class APIs that allows instances (beans) to be used in various contexts and using various tools *without* explicitly writing Java code. The patterns consists of conventions for defining getters and setters for *properties*, for defining constructors, and for defining event listener APIs.

Section 150.1: Basic Java Bean

```
public class BasicJavaBean implements java.io.Serializable{

    private int value1;
    private String value2;
    private boolean value3;

    public BasicJavaBean(){}

    public void setValue1(int value1){
        this.value1 = value1;
    }

    public int getValue1(){
        return value1;
    }

    public void setValue2(String value2){
        this.value2 = value2;
    }

    public String getValue2(){
        return value2;
    }

    public void setValue3(boolean value3){
        this.value3 = value3;
    }

    public boolean isValue3(){
        return value3;
    }
}
```

Chapter 151: Java SE 7 Features

In this topic you'll find a summary of the new features added to the Java programming language in Java SE 7. There are many other new features in other fields such as JDBC and Java Virtual Machine (JVM) that are not going to be covered in this topic.

Section 151.1: New Java SE 7 programming language features

- Binary Literals: The integral types (byte, short, int, and long) can also be expressed using the binary number system. To specify a binary literal, add the prefix 0b or 0B to the number.
- Strings in switch Statements: You can use a String object in the expression of a switch statement
- The try-with-resources Statement: The try-with-resources statement is a try statement that declares one or more resources. A resource is as an object that must be closed after the program is finished with it. The try-with-resources statement ensures that each resource is closed at the end of the statement. Any object that implements java.lang.AutoCloseable, which includes all objects which implement java.io.Closeable, can be used as a resource.
- Catching Multiple Exception Types and Rethrowing Exceptions with Improved Type Checking: a single catch block can handle more than one type of exception. This feature can reduce code duplication and lessen the temptation to catch an overly broad exception.
- Underscores in Numeric Literals: Any number of underscore characters (_) can appear anywhere between digits in a numerical literal. This feature enables you, for example, to separate groups of digits in numeric literals, which can improve the readability of your code.
- Type Inference for Generic Instance Creation: You can replace the type arguments required to invoke the constructor of a generic class with an empty set of type parameters (<>) as long as the compiler can infer the type arguments from the context. This pair of angle brackets is informally called the diamond.
- Improved Compiler Warnings and Errors When Using Non-Reifiable Formal Parameters with Varargs Methods

Section 151.2: Binary Literals

```
// An 8-bit 'byte' value:
byte aByte = (byte)0b00100001;

// A 16-bit 'short' value:
short aShort = (short)0b1010000101000101;

// Some 32-bit 'int' values:
int anInt1 = 0b10100001010001011010000101000101;
int anInt2 = 0b101;
int anInt3 = 0B101; // The B can be upper or lower case.

// A 64-bit 'long' value. Note the "L" suffix:
long aLong = 0b1010000101000101101000010100010110100001010001011010000101000101L;
```

Section 151.3: The try-with-resources statement

The example reads the first line from a file. It uses an instance of `BufferedReader` to read data from the file. `BufferedReader` is a resource that must be closed after the program is finished with it:

```
static String readFirstLineFromFile(String path) throws IOException {
  try (BufferedReader br = new BufferedReader(new FileReader(path))) {
    return br.readLine();
  }
```

```
}
```

In this example, the resource declared in the try-with-resources statement is a `BufferedReader`. The declaration statement appears within parentheses immediately after the try keyword. The class `BufferedReader`, in Java SE 7 and later, implements the interface `java.lang.AutoCloseable`. Because the `BufferedReader` instance is declared in a try-with-resource statement, it will be closed regardless of whether the try statement completes normally or abruptly (as a result of the method `BufferedReader.readLine` throwing an `IOException`).

Section 151.4: Underscores in Numeric Literals

The following example shows other ways you can use the underscore in numeric literals:

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi = 3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

You can place underscores only between digits; you cannot place underscores in the following places:

- At the beginning or end of a number
- Adjacent to a decimal point in a floating point literal
- Prior to an F or L suffix
- In positions where a string of digits is expected

Section 151.5: Type Inference for Generic Instance Creation

You can use

```
Map<String, List<String>> myMap = new HashMap<>();
```

instead of

```
Map<String, List<String>> myMap = new HashMap<String, List<String>>();
```

However, you can't use

```
List<String> list = new ArrayList<>();
list.add("A");

  // The following statement should fail since addAll expects
  // Collection<? extends String>

list.addAll(new ArrayList<>());
```

because it can't compile. Note that the diamond often works in method calls; however, it is suggested that you use the diamond primarily for variable declarations.

Section 151.6: Strings in switch Statements

```
public String getTypeOfDayWithSwitchStatement(String dayOfWeekArg) {
     String typeOfDay;
```

```
    switch (dayOfWeekArg) {
        case "Monday":
            typeOfDay = "Start of work week";
            break;
        case "Tuesday":
        case "Wednesday":
        case "Thursday":
            typeOfDay = "Midweek";
            break;
        case "Friday":
            typeOfDay = "End of work week";
            break;
        case "Saturday":
        case "Sunday":
            typeOfDay = "Weekend";
            break;
        default:
            throw new IllegalArgumentException("Invalid day of the week: " + dayOfWeekArg);
    }
    return typeOfDay;
}
```

Chapter 152: Java SE 8 Features

In this topic you'll find a summary of the new features added to the Java programming language in Java SE 8. There are many other new features in other fields such as JDBC and Java Virtual Machine (JVM) that are not going to be covered in this topic.

Section 152.1: New Java SE 8 programming language features

- Lambda Expressions, a new language feature, has been introduced in this release. They enable you to treat functionality as a method argument, or code as data. Lambda expressions let you express instances of single-method interfaces (referred to as functional interfaces) more compactly.
    - Method references provide easy-to-read lambda expressions for methods that already have a name.
    - Default methods enable new functionality to be added to the interfaces of libraries and ensure binary compatibility with code written for older versions of those interfaces.
    - New and Enhanced APIs That Take Advantage of Lambda Expressions and Streams in Java SE 8 describe new and enhanced classes that take advantage of lambda expressions and streams.
- Improved Type Inference - The Java compiler takes advantage of target typing to infer the type parameters of a generic method invocation. The target type of an expression is the data type that the Java compiler expects depending on where the expression appears. For example, you can use an assignment statement's target type for type inference in Java SE 7. However, in Java SE 8, you can use the target type for type inference in more contexts.
    - Target Typing in Lambda Expressions
    - Type Inference
- Repeating Annotations provide the ability to apply the same annotation type more than once to the same declaration or type use.
- Type Annotations provide the ability to apply an annotation anywhere a type is used, not just on a declaration. Used with a pluggable type system, this feature enables improved type checking of your code.
- Method parameter reflection - You can obtain the names of the formal parameters of any method or constructor with the method `java.lang.reflect.Executable.getParameters`. (The classes Method and Constructor extend the class Executable and therefore inherit the method `Executable.getParameters`) However, `.class` files do not store formal parameter names by default. To store formal parameter names in a particular `.class` file, and thus enable the Reflection API to retrieve formal parameter names, compile the source file with the `-parameters` option of the javac compiler.
- Date-time-api - Added new time api in `java.time`. If used this, you don't need to designate timezone.

Chapter 153: Dynamic Method Dispatch

What is Dynamic Method Dispatch?

Dynamic Method Dispatch is a process in which the call to an overridden method is resolved at runtime rather than at compile-time. When an overridden method is called by a reference, Java determines which version of that method to execute based on the type of object it refer to. This is also know as runtime polymorphism.

We will see this through an example.

Section 153.1: Dynamic Method Dispatch - Example Code

> Abstract Class :

```
package base;

/*
Abstract classes cannot be instantiated, but they can be subclassed
*/
public abstract class ClsVirusScanner {

    //With One Abstract method
    public abstract void fnStartScan();

    protected void fnCheckForUpdateVersion(){
        System.out.println("Perform Virus Scanner Version Check");
    }

    protected void fnBootTimeScan(){
        System.out.println("Perform BootTime Scan");
    }
    protected void fnInternetSecutiry(){
        System.out.println("Scan for Internet Security");
    }

    protected void fnRealTimeScan(){
        System.out.println("Perform RealTime Scan");
    }

    protected void fnVirusMalwareScan(){
        System.out.println("Detect Virus & Malware");
    }
}
```

> Overriding Abstract Method in Child Class :

```
import base.ClsVirusScanner;

//All the 3 child classes inherits the base class ClsVirusScanner
//Child Class 1
class ClsPaidVersion extends ClsVirusScanner{
    @Override
    public void fnStartScan() {
        super.fnCheckForUpdateVersion();
        super.fnBootTimeScan();
```

```
        super.fnInternetSecutiry();
        super.fnRealTimeScan();
        super.fnVirusMalwareScan();
    }
};    //ClsPaidVersion IS-A ClsVirusScanner
//Child Class 2

class ClsTrialVersion extends ClsVirusScanner{
    @Override
    public void fnStartScan() {
        super.fnInternetSecutiry();
        super.fnVirusMalwareScan();
    }
};    //ClsTrialVersion IS-A ClsVirusScanner

//Child Class 3
class ClsFreeVersion extends ClsVirusScanner{
    @Override
    public void fnStartScan() {
        super.fnVirusMalwareScan();
    }
};     //ClsTrialVersion IS-A ClsVirusScanner
```

> Dynamic/Late Binding leads to Dynamic method dispatch :

```
//Calling Class
public class ClsRunTheApplication {

    public static void main(String[] args) {

        final String VIRUS_SCANNER_VERSION = "TRIAL_VERSION";

        //Parent Refers Null
        ClsVirusScanner objVS=null;

        //String Cases Supported from Java SE 7
        switch (VIRUS_SCANNER_VERSION){
        case "FREE_VERSION":

            //Parent Refers Child Object 3
            //ClsFreeVersion IS-A ClsVirusScanner
            objVS = new ClsFreeVersion(); //Dynamic or Runtime Binding
            break;
        case "PAID_VERSION":

            //Parent Refers Child Object 1
            //ClsPaidVersion IS-A ClsVirusScanner
            objVS = new ClsPaidVersion(); //Dynamic or Runtime Binding
            break;
        case "TRIAL_VERSION":

            //Parent Refers Child Object 2
            objVS = new ClsTrialVersion(); //Dynamic or Runtime Binding
            break;
        }

        //Method fnStartScan() is the Version of ClsTrialVersion()
        objVS.fnStartScan();

    }
```

```
}
```

> Result :

```
Scan for Internet Security
Detect Virus & Malware
```

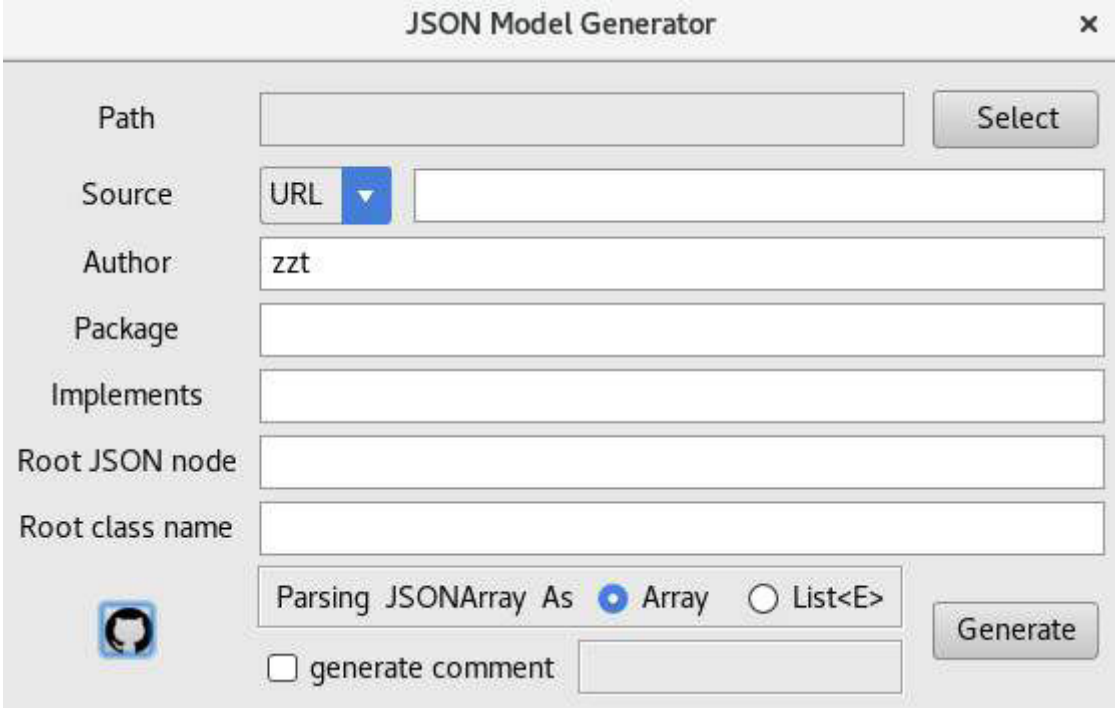
> Upcasting :

```
objVS = new ClsFreeVersion();
objVS = new ClsPaidVersion();
objVS = new ClsTrialVersion()
```

Chapter 154: Generating Java Code

Section 154.1: Generate POJO From JSON

- Install JSON Model Genrator plugin of Intellij by searching in Intellij setting.
- Start the plugin from 'Tools'
- Input the field of UI as following shows ('Path'、'Source'、'Package' is required):



- Click 'Generate' button and your are done.

Chapter 155: JShell

JShell is an interactive REPL for Java added in JDK 9. It allows developers to instantly evaluate expressions, test classes, and experiment with the Java language. Early access for jdk 9 can be obtained from: http://jdk.java.net/9/

Section 155.1: Editting Snippets

The basic unit of code used by JShell is the **snippet**, or **source entry**. Every time you declare a local variable or define a local method or class, you create a snippet whose name is the identifier of the variable/method/class. At any time, you can edit a snippet you have created with the `/edit` command. For example, let's say I have created the class `Foo` with a single, method, `bar`:

```
jshell> class Foo {
   ...> void bar() {
   ...> }
   ...> }
```

Now, I want to fill in the body of my method. Rather than rewrite the entire class, I can edit it:

```
jshell> /edit Foo
```

By default, a swing editor will pop up with the most basic features possible. However you can change the editor that JShell uses:

```
jshell> /set editor emacs
jshell> /set editor vi
jshell> /set editor nano
jshell> /set editor -default
```

Note that if **the new version of the snippet contains any syntax errors, it may not be saved.** Likewise, a snippet is only created if the original declaration/definition is syntactically correct; the following does not work:

```
jshell> String st = String 3
//error omitted
jshell> /edit st
|  No such snippet: st
```

However, snippets may be compiled and hence editable despite certain compile-time errors, such as mismatched types—the following works:

```
jshell> int i = "hello"
//error omitted
jshell> /edit i
```

Finally, snippets may be deleted using the `/drop` command:

```
jshell> int i = 13
jshell> /drop i
jshell> System.out.println(i)
|  Error:
|  cannot find symbol
|    symbol:   variable i
|  System.out.println(i)
|
```

To delete all snippets, thereby reseting the state of the JVM, use `\reset`:

```
jshell> int i = 2

jshell> String s = "hi"

jshell> /reset
|  Resetting state.

jshell> i
|  Error:
|  cannot find symbol
|    symbol:   variable i
|  i
|  ^

jshell> s
|  Error:
|  cannot find symbol
|    symbol:   variable s
|  s
|  ^
```

Section 155.2: Entering and Exiting JShell

Starting JShell

Before trying to start JShell, make sure your `JAVA_HOME` environment variable points to a JDK 9 installation. To start JShell, run the following command:

```
$ jshell
```

If all goes well, you should see a `jshell>` prompt.

Exiting JShell

To exit JShell, run the following command from the JShell prompt:

```
jshell> /exit
```

Section 155.3: Expressions

Within JShell, you can evaluate Java expressions, with or without semicolons. These can range from basic expressions and statements to more complex ones:

```
jshell> 4+2
jshell> System.out.printf("I am %d years old.\n", 421)
```

Loops and conditionals are fine, too:

```
jshell> for (int i = 0; i<3; i++) {
   ...> System.out.println(i);
   ...> }
```

It is important to note that **expressions within blocks must have semicolons!**

Section 155.4: Methods and Classes

You can define methods and classes within JShell:

```
jshell> void speak() {
   ...> System.out.println("hello");
   ...> }

jshell> class MyClass {
   ...> void doNothing() {}
   ...> }
```

No access modifiers are necessary. As with other blocks, semicolons are required inside of method bodies. Keep in mind that, as with variables, it is possible to redefine methods and classes. To see a list of methods or classes, enter `/methods` or `/types` at the JShell prompt, respectively.

Section 155.5: Variables

You can declare local variables within JShell:

```
jshell> String s = "hi"
jshell> int i = s.length
```

Keep in mind that variables can be redeclared with different types; this is perfectly valid in JShell:

```
jshell> String var = "hi"
jshell> int var = 3
```

To see a list of variables, enter `/vars` at the JShell prompt.

Chapter 156: Stack-Walking API

Prior to Java 9, access to the thread stack frames was limited to an internal class `sun.reflect.Reflection`. Specifically the method `sun.reflect.Reflection::getCallerClass`. Some libraries relies on this method which is deprecated.

An alternative standard API is now provided in JDK 9 via the `java.lang.StackWalker` class, and is designed to be efficient by allowing lazy access to the stack frames. Some applications may use this API to traverse the execution stack and filter on classes.

Section 156.1: Print all stack frames of the current thread

The following prints all stack frames of the current thread:

```
1  package test;
2
3  import java.lang.StackWalker.StackFrame;
4  import java.lang.reflect.InvocationTargetException;
5  import java.lang.reflect.Method;
6  import java.util.List;
7  import java.util.stream.Collectors;
8
9  public class StackWalkerExample {
10
11    public static void main(String[] args) throws NoSuchMethodException, SecurityException,
IllegalAccessException, IllegalArgumentException, InvocationTargetException {
12        Method fooMethod = FooHelper.class.getDeclaredMethod("foo", (Class<?>[])null);
13        fooMethod.invoke(null, (Object[]) null);
14    }
15 }
16
17 class FooHelper {
18    protected static void foo() {
19        BarHelper.bar();
20    }
21 }
22
23 class BarHelper {
24    protected static void bar() {
25        List<StackFrame> stack = StackWalker.getInstance()
26                .walk((s) -> s.collect(Collectors.toList()));
27        for(StackFrame frame : stack) {
28            System.out.println(frame.getClassName() + " " + frame.getLineNumber() + " " +
frame.getMethodName());
29        }
30    }
31 }
```

Output:

```
test.BarHelper 26 bar
test.FooHelper 19 foo
test.StackWalkerExample 13 main
```

Section 156.2: Print current caller class

The following prints the current caller class. Note that in this case, the `StackWalker` needs to be created with the option `RETAIN_CLASS_REFERENCE`, so that **`Class`** instances are retained in the `StackFrame` objects. Otherwise an exception would occur.

```
public class StackWalkerExample {

    public static void main(String[] args) {
        FooHelper.foo();
    }

}

class FooHelper {
    protected static void foo() {
        BarHelper.bar();
    }
}

class BarHelper {
    protected static void bar() {

System.out.println(StackWalker.getInstance(Option.RETAIN_CLASS_REFERENCE).getCallerClass());
    }
}
```

Output:

```
class test.FooHelper
```

Section 156.3: Showing reflection and other hidden frames

A couple of other options allow stack traces to include implementation and/or reflection frames. This may be useful for debugging purposes. For instance, we can add the `SHOW_REFLECT_FRAMES` option to the `StackWalker` instance upon creation, so that the frames for the reflective methods are printed as well:

```
package test;

import java.lang.StackWalker.Option;
import java.lang.StackWalker.StackFrame;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.List;
import java.util.stream.Collectors;

public class StackWalkerExample {

    public static void main(String[] args) throws NoSuchMethodException, SecurityException,
IllegalAccessException, IllegalArgumentException, InvocationTargetException {
        Method fooMethod = FooHelper.class.getDeclaredMethod("foo", (Class<?>[])null);
        fooMethod.invoke(null, (Object[]) null);
    }
}

class FooHelper {
    protected static void foo() {
```

```
        BarHelper.bar();
    }
}

class BarHelper {
    protected static void bar() {
        // show reflection methods
        List<StackFrame> stack = StackWalker.getInstance(Option.SHOW_REFLECT_FRAMES)
                .walk((s) -> s.collect(Collectors.toList()));
        for(StackFrame frame : stack) {
            System.out.println(frame.getClassName() + " " + frame.getLineNumber() + " " +
frame.getMethodName());
        }
    }
}
```

Output:

```
test.BarHelper 27 bar
test.FooHelper 20 foo
jdk.internal.reflect.NativeMethodAccessorImpl -2 invoke0
jdk.internal.reflect.NativeMethodAccessorImpl 62 invoke
jdk.internal.reflect.DelegatingMethodAccessorImpl 43 invoke
java.lang.reflect.Method 563 invoke
test.StackWalkerExample 14 main
```

Note that line numbers for some reflection methods may not be available so `StackFrame.getLineNumber()` may return negative values.

Chapter 157: Sockets

A socket is one end-point of a two-way communication link between two programs running on the network.

Section 157.1: Read from socket

```
String hostName = args[0];
int portNumber = Integer.parseInt(args[1]);

try (
    Socket echoSocket = new Socket(hostName, portNumber);
    PrintWriter out =
        new PrintWriter(echoSocket.getOutputStream(), true);
    BufferedReader in =
        new BufferedReader(
            new InputStreamReader(echoSocket.getInputStream()));
    BufferedReader stdIn =
        new BufferedReader(
            new InputStreamReader(System.in))
) {
    //Use the socket
}
```

Chapter 158: Java Sockets

Sockets are a low-level network interface that helps in creating a connection between two program mainly clients which may or may not be running on the same machine.

Socket Programming is one of the most widely used networking concepts.

Section 158.1: A simple TCP echo back server

Our TCP echo back server will be a separate thread. It's simple as its a start. It will just echo back whatever you send it but in capitalised form.

```
public class CAPECHOServer extends Thread{

    // This class implements server sockets. A server socket waits for requests to come
    // in over the network only when it is allowed through the local firewall
    ServerSocket serverSocket;

    public CAPECHOServer(int port, int timeout){
        try {
            // Create a new Server on specified port.
            serverSocket = new ServerSocket(port);
            // SoTimeout is basiacally the socket timeout.
            // timeout is the time until socket timeout in milliseconds
            serverSocket.setSoTimeout(timeout);
        } catch (IOException ex) {
            Logger.getLogger(CAPECHOServer.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    @Override
    public void run(){
        try {
            // We want the server to continuously accept connections
            while(!Thread.interrupted()){

            }
            // Close the server once done.
            serverSocket.close();
        } catch (IOException ex) {
            Logger.getLogger(CAPECHOServer.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

}
```

Now to accept connections. Let's update the run method.

```
@Override
public void run(){
    while(!Thread.interrupted()){
        try {
            // Log with the port number and machine ip
            Logger.getLogger((this.getClass().getName())).log(Level.INFO, "Listening for Clients at
{0} on {1}", new Object[]{serverSocket.getLocalPort(),
InetAddress.getLocalHost().getHostAddress()});
            Socket client = serverSocket.accept();  // Accept client conncetion
            // Now get DataInputStream and DataOutputStreams
```

```
            DataInputStream istream = new DataInputStream(client.getInputStream()); // From
client's input stream
            DataOutputStream ostream = new DataOutputStream(client.getOutputStream());
            // Important Note
            /*
                The server's input is the client's output
                The client's input is the server's output
            */
            // Send a welcome message
            ostream.writeUTF("Welcome!");

            // Close the connection
            istream.close();
            ostream.close();
            client.close();
        } catch (IOException ex) {
            Logger.getLogger(CAPECHOServer.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    // Close the server once done

    try {
        serverSocket.close();
    } catch (IOException ex) {
        Logger.getLogger(CAPECHOServer.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

Now if you can open telnet and try connecting You'll see a Welcome message.

You must connect with the port you specified and IP Adress.

You should see a result similar to this:

```
Welcome!

Connection to host lost.
```

Well, the connection was lost because we terminated it. Sometimes we would have to program our own TCP client. In this case, we need a client to request input from the user and send it across the network, receive the capitalised input.

If the server sends data first, then the client must read the data first.

```
public class CAPECHOClient extends Thread{

Socket server;
Scanner key; // Scanner for input

    public CAPECHOClient(String ip, int port){
        try {
            server = new Socket(ip, port);
            key = new Scanner(System.in);
        } catch (IOException ex) {
            Logger.getLogger(CAPECHOClient.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    @Override
```

```
    public void run(){
        DataInputStream istream = null;
        DataOutputStream ostream = null;
        try {
            istream = new DataInputStream(server.getInputStream()); // Familiar lines
            ostream = new DataOutputStream(server.getOutputStream());
            System.out.println(istream.readUTF());  // Print what the server sends
            System.out.print(">");
            String tosend = key.nextLine();
            ostream.writeUTF(tosend);   // Send whatever the user typed to the server
            System.out.println(istream.readUTF());  // Finally read what the server sends before
exiting.
        } catch (IOException ex) {
            Logger.getLogger(CAPECHOClient.class.getName()).log(Level.SEVERE, null, ex);
        } finally {
            try {
                istream.close();
                ostream.close();
                server.close();
            } catch (IOException ex) {
                Logger.getLogger(CAPECHOClient.class.getName()).log(Level.SEVERE, null, ex);
            }
        }
    }
}
```

Now update the server

```
ostream.writeUTF("Welcome!");

String inString = istream.readUTF();    // Read what the user sent
String outString = inString.toUpperCase();  // Change it to caps
ostream.writeUTF(outString);

// Close the connection
istream.close();
```

And now run the server and client, You should have an output similar to this

```
Welcome!
>
```

Chapter 159: FTP (File Transfer Protocol)

Parameters	Details
host	Either the host name or IP address of the FTP server
port	The FTP server port
username	The FTP server username
password	The FTP server password

Section 159.1: Connecting and Logging Into a FTP Server

To start using FTP with Java, you will need to create a new `FTPClient` and then connect and login to the server using `.connect(String server, int port)` and `.login(String username, String password)`.

```
import java.io.IOException;
import org.apache.commons.net.ftp.FTPClient;
import org.apache.commons.net.ftp.FTPReply;
//Import all the required resource for this project.

public class FTPConnectAndLogin {
    public static void main(String[] args) {
        // SET THESE TO MATCH YOUR FTP SERVER //
        String server = "www.server.com"; //Server can be either host name or IP address.
        int port = 21;
        String user = "Username";
        String pass = "Password";

        FTPClient ftp = new FTPClient;
        ftp.connect(server, port);
        ftp.login(user, pass);
    }
}
```

Now we have the basics done. But what if we have an error connecting to the server? We'll want to know when something goes wrong and get the error message. Let's add some code to catch errors while connecting.

```
try {
    ftp.connect(server, port);
    showServerReply(ftp);
    int replyCode = ftp.getReplyCode();
    if (!FTPReply.isPositiveCompletion(replyCode)) {
        System.out.printIn("Operation failed. Server reply code: " + replyCode)
        return;
    }
    ftp.login(user, pass);
} catch {

}
```

Let's break down what we just did, step by step.

```
showServerReply(ftp);
```

This refers to a function we will be making in a later step.

```
int replyCode = ftp.getReplyCode();
```

This grabs the reply/error code from the server and stores it as an integer.

```
if (!FTPReply.isPositiveCompletion(replyCode)) {
    System.out.printIn("Operation failed. Server reply code: " + replyCode)
    return;
}
```

This checks the reply code to see if there was an error. If there was an error, it will simply print "Operation failed. Server reply code: " followed by the error code. We also added a try/catch block which we will add to in the next step. Next, let's also create a function that checks `ftp.login()` for errors.

```
boolean success = ftp.login(user, pass);
showServerReply(ftp);
if (!success) {
    System.out.println("Failed to log into the server");
        return;
    } else {
        System.out.println("LOGGED IN SERVER");
    }
```

Let's break this block down too.

```
boolean success = ftp.login(user, pass);
```

This will not just attempt to login to the FTP server, it will also store the result as a boolean.

```
showServerReply(ftp);
```

This will check if the server sent us any messages, but we will first need to create the function in the next step.

```
if (!success) {
System.out.println("Failed to log into the server");
    return;
} else {
    System.out.println("LOGGED IN SERVER");
}
```

This statement will check if we logged in successfully; if so, it will print "LOGGED IN SERVER", otherwise it will print "Failed to log into the server". This is our script so far:

```
import java.io.IOException;
import org.apache.commons.net.ftp.FTPClient;
import org.apache.commons.net.ftp.FTPReply;

public class FTPConnectAndLogin {
    public static void main(String[] args) {
        // SET THESE TO MATCH YOUR FTP SERVER //
        String server = "www.server.com";
        int port = 21;
        String user = "username"
        String pass = "password"

        FTPClient ftp = new FTPClient
        try {
            ftp.connect(server, port)
            showServerReply(ftp);
            int replyCode = ftpClient.getReplyCode();
            if (!FTPReply.isPositiveCompletion(replyCode)) {
```

```
                System.out.println("Operation failed. Server reply code: " + replyCode);
                return;
            }
            boolean success = ftp.login(user, pass);
            showServerReply(ftp);
            if (!success) {
                System.out.println("Failed to log into the server");
                return;
            } else {
                System.out.println("LOGGED IN SERVER");
            }
        } catch {

        }
    }
}
```

Now next let's create complete the Catch block in case we run into any errors with the whole process.

```
} catch (IOException ex) {
    System.out.println("Oops! Something went wrong.");
    ex.printStackTrace();
}
```

The completed catch block will now print "Oops! Something went wrong." and the stacktrace if there is an error. Now our final step is to create the `showServerReply()` we have been using for a while now.

```
private static void showServerReply(FTPClient ftp) {
    String[] replies = ftp.getReplyStrings();
    if (replies != null && replies.length > 0) {
    for (String aReply : replies) {
        System.out.println("SERVER: " + aReply);
        }
    }
}
```

This function takes an `FTPClient` as a variable, and calls it "ftp". After that it stores any server replies from the server in a string array. Next it checks if any messages were stored. If there is any, it prints each of them as "SERVER: [reply]". Now that we have that function done, this is the completed script:

```
import java.io.IOException;
import org.apache.commons.net.ftp.FTPClient;
import org.apache.commons.net.ftp.FTPReply;

public class FTPConnectAndLogin {
    private static void showServerReply(FTPClient ftp) {
        String[] replies = ftp.getReplyStrings();
        if (replies != null && replies.length > 0) {
        for (String aReply : replies) {
            System.out.println("SERVER: " + aReply);
            }
        }
    }

    public static void main(String[] args) {
        // SET THESE TO MATCH YOUR FTP SERVER //
        String server = "www.server.com";
        int port = 21;
        String user = "username"
        String pass = "password"
```

```
        FTPClient ftp = new FTPClient
        try {
            ftp.connect(server, port)
            showServerReply(ftp);
            int replyCode = ftpClient.getReplyCode();
            if (!FTPReply.isPositiveCompletion(replyCode)) {
                System.out.println("Operation failed. Server reply code: " + replyCode);
                return;
            }
            boolean success = ftp.login(user, pass);
            showServerReply(ftp);
            if (!success) {
                System.out.println("Failed to log into the server");
                return;
            } else {
                System.out.println("LOGGED IN SERVER");
            }
        } catch (IOException ex) {
            System.out.println("Oops! Something went wrong.");
            ex.printStackTrace();
        }
    }
}
```

We first need to create a new `FTPClient` and try connecting to the server it and logging into it using `.connect(String server, int port)` and `.login(String username, String password)`. It is important to connect and login using a try/catch block in case our code fails to connect with the server. We will also need to create a function that checks and displays any messages we may receive from the server as we try connecting and logging in. We will call this function "`showServerReply(FTPClient ftp)`".

```
import java.io.IOException;
import org.apache.commons.net.ftp.FTPClient;
import org.apache.commons.net.ftp.FTPReply;

public class FTPConnectAndLogin {
    private static void showServerReply(FTPClient ftp) {
        if (replies != null && replies.length > 0) {
        for (String aReply : replies) {
            System.out.println("SERVER: " + aReply);
            }
        }
    }

public static void main(String[] args) {
    // SET THESE TO MATCH YOUR FTP SERVER //
    String server = "www.server.com";
    int port = 21;
    String user = "username"
    String pass = "password"

FTPClient ftp = new FTPClient
try {
    ftp.connect(server, port)
    showServerReply(ftp);
    int replyCode = ftpClient.getReplyCode();
    if (!FTPReply.isPositiveCompletion(replyCode)) {
            System.out.println("Operation failed. Server reply code: " + replyCode);
            return;
        }
    boolean success = ftp.login(user, pass);
    showServerReply(ftp);
```

```
    if (!success) {
        System.out.println("Failed to log into the server");
            return;
        } else {
            System.out.println("LOGGED IN SERVER");
        }
    } catch (IOException ex) {
        System.out.println("Oops! Something went wrong.");
        ex.printStackTrace();
        }
    }
}
```

After this, you should now have your FTP server connected to you Java script.

Chapter 160: Using Other Scripting Languages in Java

Java in itself is an extremely powerful language, but its power can further be extended Thanks to JSR223 (Java Specification Request 223) introducing a script engine

Section 160.1: Evaluating A JavaScript file in -scripting mode of nashorn

```
public class JSEngine {

    /*
    * Note Nashorn is only available for Java-8 onwards
    * You can use rhino from ScriptEngineManager.getEngineByName("js");
    */

    ScriptEngine engine;
    ScriptContext context;
    public Bindings scope;

    // Initialize the Engine from its factory in scripting mode
    public JSEngine(){
        engine = new NashornScriptEngineFactory().getScriptEngine("-scripting");
        // Script context is an interface so we need an implementation of it
        context = new SimpleScriptContext();
        // Create bindings to expose variables into
        scope = engine.createBindings();
    }

    // Clear the bindings to remove the previous variables
    public void newBatch(){
        scope.clear();
    }

    public void execute(String file){
        try {
            // Get a buffered reader for input
            BufferedReader br = new BufferedReader(new FileReader(file));
            // Evaluate code, with input as bufferdReader
            engine.eval(br);
        } catch (FileNotFoundException ex) {
            Logger.getLogger(JSEngine.class.getName()).log(Level.SEVERE, null, ex);
        } catch (ScriptException ex) {
            // Script Exception is basically when there is an error in script
            Logger.getLogger(JSEngine.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    public void eval(String code){
        try {
            // Engine.eval basically treats any string as a line of code and evaluates it, executes
it
            engine.eval(code);
        } catch (ScriptException ex) {
            // Script Exception is basically when there is an error in script
            Logger.getLogger(JSEngine.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
```

```
    // Apply the bindings to the context and set the engine's default context
    public void startBatch(int SCP){
        context.setBindings(scope, SCP);
        engine.setContext(context);
    }

    // We use the invocable interface to access methods from the script
    // Invocable is an optional interface, please check if your engine implements it
    public Invocable invocable(){
        return (Invocable)engine;
    }

}
```

Now the main method

```
public static void main(String[] args) {
    JSEngine jse = new JSEngine();
    // Create a new batch probably unecessary
    jse.newBatch();
    // Expose variable x into script with value of hello world
    jse.scope.put("x", "hello world");
    // Apply the bindings and start the batch
    jse.startBatch(ScriptContext.ENGINE_SCOPE);
    // Evaluate the code
    jse.eval("print(x);");
}
```

Your output should be similar to this

```
hello world
```

As you can see the exposed variable x has been printed. Now testing with a file.

> Here we have test.js

```
print(x);
function test(){
    print("hello test.js:test");
}
test();
```

And the updated main method

```
public static void main(String[] args) {
    JSEngine jse = new JSEngine();
    // Create a new batch probably unecessary
    jse.newBatch();
    // Expose variable x into script with value of hello world
    jse.scope.put("x", "hello world");
    // Apply the bindings and start the batch
    jse.startBatch(ScriptContext.ENGINE_SCOPE);
    // Evaluate the code
    jse.execute("./test.js");
}
```

Assuming that test.js is in the same directory as your application You should have output similar to this

```
hello world
hello test.js:test
```

Chapter 161: C++ Comparison

Java and C++ are similar languages. This topic serves as a quick reference guide for Java and C++ Engineers.

Section 161.1: Static Class Members

Static members have class scope as opposed to object scope

C++ Example

```
// define in header
class Singleton {
   public:
      static Singleton *getInstance();

   private:
      Singleton() {}
      static Singleton *instance;
};

// initialize in .cpp
Singleton* Singleton::instance = 0;
```

Java Example

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Section 161.2: Classes Defined within Other Constructs

Defined within Another Class

C++

Nested Class[ref] (needs a reference to enclosing class)

```
class Outer {
   class Inner {
      public:
         Inner(Outer* o) :outer(o) {}

      private:
         Outer*  outer;
   };
};
```

Java

[non-static] Nested Class (aka Inner Class or Member Class)

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
}
```

Statically Defined within Another Class

C++

Static Nested Class

```
class Outer {
   class Inner {
      ...
   };
};
```

Java

Static Nested Class (aka Static Member Class)[ref]

```
class OuterClass {
    ...
    static class StaticNestedClass {
        ...
    }
}
```

Defined within a Method

(e.g. event handling)

C++

Local Class[ref]

```
void fun() {
   class Test {
      /* members of Test class */
   };
}
```

Java

Local Class[ref]

```
class Test {
    void f() {
        new Thread(new Runnable() {
            public void run() {
                doSomethingBackgroundish();
            }
        }).start();
    }
}
```

Section 161.3: Pass-by-value & Pass-by-reference

Many argue that Java is ONLY pass-by-value, but it's more nuanced than that. Compare the following C++ and Java examples to see the many flavors of pass-by-value (aka copy) and pass-by-reference (aka alias).

C++ Example (complete code)

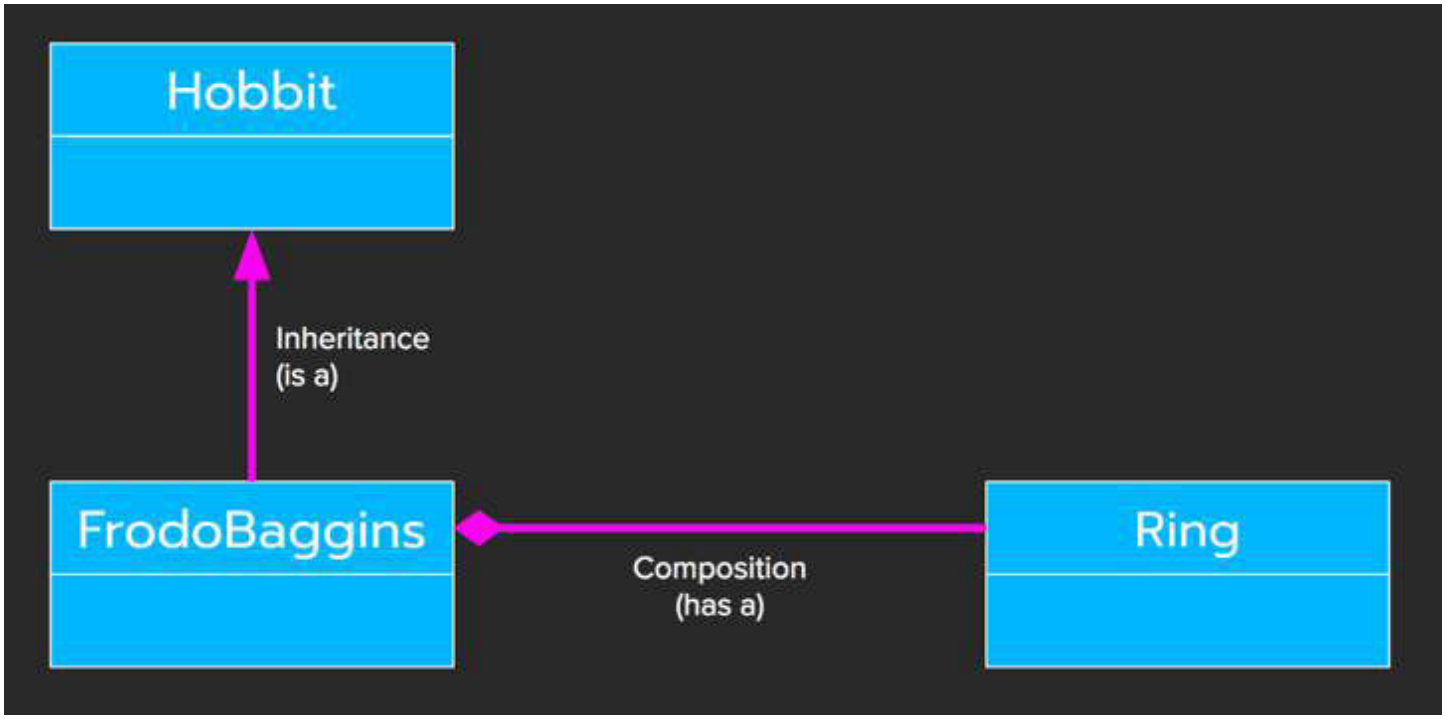
```
// passes a COPY of the object
static void passByCopy(PassIt obj) {
   obj.i = 22;  // only a "local" change
}

// passes a pointer
static void passByPointer(PassIt* ptr) {
   ptr->i = 33;
   ptr = 0; // better to use nullptr instead if '0'
}

// passes an alias (aka reference)
static void passByAlias(PassIt& ref) {
   ref.i = 44;
}

// This is an old-school way of doing it.
// Check out std::swap for the best way to do this
static void swap(PassIt** pptr1, PassIt** pptr2) {
   PassIt* tmp = *pptr1;
   *pptr1 = *pptr2;
   *pptr2 = tmp;
}
```

Java Example (complete code)

```
// passes a copy of the variable
// NOTE: in java only primitives are pass-by-copy
public static void passByCopy(int copy) {
   copy = 33;  // only a "local" change
}

// No such thing as pointers in Java
/*
public static void passByPointer(PassIt *ptr) {
   ptr->i = 33;
   ptr = 0; // better to use nullptr instead if '0'
}
*/

// passes an alias (aka reference)
public static void passByAlias(PassIt ref) {
   ref.i = 44;
}

// passes aliases (aka references),
// but need to do "manual", potentially expensive copies
public static void swap(PassIt ref1, PassIt ref2) {
   PassIt tmp = new PassIt(ref1);
   ref1.copy(ref2);
   ref2.copy(tmp);
}
```

Section 161.4: Inheritance vs Composition

C++ & Java are both object-oriented languages, thus the following diagram applies to both.



Section 161.5: Outcast Downcasting

Beware of using "downcasting" - Downcasting is casting down the inheritance hierarchy from a base class to a subclass (i.e. opposite of polymorphism). In general, use polymorphism & overriding instead of instanceof & downcasting.

C++ Example

```
// explicit type case required
Child *pChild =  (Child *) &parent;
```

Java Example

```
if(mySubClass instanceof SubClass) {
   SubClass mySubClass = (SubClass)someBaseClass;
   mySubClass.nonInheritedMethod();
}
```

Section 161.6: Abstract Methods & Classes

Abstract Method

declared without an implementation

C++

pure virtual method

```
virtual void eat(void) = 0;
```

Java

abstract method

```
abstract void draw();
```

Abstract Class

cannot be instantiated

C++

cannot be instantiated; has at least 1 pure virtual method

```
class AB {public: virtual void f() = 0;};
```

Java

cannot be instantiated; can have non-abstract methods

```
abstract class GraphicObject {}
```

Interface

no instance fields

C++

nothing comparable to Java

Java

very similar to abstract class, but 1) supports multiple inheritance; 2) no instance fields

```
interface TestInterface {}
```

Chapter 162: Audio

Section 162.1: Play a MIDI file

MIDI files can be played by using several classes from the `javax.sound.midi` package. A `Sequencer` performs playback of the MIDI file, and many of its methods can be used to set playback controls such as loop count, tempo, track muting, and others.

General playback of MIDI data can be done in this way:

```
import java.io.File;
import java.io.IOException;
import javax.sound.midi.InvalidMidiDataException;
import javax.sound.midi.MidiSystem;
import javax.sound.midi.MidiUnavailableException;
import javax.sound.midi.Sequence;
import javax.sound.midi.Sequencer;

public class MidiPlayback {
    public static void main(String[] args) {
        try {
            Sequencer sequencer = MidiSystem.getSequencer(); // Get the default Sequencer
            if (sequencer==null) {
                System.err.println("Sequencer device not supported");
                return;
            }
            sequencer.open(); // Open device
            // Create sequence, the File must contain MIDI file data.
            Sequence sequence = MidiSystem.getSequence(new File(args[0]));
            sequencer.setSequence(sequence); // load it into sequencer
            sequencer.start();  // start the playback
        } catch (MidiUnavailableException | InvalidMidiDataException | IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

To stop the playback use:

```
sequencer.stop(); // Stop the playback
```

A sequencer can be set to mute one or more of the sequence's tracks during playback so none of the instruments in those specified play. The following example sets the first track in the sequence to be muted:

```
import javax.sound.midi.Track;
// ...

Track[] track = sequence.getTracks();
sequencer.setTrackMute(track[0]);
```

A sequencer can play a sequence repeatedly if the loop count is given. The following sets the sequencer to play a sequence four times and indefinitely:

```
sequencer.setLoopCount(3);
sequencer.setLoopCount(Sequencer.LOOP_CONTINUOUSLY);
```

The sequencer does not always have to play the sequence from the beginning, nor does it have to play the

sequence until the end. It can start and end at any point by specifying the *tick* in the sequence to start and end at. It is also possible to specify manually which tick in the sequence the sequencer should play from:

```
sequencer.setLoopStartPoint(512);
sequencer.setLoopEndPoint(32768);
sequencer.setTickPosition(8192);
```

Sequencers can also play a MIDI file at a certain tempo, which can be controlled by specifying the tempo in beats per minute (BPM) or microseconds per quarter note (MPQ). The factor at which the sequence is played can be adjusted as well.

```
sequencer.setTempoInBPM(1250f);
sequencer.setTempoInMPQ(4750f);
sequencer.setTempoFactor(1.5f);
```

When you finished using the `Sequencer`, remeber to close it

```
sequencer.close();
```

Section 162.2: Play an Audio file Looped

Needed imports:

```
import javax.sound.sampled.AudioSystem;
import javax.sound.sampled.Clip;
```

This code will create a clip and play it continuously once started:

```
Clip clip = AudioSystem.getClip();
clip.open(AudioSystem.getAudioInputStream(new URL(filename)));
clip.start();
clip.loop(Clip.LOOP_CONTINUOUSLY);
```

Get an Array with all supported file types:

```
AudioFileFormat.Type [] audioFileTypes = AudioSystem.getAudioFileTypes();
```

Section 162.3: Basic audio output

The Hello Audio! of Java that plays a sound file from local or internet storage looks as follows. It works for uncompressed .wav files and should not be used for playing mp3 or compressed files.

```
import java.io.*;
import java.net.URL;
import javax.sound.sampled.*;

public class SoundClipTest {

   // Constructor
   public SoundClipTest() {
      try {
         // Open an audio input stream.
          File soundFile = new File("/usr/share/sounds/alsa/Front_Center.wav"); //you could also
get the sound file with an URL
          AudioInputStream audioIn = AudioSystem.getAudioInputStream(soundFile);
          AudioFormat format = audioIn.getFormat();
```

```
            // Get a sound clip resource.
            DataLine.Info info = new DataLine.Info(Clip.class, format);
            Clip clip = (Clip)AudioSystem.getLine(info);
            // Open audio clip and load samples from the audio input stream.
            clip.open(audioIn);
            clip.start();
        } catch (UnsupportedAudioFileException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (LineUnavailableException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        new SoundClipTest();
    }
}
```

Section 162.4: Bare metal sound

You can also go almost bare-metal when producing sound with java. This code will write raw binary data into the OS audio buffer to generate sound. It's extremely important to understand the limitations and necessary calculations to generating sound like this. Since playback is basically instantaneous, calculations need to be performed at almost real-time.

As such this method is unusable for more complicated sound-sampling. For such purposes using specialized tools is the better approach.

The following method generates and directly outputs a rectangle-wave of a given frequency in a given volume for a given duration.

```
public void rectangleWave(byte volume, int hertz, int msecs) {
    final SourceDataLine dataLine;
    // 24 kHz x 8bit, single-channel, signed little endian AudioFormat
    AudioFormat af = new AudioFormat(24_000, 8, 1, true, false);
    try {
        dataLine = AudioSystem.getSourceDataLine(af);
        dataLine.open(af, 10_000); // audio buffer size: 10k samples
    } catch (LineUnavailableException e) {
        throw new RuntimeException(e);
    }

    int waveHalf = 24_000 / hertz; // samples for half a period
    byte[] buffer = new byte[waveHalf * 20];
    int samples = msecs * (24_000 / 1000); // 24k (samples / sec) / 1000 (ms/sec) * time(ms)

    dataLine.start(); // starts playback
    int sign = 1;

    for (int i = 0; i < samples; i += buffer.length) {
        for (int j = 0; j < 20; j++) { // generate 10 waves into buffer
            sign *= -1;
            // fill from the jth wave-half to the j+1th wave-half with volume
            Arrays.fill(buffer, waveHalf * j, waveHalf * (j+1), (byte) (volume * sign));
        }
        dataLine.write(buffer, 0, buffer.length); //
    }
    dataLine.drain(); // forces buffer drain to hardware
```

```
    dataLine.stop();  // ends playback
}
```

For a more differentiated way to generate different soundwaves sinus calculations and possibly larger sample sizes are necessary. This results in significantly more complex code and is accordingly omitted here.

Chapter 163: Java Print Service

The Java Print Service API provides functionalities to discover print services and send print requests for them.

It includes extensible print attributes based on the standard attributes specified in the Internet Printing Protocol (IPP) 1.1 from the IETF Specification, RFC 2911.

Section 163.1: Building the Doc that will be printed

`Doc` is an interface and the Java Print Service API provide a simple implementation called `SimpleDoc`.

Every `Doc` instance is basically made of two aspects:

- the print data content itself (an E-mail, an image, a document etc)
- the print data format, called `DocFlavor` (MIME type + Representation class).

Before creating the `Doc` object, we need to load our document from somewhere. In the example, we will load an specific file from the disk:

```
FileInputStream pdfFileInputStream = new FileInputStream("something.pdf");
```

So now, we have to choose a `DocFlavor` that matches our content. The `DocFlavor` class has a bunch of constants to represent the most usual types of data. Let's pick the `INPUT_STREAM.PDF` one:

```
DocFlavor pdfDocFlavor = DocFlavor.INPUT_STREAM.PDF;
```

Now, we can create a new instance of `SimpleDoc`:

```
Doc doc = new SimpleDoc(pdfFileInputStream, pdfDocFlavor , null);
```

The `doc` object now can be sent to the print job request (see Creating a print job from a print service).

Section 163.2: Discovering the available print services

To discovery all the available print services, we can use the `PrintServiceLookup` class. Let's see how:

```
import javax.print.PrintService;
import javax.print.PrintServiceLookup;

public class DiscoveringAvailablePrintServices {

    public static void main(String[] args) {
        discoverPrintServices();
    }

    public static void discoverPrintServices() {
        PrintService[] allPrintServices = PrintServiceLookup.lookupPrintServices(null, null);

        for (Printservice printService : allPrintServices) {
            System.out.println("Print service name: " + printService.getName());
        }
    }

}
```

This program, when executed on a Windows environment, will print something like this:

```
Print service name: Fax
Print service name: Microsoft Print to PDF
Print service name: Microsoft XPS Document Viewer
```

Section 163.3: Defining print request attributes

Sometimes we need to determine some aspects of the print request. We will call them *attribute*.

Are examples of print request attributes:

- amount of copies (1, 2 etc),
- orientation (portrait or landscape)
- chromacity (monochrome, color)
- quality (draft, normal, high)
- sides (one-sided, two-sided etc)
- and so on...

Before choosing one of them and which value each one will have, first we need to build a set of attributes:

```
PrintRequestAttributeSet pras = new HashPrintRequestAttributeSet();
```

Now we can add them. Some examples are:

```
pras.add(new Copies(5));
pras.add(MediaSize.ISO_A4);
pras.add(OrientationRequested.PORTRAIT);
pras.add(PrintQuality.NORMAL);
```

The `pras` object now can be sent to the print job request (see Creating a print job from a print service).

Section 163.4: Listening print job request status change

For the most printing clients, is extremely useful to know if a print job has finished or failed.

The Java Print Service API provide some functionalities to get informed about these scenarios. All we have to do is:

- provide an implementation for `PrintJobListener` interface and
- register this implementation at the print job.

When the print job state changes, we will be notified. We can do anything is needed, for example:

- update a user interface,
- start another business process,
- record something in the database,
- or simply log it.

In the example bellow, we will log every print job status change:

```
import javax.print.event.PrintJobEvent;
import javax.print.event.PrintJobListener;

public class LoggerPrintJobListener implements PrintJobListener {

```

```
    // Your favorite Logger class goes here!
    private static final Logger LOG = Logger.getLogger(LoggerPrintJobListener.class);


    public void printDataTransferCompleted(PrintJobEvent pje) {
        LOG.info("Print data transfer completed ;) ");
    }

    public void printJobCompleted(PrintJobEvent pje) {
        LOG.info("Print job completed =) ");
    }

    public void printJobFailed(PrintJobEvent pje) {
        LOG.info("Print job failed =( ");
    }

    public void printJobCanceled(PrintJobEvent pje) {
        LOG.info("Print job canceled :| ");
    }

    public void printJobNoMoreEvents(PrintJobEvent pje) {
        LOG.info("No more events to the job ");
    }

    public void printJobRequiresAttention(PrintJobEvent pje) {
        LOG.info("Print job requires attention :O ");
    }
}
```

Finally, we can add our print job listener implementation on the print job before the print request itself, as follows:

```
DocPrintJob printJob = printService.createPrintJob();

printJob.addPrintJobListener(new LoggerPrintJobListener());

printJob.print(doc, pras);
```

The *PrintJobEvent pje* argument

Notice that every method has a `PrintJobEvent pje` argument. We don't use it in this example for simplicity purposes, but you can use it to explore the status. For example:

```
pje.getPrintJob().getAttributes();
```

Will return a `PrintJobAttributeSet` object instance and you can run them in a for-each way.

Another way to achieve the same goal

Another option to achieve the same goal is extending the `PrintJobAdapter` class, as the name says, is an adapter for `PrintJobListener`. Implementing the interface we compulsorily have to implement all of them. The advantage of this way it's we need to override only the methods we want. Let's see how it works:

```
import javax.print.event.PrintJobEvent;
import javax.print.event.PrintJobAdapter;

public class LoggerPrintJobAdapter extends PrintJobAdapter {

    // Your favorite Logger class goes here!
```

```
    private static final Logger LOG = Logger.getLogger(LoggerPrintJobAdapter.class);

    public void printJobCompleted(PrintJobEvent pje) {
        LOG.info("Print job completed =) ");
    }

    public void printJobFailed(PrintJobEvent pje) {
        LOG.info("Print job failed =( ");
    }
}
```

Notice that we override only some specific methods.

As the same way in the example implementing the interface `PrintJobListener`, we add the listener to the print job before sending it to print:

```
printJob.addPrintJobListener(new LoggerPrintJobAdapter());

printJob.print(doc, pras);
```

Section 163.5: Discovering the default print service

To discovery the default print service, we can use the `PrintServiceLookup` class. Let's see how::

```
import javax.print.PrintService;
import javax.print.PrintServiceLookup;

public class DiscoveringDefaultPrintService {

    public static void main(String[] args) {
        discoverDefaultPrintService();
    }

    public static void discoverDefaultPrintService() {
        PrintService defaultPrintService = PrintServiceLookup.lookupDefaultPrintService();
        System.out.println("Default print service name: " + defaultPrintService.getName());
    }

}
```

Section 163.6: Creating a print job from a print service

A print job is a request of printing something in a specific print service. It consists, basically, by:

- the data that will be printed (see Building the Doc that will be printed)
- a set of attributes

After picking-up the right print service instance, we can request the creation of a print job:

```
DocPrintJob printJob = printService.createPrintJob();
```

The `DocPrintJob` interface provide us the `print` method:

```
printJob.print(doc, pras);
```

The `doc` argument is a `Doc`: the data that will be printed.

And the `pras` argument is a `PrintRequestAttributeSet` interface: a set of `PrintRequestAttribute`. Are examples of print request attributes:

- amount of copies (1, 2 etc),
- orientation (portrait or landscape)
- chromacity (monochrome, color)
- quality (draft, normal, high)
- sides (one-sided, two-sided etc)
- and so on...

The print method may throw a `PrintException`.

Chapter 164: CompletableFuture

CompletableFuture is a class added to Java SE 8 which implements the Future interface from Java SE 5. In addition to supporting the Future interface it adds many methods that allow asynchronous callback when the future is completed.

Section 164.1: Simple Example of CompletableFuture

In the example below, the `calculateShippingPrice` method calculates shipping cost, which takes some processing time. In a real world example, this would e.g. be contacting another server which returns the price based on the weight of the product and the shipping method.

By modeling this in an async way via `CompletableFuture`, we can continue different work in the method (i.e. calculating packaging costs).

```
public static void main(String[] args) {
    int price = 15; // Let's keep it simple and work with whole number prices here
    int weightInGrams = 900;

    calculateShippingPrice(weightInGrams) // Here, we get the future
        .thenAccept(shippingPrice -> { // And then immediately work on it!
            // This fluent style is very useful for keeping it concise
            System.out.println("Your total price is: " + (price + shippingPrice));
        });
    System.out.println("Please stand by. We are calculating your total price.");
}


public static CompletableFuture<Integer> calculateShippingPrice(int weightInGrams) {
    return CompletableFuture.supplyAsync(() -> {
        // supplyAsync is a factory method that turns a given
        // Supplier<U> into a CompletableFuture<U>

        // Let's just say each 200 grams is a new dollar on your shipping costs
        int shippingCosts = weightInGrams / 200;

        try {
            Thread.sleep(2000L); // Now let's simulate some waiting time...
        } catch(InterruptedException e) { /* We can safely ignore that */ }

        return shippingCosts; // And send the costs back!
    });
}
```

Chapter 165: Runtime Commands

Section 165.1: Adding shutdown hooks

Sometimes you need a piece of code to execute when the program stops, such as releasing system resources that you open. You can make a thread run when the program stops with the `addShutdownHook` method:

```
Runtime.getRuntime().addShutdownHook(new Thread(() -> {
    ImportantStuff.someImportantIOStream.close();
}));
```

Chapter 166: Unit Testing

Unit testing is an integral part of test-driven development, and an important feature for building any robust application. In Java, Unit testing is almost exclusively performed using external libraries and frameworks, most of which have their own documentation tag. This stub serves as a means of introducing the reader to the tools available, and their respective documentation.

Section 166.1: What is Unit Testing?

This is a bit of a primer. It's mostly put it in because documentation is forced to have an example, even if it's intended as a stub article. If you already know unit-testing basics, feel free to skip forward to the remarks, where specific frameworks are mentioned.

Unit testing is ensuring that a given module behaves as expected. In large-scale applications, ensuring the appropriate execution of modules in a vacuum is an integral part of ensuring application fidelity.

Consider the following (trivial) pseudo-example:

```
public class Example {
  public static void main (String args[]) {
    new Example();
  }

  // Application-level test.
  public Example() {
    Consumer c = new Consumer();
    System.out.println("VALUE = " + c.getVal());
  }

  // Your Module.
  class Consumer {
    private Capitalizer c;

    public Consumer() {
      c = new Capitalizer();
    }

    public String getVal() {
      return c.getVal();
    }
  }

  // Another team's module.
  class Capitalizer {
    private DataReader dr;

    public Capitalizer() {
      dr = new DataReader();
    }

    public String getVal() {
      return dr.readVal().toUpperCase();
    }
  }

  // Another team's module.
  class DataReader {
    public String readVal() {
```

```
      // Refers to a file somewhere in your application deployment, or
      // perhaps retrieved over a deployment-specific network.
      File f;
      String s = "data";
      // ... Read data from f into s ...
      return s;
    }
  }
}
```

So this example is trivial; `DataReader` gets the data from a file, passes it to the `Capitalizer`, which converts all the characters to upper-case, which then gets passed to the `Consumer`. But the `DataReader` is heavily-linked to our application environment, so we defer testing of this chain until we are ready to deploy a test release.

Now, assume, somewhere along the way in a release, for reasons unknown, the `getVal()` method in `Capitalizer` changed from returning a `toUpperCase()` String to a `toLowerCase()` String:

```
// Another team's module.
class Capitalizer {
  ...

  public String getVal() {
    return dr.readVal().toLowerCase();
  }
}
```

Clearly, this breaks expected behavior. But, because of the arduous processes involved with execution of the `DataReader`, we won't notice this until our next test deployment. So days/weeks/months go by with this bug sitting in our system, and then the product manager sees this, and instantly turns to you, the team leader associated with the `Consumer`. "Why is this happening? What did you guys change?" Obviously, you're clueless. You have no idea what's going on. You didn't change any code that should be touching this; why is it suddenly broken?

Eventually, after discussion between the teams and collaboration, the issue is traced, and the problem solved. But, it begs the question; how could this have been prevented?

There are two obvious things:

Tests need to be automated

Our reliance upon manual testing let this bug go by unnoticed far too long. We need a way to automate the process under which bugs are introduced **instantly**. Not 5 weeks from now. Not 5 days from now. Not 5 minutes from now. Right now.

You have to appreciate that, in this example, I've expressed one **very trivial** bug that was introduced and unnoticed. In an industrial application, with dozens of modules constantly being updated, these can creep in all over the place. You fix something with one module, only to realize that the very behavior you "fixed" was relied upon in some manner elsewhere (either internally or externally).

Without rigorous validation, things will creep into the system. It's possible that, if neglected far enough, this will result in so much extra work trying to fix changes (and then fixing those fixes, etc.), that a product will actually **increase** in remaining work as effort is put into it. You do not want to be in this situation.

Tests need to be fine-grained

The second problem noted in our above example is the amount of time it took to trace the bug. The product manager pinged you when the testers noticed it, you investigated and found that the `Capitalizer` was returning

seemingly bad data, you pinged the `Capitalizer` team with your findings, they investigated, etc. etc. etc.

The same point I made above about the quantity and difficulty of this trivial example hold here. Obviously anyone reasonably well-versed with Java could find the introduced problem quickly. But it's often much, much more difficult to trace and communicate issues. Maybe the `Capitalizer` team provided you a JAR with no source. Maybe they're located on the other side of the world, and communication hours are very limited (perhaps to e-mails that get sent once daily). It can result in bugs taking weeks or longer to trace (and, again, there could be several of these for a given release).

In order to mitigate against this, we want rigorous testing on as **fine** a level as possible (you also want coarse-grained testing to ensure modules interact properly, but that's not our focal point here). We want to rigorously specify how all outward-facing functionality (at minimum) operates, and tests for that functionality.

Enter unit-testing

Imagine if we had a test, specifically ensuring that the `getVal()` method of `Capitalizer` returned a capitalized string for a given input string. Furthermore, imagine that test was run before we even committed any code. The bug introduced into the system (that is, `toUpperCase()` being replaced with `toLowerCase()`) would cause no issues because the bug would never **be** introduced into the system. We would catch it in a test, the developer would (hopefully) realize their mistake, and an alternative solution would be reached as to how to introduce their intended effect.

There's some omissions made here as to **how** to implement these tests, but those are covered in the framework-specific documentation (linked in the remarks). Hopefully, this serves as an example of **why** unit testing is important.

Chapter 167: Asserting

Parameter	Details
expression1	The assertion statement throws an `AssertionError` if this expression evaluates to **`false`**.
expression2	Optional. When used, `AssertionErrors` thrown by the assert statement have this message.

Section 167.1: Checking arithmetic with assert

```
a = 1 - Math.abs(1 - a % 2);

// This will throw an error if my arithmetic above is wrong.
assert a >= 0 && a <= 1 : "Calculated value of " + a + " is outside of expected bounds";

return a;
```

Chapter 168: Multi-Release JAR Files

One of the features introduced in Java 9 is the multi-release Jar (MRJAR) which allows bundling code targeting multiple Java releases within the same Jar file. The feature is specified in JEP 238.

Section 168.1: Example of a multi-release Jar file's contents

By setting `Multi-Release: true` in the MANIFEST.MF file, the Jar file becomes a multi-release Jar and the Java runtime (as long as it supports the MRJAR format) will pick the appropriate versions of classes depending on the current major version.

The structure of such a Jar is the following:

```
jar root
  - A.class
  - B.class
  - C.class
  - D.class
  - META-INF
     - versions
        - 9
           - A.class
           - B.class
        - 10
           - A.class
```

- On JDKs < 9, only the classes in the root entry are visible to the Java runtime.
- On a JDK 9, the classes A and B will be loaded from the directory `root/META-INF/versions/9`, while C and D will be loaded from the base entry.
- On a JDK 10, class A would be loaded from the directory `root/META-INF/versions/10`.

Section 168.2: Creating a multi-release Jar using the jar tool

The `jar` command can be used to create a multi-release Jar containing two versions of the same class compiled for both Java 8 and Java 9, albeit with a warning telling that the classes are identical:

```
C:\Users\manouti>jar --create --file MR.jar -C sampleproject-base demo --release 9 -C
sampleproject-9 demo
Warning: entry META-INF/versions/9/demo/SampleClass.class contains a class that
is identical to an entry already in the jar
```

The `--release 9` option tells `jar` to include everything that follows (the `demo` package inside the `sampleproject-9` directory) inside a versioned entry in the MRJAR, namely under `root/META-INF/versions/9`. The result is the following contents:

```
jar root
  - demo
     - SampleClass.class
  - META-INF
     - versions
        - 9
           - demo
              - SampleClass.class
```

Let us now create a class called Main that prints the URL of the `SampleClass`, and add it for the Java 9 version:

```
package demo;

import java.net.URL;

public class Main {

    public static void main(String[] args) throws Exception {
        URL url = Main.class.getClassLoader().getResource("demo/SampleClass.class");
        System.out.println(url);
    }
}
```

If we compile this class and re-run the jar command, we get an error:

```
C:\Users\manouti>jar --create --file MR.jar -C sampleproject-base demo --release 9 -C
sampleproject-9 demoentry: META-INF/versions/9/demo/Main.class, contains a new public class not
found in base entries
Warning: entry META-INF/versions/9/demo/Main.java, multiple resources with same name
Warning: entry META-INF/versions/9/demo/SampleClass.class contains a class that
is identical to an entry already in the jar
invalid multi-release jar file MR.jar deleted
```

The reason is that the `jar` tool prevents adding public classes to versioned entries if they are not added to the base entries as well. This is done so that the MRJAR exposes the same public API for the different Java versions. Note that at runtime, this rule is not required. It may be only applied by tools like `jar`. In this particular case, the purpose of `Main` is to run sample code, so we can simply add a copy in the base entry. If the class were part of a newer implementation that we only need for Java 9, it could be made non-public.

To add `Main` to the root entry, we first need to compile it to target a pre-Java 9 release. This can be done using the new `--release` option of `javac`:

```
C:\Users\manouti\sampleproject-base\demo>javac --release 8 Main.java
C:\Users\manouti\sampleproject-base\demo>cd ../..
C:\Users\manouti>jar --create --file MR.jar -C sampleproject-base demo --release 9 -C
sampleproject-9 demo
```

Running the Main class shows that the SampleClass gets loaded from the versioned directory:

```
C:\Users\manouti>java --class-path MR.jar demo.Main
jar:file:/C:/Users/manouti/MR.jar!/META-INF/versions/9/demo/SampleClass.class
```

Section 168.3: URL of a loaded class inside a multi-release Jar

Given the following multi-release Jar:

```
jar root
  - demo
     - SampleClass.class
  - META-INF
     - versions
        - 9
           - demo
              - SampleClass.class
```

The following class prints the URL of the `SampleClass`:

```
package demo;
```

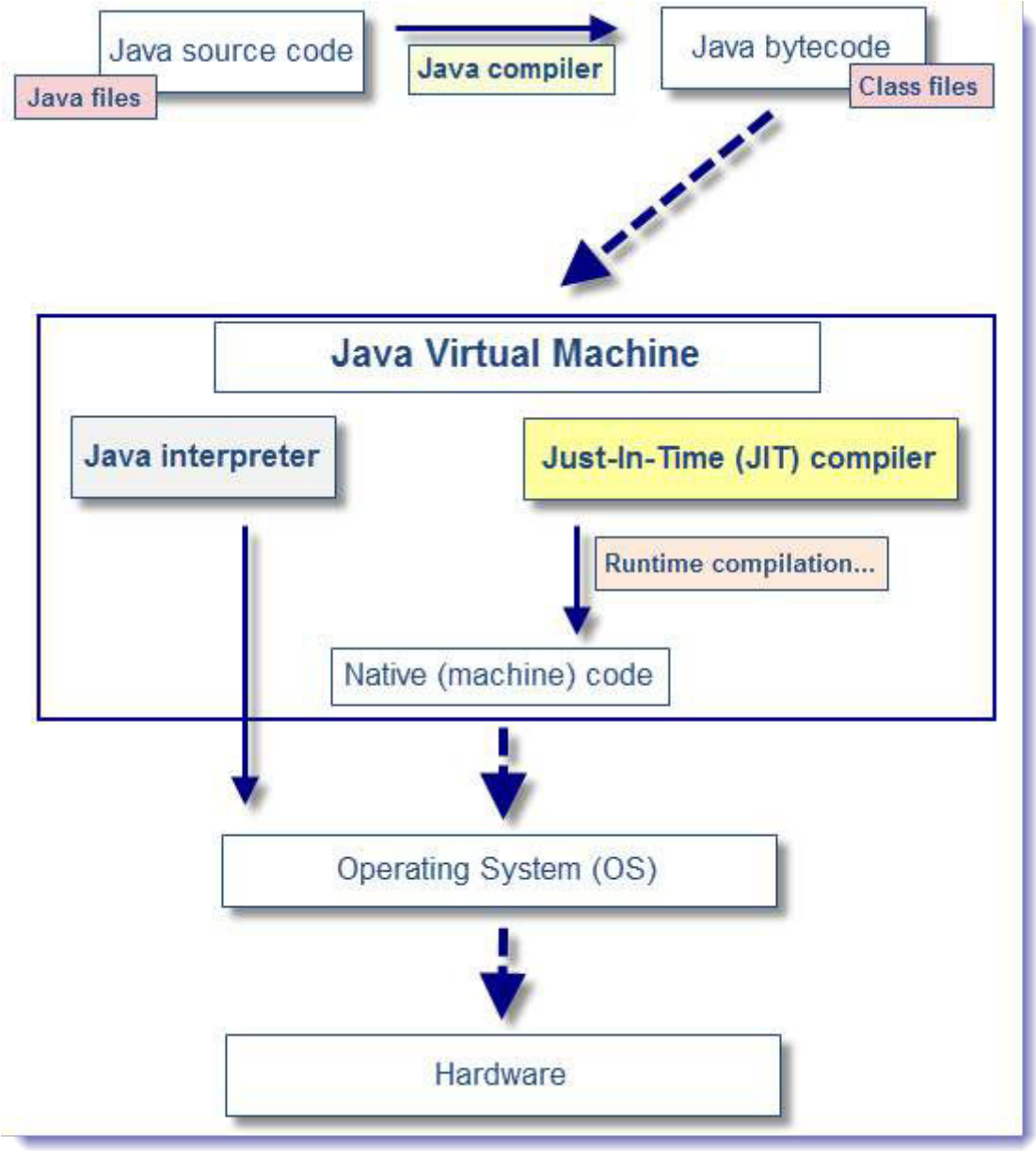
```
import java.net.URL;

public class Main {

    public static void main(String[] args) throws Exception {
        URL url = Main.class.getClassLoader().getResource("demo/SampleClass.class");
        System.out.println(url);
    }
}
```

If the class is compiled and added on the versioned entry for Java 9 in the MRJAR, running it would result in:

```
C:\Users\manouti>java --class-path MR.jar demo.Main
jar:file:/C:/Users/manouti/MR.jar!/META-INF/versions/9/demo/SampleClass.class
```

Chapter 169: Just in Time (JIT) compiler

Section 169.1: Overview



The Just-In-Time (JIT) compiler is a component of the Java™ Runtime Environment that improves the performance of Java applications at run time.

- Java programs consists of classes, which contain platform-neutral bytecodes that can be interpreted by a JVM on many different computer architectures.
- At run time, the JVM loads the class files, determines the semantics of each individual bytecode, and performs the appropriate computation.

> The additional processor and memory usage during interpretation means that a Java application performs more slowly than a native application.

> The JIT compiler helps improve the performance of Java programs by compiling bytecodes into native machine code at run time.

The JIT compiler is enabled by default, and is activated when a Java method is called. The JIT compiler compiles the bytecodes of that method into native machine code, compiling it `"just in time"` to run.

When a method has been compiled, the JVM calls the compiled code of that method directly instead of interpreting it. Theoretically, if compilation did not require processor time and memory usage, compiling every method could allow the speed of the Java program to approach that of a native application.

JIT compilation does require processor time and memory usage. When the JVM first starts up, thousands of methods are called. Compiling all of these methods can significantly affect startup time, even if the program eventually achieves very good peak performance.

- In practice, methods are not compiled the first time they are called. For each method, the JVM maintains a `call count` which is incremented every time the method is called.
- The JVM interprets a method until its call count exceeds a JIT compilation threshold.
- Therefore, often-used methods are compiled soon after the JVM has started, and less-used methods are compiled much later, or not at all.
- The JIT compilation threshold helps the JVM start quickly and still have improved performance.
- The threshold has been carefully selected to obtain an optimal balance between startup times and long term performance.
- After a method is compiled, its call count is reset to zero and subsequent calls to the method continue to increment its count.
- When the call count of a method reaches a JIT recompilation threshold, the JIT compiler compiles it a second time, applying a larger selection of optimizations than on the previous compilation.
- This process is repeated until the maximum optimization level is reached.

> The busiest methods of a Java program are always optimized most aggressively, maximizing the performance benefits of using the JIT compiler.

The JIT compiler can also measure `operational data at run time`, and use that data to improve the quality of further recompilations.

> The JIT compiler can be disabled, in which case the entire Java program will be interpreted. Disabling the JIT compiler is not recommended except to diagnose or work around JIT compilation problems.

Chapter 170: Bytecode Modification

Section 170.1: What is Bytecode?

Bytecode is the set of instructions used by the JVM. To illustrate this let's take this Hello World program.

```
public static void main(String[] args){
    System.out.println("Hello World");
}
```

This is what it turns into when compiled into bytecode.

```
public static main([Ljava/lang/String; args)V
    getstatic java/lang/System out Ljava/io/PrintStream;
    ldc "Hello World"
    invokevirtual java/io/PrintStream print(Ljava/lang/String;)V
```

What's the logic behind this?

getstatic - Retrieves the value of a static field of a class. In this case, the *PrintStream "Out"* of *System*.

ldc - Push a constant onto the stack. In this case, the String "Hello World"

invokevirtual - Invokes a method on a loaded reference on the stack and puts the result on the stack. Parameters of the method are also taken from the stack.

Well, there has to be more right?

There are 255 opcodes, but not all of them are implemented yet. A table with all of the current opcodes can be found here: Java bytecode instruction listings.

How can I write / edit bytecode?

There's multiple ways to write and edit bytecode. You can use a compiler, use a library, or use a program.

For writing:

- Jasmin
- Krakatau

For editing:

- Libraries
  - ASM
  - Javassist
  - BCEL - *Doesn't support Java 8+*
- Tools
  - Bytecode-Viewer
  - JBytedit
  - reJ - *Doesn't support Java 8+*
  - JBE - *Doesn't support Java 8+*

I'd like to learn more about bytecode!

There's probably a specific documentation page specificially for bytecode. This page focuses on the modification of

bytecode using different libraries and tools.

Section 170.2: How to edit jar files with ASM

Firstly the classes from the jar need to be loaded. We'll use three methods for this process:

- loadClasses(File)
- readJar(JarFile, JarEntry, Map)
- getNode(byte[])

```
Map<String, ClassNode> loadClasses(File jarFile) throws IOException {
    Map<String, ClassNode> classes = new HashMap<String, ClassNode>();
    JarFile jar = new JarFile(jarFile);
    Stream<JarEntry> str = jar.stream();
    str.forEach(z -> readJar(jar, z, classes));
    jar.close();
    return classes;
}

Map<String, ClassNode> readJar(JarFile jar, JarEntry entry, Map<String, ClassNode> classes) {
    String name = entry.getName();
    try (InputStream jis = jar.getInputStream(entry)){
        if (name.endsWith(".class")) {
            byte[] bytes = IOUtils.toByteArray(jis);
            String cafebabe = String.format("%02X%02X%02X%02X", bytes[0], bytes[1], bytes[2],
bytes[3]);
            if (!cafebabe.toLowerCase().equals("cafebabe")) {
                // This class doesn't have a valid magic
                return classes;
            }
            try {
                ClassNode cn = getNode(bytes);
                classes.put(cn.name, cn);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return classes;
}

ClassNode getNode(byte[] bytes) {
    ClassReader cr = new ClassReader(bytes);
    ClassNode cn = new ClassNode();
    try {
        cr.accept(cn, ClassReader.EXPAND_FRAMES);
    } catch (Exception e) {
        e.printStackTrace();
    }
    cr = null;
    return cn;
}
```

With these methods loading and changing a jar file becomes a simple matter of changing ClassNodes in a map. In this example we will replace all Strings in the jar with capitalized ones using the Tree API.

```
File jarFile = new File("sample.jar");
Map<String, ClassNode> nodes = loadClasses(jarFile);
```

```
// Iterate ClassNodes
for (ClassNode cn : nodes.values()){
    // Iterate methods in class
    for (MethodNode mn : cn.methods){
        // Iterate instructions in method
        for (AbstractInsnNode ain : mn.instructions.toArray()){
            // If the instruction is loading a constant value
            if (ain.getOpcode() == Opcodes.LDC){
                // Cast current instruction to Ldc
                // If the constant is a string then capitalize it.
                LdcInsnNode ldc = (LdcInsnNode) ain;
                if (ldc.cst instanceof String){
                    ldc.cst = ldc.cst.toString().toUpperCase();
                }
            }
        }
    }
}
```

Now that all of the ClassNode's strings have been modified we need to save the changes. In order to save the changes and have a working output a few things have to be done:

- Export ClassNodes to bytes
- Load non-class jar entries *(Ex: Manifest.mf / other binary resources in jar)* as bytes
- Save all bytes to a new jar

From the last portion above, we'll create three methods.

- processNodes(Map<String, ClassNode> nodes)
- loadNonClasses(File jarFile)
- saveAsJar(Map<String, byte[]> outBytes, String fileName)

Usage:

```
Map<String, byte[]> out = process(nodes, new HashMap<String, MappedClass>());
out.putAll(loadNonClassEntries(jarFile));
saveAsJar(out, "sample-edit.jar");
```

The methods used:

```
static Map<String, byte[]> processNodes(Map<String, ClassNode> nodes, Map<String, MappedClass>
mappings) {
    Map<String, byte[]> out = new HashMap<String, byte[]>();
    // Iterate nodes and add them to the map of <Class names , Class bytes>
    // Using Compute_Frames ensures that stack-frames will be re-calculated automatically
    for (ClassNode cn : nodes.values()) {
        ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
        out.put(mappings.containsKey(cn.name) ? mappings.get(cn.name).getNewName() : cn.name,
cw.toByteArray());
    }
    return out;
}

static Map<String, byte[]> loadNonClasses(File jarFile) throws IOException {
    Map<String, byte[]> entries = new HashMap<String, byte[]>();
    ZipInputStream jis = new ZipInputStream(new FileInputStream(jarFile));
    ZipEntry entry;
    // Iterate all entries
    while ((entry = jis.getNextEntry()) != null) {
```

```
        try {
            String name = entry.getName();
            if (!name.endsWith(".class") && !entry.isDirectory()) {
                // Apache Commons - byte[] toByteArray(InputStream input)
                //
                // Add each entry to the map <Entry name , Entry bytes>
                byte[] bytes = IOUtils.toByteArray(jis);
                entries.put(name, bytes);
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            jis.closeEntry();
        }
    }
    jis.close();
    return entries;
}

static void saveAsJar(Map<String, byte[]> outBytes, String fileName) {
    try {
        // Create jar output stream
        JarOutputStream out = new JarOutputStream(new FileOutputStream(fileName));
        // For each entry in the map, save the bytes
        for (String entry : outBytes.keySet()) {
            // Appent class names to class entries
            String ext = entry.contains(".") ? "" : ".class";
            out.putNextEntry(new ZipEntry(entry + ext));
            out.write(outBytes.get(entry));
            out.closeEntry();
        }
        out.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

That's it. All the changes will be saved to "sample-edit.jar".

Section 170.3: How to load a ClassNode as a Class

```
/**
 * Load a class by from a ClassNode
 *
 * @param cn
 *            ClassNode to load
 * @return
 */
public static Class<?> load(ClassNode cn) {
    ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
    return new ClassDefiner(ClassLoader.getSystemClassLoader()).get(cn.name.replace("/", "."),
cw.toByteArray());
}

/**
 * Classloader that loads a class from bytes.
 */
static class ClassDefiner extends ClassLoader {
    public ClassDefiner(ClassLoader parent) {
        super(parent);
    }
```

```
    public Class<?> get(String name, byte[] bytes) {
        Class<?> c = defineClass(name, bytes, 0, bytes.length);
        resolveClass(c);
        return c;
    }
}
```

Section 170.4: How to rename classes in a jar file

```
public static void main(String[] args) throws Exception {
    File jarFile = new File("Input.jar");
    Map<String, ClassNode> nodes = JarUtils.loadClasses(jarFile);

    Map<String, byte[]> out = JarUtils.loadNonClassEntries(jarFile);
    Map<String, String> mappings = new HashMap<String, String>();
    mappings.put("me/example/ExampleClass", "me/example/ExampleRenamed");
    out.putAll(process(nodes, mappings));
    JarUtils.saveAsJar(out, "Input-new.jar");
}

static Map<String, byte[]> process(Map<String, ClassNode> nodes, Map<String, String> mappings) {
    Map<String, byte[]> out = new HashMap<String, byte[]>();
    Remapper mapper = new SimpleRemapper(mappings);
    for (ClassNode cn : nodes.values()) {
        ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
        ClassVisitor remapper = new ClassRemapper(cw, mapper);
        cn.accept(remapper);
        out.put(mappings.containsKey(cn.name) ? mappings.get(cn.name) : cn.name, cw.toByteArray());
    }
    return out;
}
```

SimpleRemapper is an existing class in the ASM library. However it only allows for class names to be changed. If you wish to rename fields and methods you should create your own implemenation of the Remapper class.

Section 170.5: Javassist Basic

Javassist is a bytecode instrumentation library that allows you to modify bytecode injecting Java code that will be converted to bytecode by Javassist and added to the instrumented class/method at runtime.

Lets write the first transformer that actually take an hypothetical class "com.my.to.be.instrumented.MyClass" and add to the instructions of each method a log call.

```
import java.lang.instrument.ClassFileTransformer;
import java.lang.instrument.IllegalClassFormatException;
import java.security.ProtectionDomain;
import javassist.ClassPool;
import javassist.CtClass;
import javassist.CtMethod;

public class DynamicTransformer implements ClassFileTransformer {

    public byte[] transform(ClassLoader loader, String className, Class classBeingRedefined,
        ProtectionDomain protectionDomain, byte[] classfileBuffer) throws
IllegalClassFormatException {

        byte[] byteCode = classfileBuffer;

        // into the transformer will arrive every class loaded so we filter
```

```
        // to match only what we need
        if (className.equals("com/my/to/be/instrumented/MyClass")) {

            try {
                // retrive default Javassist class pool
                ClassPool cp = ClassPool.getDefault();
                // get from the class pool our class with this qualified name
                CtClass cc = cp.get("com.my.to.be.instrumented.MyClass");
                // get all the methods of the retrieved class
                CtMethod[] methods = cc.getDeclaredMethods()
                for(CtMethod meth : methods) {
                    // The instrumentation code to be returned and injected
                    final StringBuffer buffer = new StringBuffer();
                    String name = meth.getName();
                    // just print into the buffer a log for example
                    buffer.append("System.out.println(\"Method " + name + " executed\" );");
                    meth.insertBefore(buffer.toString())
                }
                // create the byteclode of the class
                byteCode = cc.toBytecode();
                // remove the CtClass from the ClassPool
                cc.detach();
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }

        return byteCode;
    }
}
```

Now in order to use this transformer (so that our JVM will call the method transform on each class at load time) we need to add this instrumentor this with an agent:

```
import java.lang.instrument.Instrumentation;

public class EasyAgent {

    public static void premain(String agentArgs, Instrumentation inst) {

        // registers the transformer
        inst.addTransformer(new DynamicTransformer());
    }
}
```

Last step to start our first instrumentor experiment is to actually register this agent class to the JVM machine execution. The easiest way to actually do it is to register it with an option into the shell command:

```
java -javaagent:myAgent.jar MyJavaApplication
```

As we can see the agent/transformer project is added as a jar to the execution of any application named MyJavaApplication that is supposed to contain a class named "com.my.to.be.instrumented.MyClass" to actually execute our injected code.

Chapter 171: Disassembling and Decompiling

Name	Description
<classes>	List of classes to disassemble. Can be in either `package1.package2.Classname` format, *or* `package1/package2/Classname` format. Do **not** include the **`.class`** extension.
`-help`, `--help`, `-?`	Print this usage message
`-version`	Version information
`-v`, `-verbose`	Print additional information
`-l`	Print line number and local variable tables
`-public`	Show only public classes and members
`-protected`	Show protected/public classes and members
`-package`	Show package/protected/public classes and members (default)
`-p`, **`-private`**	Show all classes and members
`-c`	Disassemble the code
`-s`	Print internal type signatures
`-sysinfo`	Show system info (path, size, date, MD5 hash) of class being processed
`-constants`	Show final constants
`-classpath <path>`	Specify where to find user class files
`-cp <path>`	Specify where to find user class files
`-bootclasspath <path>`	Override location of bootstrap class files

Section 171.1: Viewing bytecode with javap

If you want to see the generated bytecode for a Java program, you can use the provided `javap` command to view it.

Assuming that we have the following Java source file:

```
package com.stackoverflow.documentation;

import org.springframework.stereotype.Service;

import java.io.IOException;
import java.io.InputStream;
import java.util.List;

@Service
public class HelloWorldService {

    public void sayHello() {
        System.out.println("Hello, World!");
    }

    private Object[] pvtMethod(List<String> strings) {
        return new Object[]{strings};
    }

    protected String tryCatchResources(String filename) throws IOException {
        try (InputStream inputStream = getClass().getResourceAsStream(filename)) {
            byte[] bytes = new byte[8192];
            int read = inputStream.read(bytes);
            return new String(bytes, 0, read);
        } catch (IOException | RuntimeException e) {
```

```
            e.printStackTrace();
            throw e;
        }
    }

    void stuff() {
        System.out.println("stuff");
    }
}
```

After compiling the source file, the most simple usage is:

```
cd <directory containing classes> (e.g. target/classes)
javap com/stackoverflow/documentation/SpringExample
```

Which produces the output

```
Compiled from "HelloWorldService.java"
public class com.stackoverflow.documentation.HelloWorldService {
public com.stackoverflow.documentation.HelloWorldService();
public void sayHello();
protected java.lang.String tryCatchResources(java.lang.String) throws java.io.IOException;
void stuff();
}
```

This lists all non-private methods in the class, but that is not particularly useful for most purposes. The following command is a lot more useful:

```
javap -p -c -s -constants -l -v com/stackoverflow/documentation/HelloWorldService
```

Which produces the output:

```
Classfile /Users/pivotal/IdeaProjects/stackoverflow-spring-
docs/target/classes/com/stackoverflow/documentation/HelloWorldService.class
Last modified Jul 22, 2016; size 2167 bytes
MD5 checksum 6e33b5c292ead21701906353b7f06330
Compiled from "HelloWorldService.java"
public class com.stackoverflow.documentation.HelloWorldService
minor version: 0
major version: 51
flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
#1 = Methodref          #5.#60         // java/lang/Object."":()V
#2 = Fieldref           #61.#62        // java/lang/System.out:Ljava/io/PrintStream;
#3 = String             #63            // Hello, World!
#4 = Methodref          #64.#65        // java/io/PrintStream.println:(Ljava/lang/String;)V
#5 = Class              #66            // java/lang/Object
#6 = Methodref          #5.#67         // java/lang/Object.getClass:()Ljava/lang/Class;
#7 = Methodref          #68.#69        //
java/lang/Class.getResourceAsStream:(Ljava/lang/String;)Ljava/io/InputStream;
#8 = Methodref          #70.#71        // java/io/InputStream.read:([B)I
#9 = Class              #72            // java/lang/String
#10 = Methodref          #9.#73         // java/lang/String."":([BII)V
#11 = Methodref          #70.#74        // java/io/InputStream.close:()V
#12 = Class              #75            // java/lang/Throwable
#13 = Methodref          #12.#76        //
java/lang/Throwable.addSuppressed:(Ljava/lang/Throwable;)V
#14 = Class              #77            // java/io/IOException
#15 = Class              #78            // java/lang/RuntimeException
```

```
#16 = Methodref          #79.#80       // java/lang/Exception.printStackTrace:()V
#17 = String             #55           // stuff
#18 = Class              #81           // com/stackoverflow/documentation/HelloWorldService
#19 = Utf8
#20 = Utf8               ()V
#21 = Utf8               Code
#22 = Utf8               LineNumberTable
#23 = Utf8               LocalVariableTable
#24 = Utf8               this
#25 = Utf8               Lcom/stackoverflow/documentation/HelloWorldService;
#26 = Utf8               sayHello
#27 = Utf8               pvtMethod
#28 = Utf8               (Ljava/util/List;)[Ljava/lang/Object;
#29 = Utf8               strings
#30 = Utf8               Ljava/util/List;
#31 = Utf8               LocalVariableTypeTable
#32 = Utf8               Ljava/util/List;
#33 = Utf8               Signature
#34 = Utf8               (Ljava/util/List;)[Ljava/lang/Object;
#35 = Utf8               tryCatchResources
#36 = Utf8               (Ljava/lang/String;)Ljava/lang/String;
#37 = Utf8               bytes
#38 = Utf8               [B
#39 = Utf8               read
#40 = Utf8               I
#41 = Utf8               inputStream
#42 = Utf8               Ljava/io/InputStream;
#43 = Utf8               e
#44 = Utf8               Ljava/lang/Exception;
#45 = Utf8               filename
#46 = Utf8               Ljava/lang/String;
#47 = Utf8               StackMapTable
#48 = Class              #81           // com/stackoverflow/documentation/HelloWorldService
#49 = Class              #72           // java/lang/String
#50 = Class              #82           // java/io/InputStream
#51 = Class              #75           // java/lang/Throwable
#52 = Class              #38           // "[B"
#53 = Class              #83           // java/lang/Exception
#54 = Utf8               Exceptions
#55 = Utf8               stuff
#56 = Utf8               SourceFile
#57 = Utf8               HelloWorldService.java
#58 = Utf8               RuntimeVisibleAnnotations
#59 = Utf8               Lorg/springframework/stereotype/Service;
#60 = NameAndType        #19:#20       // "":()V
#61 = Class              #84           // java/lang/System
#62 = NameAndType        #85:#86       // out:Ljava/io/PrintStream;
#63 = Utf8               Hello, World!
#64 = Class              #87           // java/io/PrintStream
#65 = NameAndType        #88:#89       // println:(Ljava/lang/String;)V
#66 = Utf8               java/lang/Object
#67 = NameAndType        #90:#91       // getClass:()Ljava/lang/Class;
#68 = Class              #92           // java/lang/Class
#69 = NameAndType        #93:#94       //
getResourceAsStream:(Ljava/lang/String;)Ljava/io/InputStream;
#70 = Class              #82           // java/io/InputStream
#71 = NameAndType        #39:#95       // read:([B)I
#72 = Utf8               java/lang/String
#73 = NameAndType        #19:#96       // "":([BII)V
#74 = NameAndType        #97:#20       // close:()V
#75 = Utf8               java/lang/Throwable
#76 = NameAndType        #98:#99       // addSuppressed:(Ljava/lang/Throwable;)V
#77 = Utf8               java/io/IOException
#78 = Utf8               java/lang/RuntimeException
```

```
#79 = Class              #83           // java/lang/Exception
#80 = NameAndType        #100:#20      // printStackTrace:()V
#81 = Utf8               com/stackoverflow/documentation/HelloWorldService
#82 = Utf8               java/io/InputStream
#83 = Utf8               java/lang/Exception
#84 = Utf8               java/lang/System
#85 = Utf8               out
#86 = Utf8               Ljava/io/PrintStream;
#87 = Utf8               java/io/PrintStream
#88 = Utf8               println
#89 = Utf8               (Ljava/lang/String;)V
#90 = Utf8               getClass
#91 = Utf8               ()Ljava/lang/Class;
#92 = Utf8               java/lang/Class
#93 = Utf8               getResourceAsStream
#94 = Utf8               (Ljava/lang/String;)Ljava/io/InputStream;
#95 = Utf8               ([B)I
#96 = Utf8               ([BII)V
#97 = Utf8               close
#98 = Utf8               addSuppressed
#99 = Utf8               (Ljava/lang/Throwable;)V
#100 = Utf8               printStackTrace
{
public com.stackoverflow.documentation.HelloWorldService();
descriptor: ()V
flags: ACC_PUBLIC
Code:
stack=1, locals=1, args_size=1
0: aload_0
1: invokespecial #1                  // Method java/lang/Object."":()V
4: return
LineNumberTable:
line 10: 0
LocalVariableTable:
Start  Length  Slot  Name   Signature
0       5     0  this   Lcom/stackoverflow/documentation/HelloWorldService;

public void sayHello();
descriptor: ()V
flags: ACC_PUBLIC
Code:
stack=2, locals=1, args_size=1
0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
3: ldc           #3                  // String Hello, World!
5: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
8: return
LineNumberTable:
line 13: 0
line 14: 8
LocalVariableTable:
Start  Length  Slot  Name   Signature
0       9     0  this   Lcom/stackoverflow/documentation/HelloWorldService;

private java.lang.Object[] pvtMethod(java.util.List);
descriptor: (Ljava/util/List;)[Ljava/lang/Object;
flags: ACC_PRIVATE
Code:
stack=4, locals=2, args_size=2
0: iconst_1
1: anewarray     #5                  // class java/lang/Object
4: dup
5: iconst_0
6: aload_1
7: aastore
```

```
8: areturn
LineNumberTable:
line 17: 0
LocalVariableTable:
Start  Length  Slot  Name   Signature
0       9     0  this   Lcom/stackoverflow/documentation/HelloWorldService;
0       9     1 strings   Ljava/util/List;
LocalVariableTypeTable:
Start  Length  Slot  Name   Signature
0       9     1 strings   Ljava/util/List;
Signature: #34                          // (Ljava/util/List;)[Ljava/lang/Object;

protected java.lang.String tryCatchResources(java.lang.String) throws java.io.IOException;
descriptor: (Ljava/lang/String;)Ljava/lang/String;
flags: ACC_PROTECTED
Code:
stack=5, locals=10, args_size=2
0: aload_0
1: invokevirtual #6                  // Method java/lang/Object.getClass:()Ljava/lang/Class;
4: aload_1
5: invokevirtual #7                  // Method
java/lang/Class.getResourceAsStream:(Ljava/lang/String;)Ljava/io/InputStream;
8: astore_2
9: aconst_null
10: astore_3
11: sipush        8192
14: newarray       byte
16: astore        4
18: aload_2
19: aload         4
21: invokevirtual #8                  // Method java/io/InputStream.read:([B)I
24: istore        5
26: new           #9                  // class java/lang/String
29: dup
30: aload         4
32: iconst_0
33: iload         5
35: invokespecial #10                 // Method java/lang/String."":([BII)V
38: astore        6
40: aload_2
41: ifnull        70
44: aload_3
45: ifnull        66
48: aload_2
49: invokevirtual #11                 // Method java/io/InputStream.close:()V
52: goto          70
55: astore        7
57: aload_3
58: aload         7
60: invokevirtual #13                 // Method
java/lang/Throwable.addSuppressed:(Ljava/lang/Throwable;)V
63: goto          70
66: aload_2
67: invokevirtual #11                 // Method java/io/InputStream.close:()V
70: aload         6
72: areturn
73: astore        4
75: aload         4
77: astore_3
78: aload         4
80: athrow
81: astore        8
83: aload_2
84: ifnull        113
```

```
87: aload_3
88: ifnull        109
91: aload_2
92: invokevirtual #11                 // Method java/io/InputStream.close:()V
95: goto          113
98: astore        9
100: aload_3
101: aload         9
103: invokevirtual #13                 // Method
java/lang/Throwable.addSuppressed:(Ljava/lang/Throwable;)V
106: goto          113
109: aload_2
110: invokevirtual #11                 // Method java/io/InputStream.close:()V
113: aload         8
115: athrow
116: astore_2
117: aload_2
118: invokevirtual #16                 // Method java/lang/Exception.printStackTrace:()V
121: aload_2
122: athrow
Exception table:
from    to  target type
48    52    55   Class java/lang/Throwable
11    40    73   Class java/lang/Throwable
11    40    81   any
91    95    98   Class java/lang/Throwable
73    83    81   any
0    70   116   Class java/io/IOException
0    70   116   Class java/lang/RuntimeException
73   116   116   Class java/io/IOException
73   116   116   Class java/lang/RuntimeException
LineNumberTable:
line 21: 0
line 22: 11
line 23: 18
line 24: 26
line 25: 40
line 21: 73
line 25: 81
line 26: 117
line 27: 121
LocalVariableTable:
Start  Length  Slot  Name   Signature
18      55     4 bytes   [B
26      47     5  read   I
9     107     2 inputStream   Ljava/io/InputStream;
117       6     2     e   Ljava/lang/Exception;
0     123     0  this   Lcom/stackoverflow/documentation/HelloWorldService;
0     123     1 filename   Ljava/lang/String;
StackMapTable: number_of_entries = 9
frame_type = 255 /* full_frame */
offset_delta = 55
locals = [ class com/stackoverflow/documentation/HelloWorldService, class java/lang/String, class
java/io/InputStream, class java/lang/Throwable, class "[B", int, class java/lang/String ]
stack = [ class java/lang/Throwable ]
frame_type = 10 /* same */
frame_type = 3 /* same */
frame_type = 255 /* full_frame */
offset_delta = 2
locals = [ class com/stackoverflow/documentation/HelloWorldService, class java/lang/String, class
java/io/InputStream, class java/lang/Throwable ]
stack = [ class java/lang/Throwable ]
frame_type = 71 /* same_locals_1_stack_item */
stack = [ class java/lang/Throwable ]
```

```
frame_type = 255 /* full_frame */
offset_delta = 16
locals = [ class com/stackoverflow/documentation/HelloWorldService, class java/lang/String, class
java/io/InputStream, class java/lang/Throwable, top, top, top, top, class java/lang/Throwable ]
stack = [ class java/lang/Throwable ]
frame_type = 10 /* same */
frame_type = 3 /* same */
frame_type = 255 /* full_frame */
offset_delta = 2
locals = [ class com/stackoverflow/documentation/HelloWorldService, class java/lang/String ]
stack = [ class java/lang/Exception ]
Exceptions:
throws java.io.IOException

void stuff();
descriptor: ()V
flags:
Code:
stack=2, locals=1, args_size=1
0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
3: ldc           #17                 // String stuff
5: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
8: return
LineNumberTable:
line 32: 0
line 33: 8
LocalVariableTable:
Start  Length  Slot  Name   Signature
0       9     0  this   Lcom/stackoverflow/documentation/HelloWorldService;
}
SourceFile: "HelloWorldService.java"
RuntimeVisibleAnnotations:
0: #59()
```

Chapter 172: JMX

The JMX technology provides the tools for building distributed, Web-based, modular and dynamic solutions for managing and monitoring devices, applications, and service-driven networks. By design, this standard is suitable for adapting legacy systems, implementing new management and monitoring solutions, and plugging into those of the future.

Section 172.1: Simple example with Platform MBean Server

Let's say we have some server that registers new users and greets them with some message. And we want to monitor this server and change some of it's parameters.

First, we need an interface with our monitoring and control methods

```
public interface UserCounterMBean {
    long getSleepTime();

    void setSleepTime(long sleepTime);

    int getUserCount();

    void setUserCount(int userCount);

    String getGreetingString();

    void setGreetingString(String greetingString);

    void stop();
}
```

And some simple implementation that will let us see how it's working and how we affect it

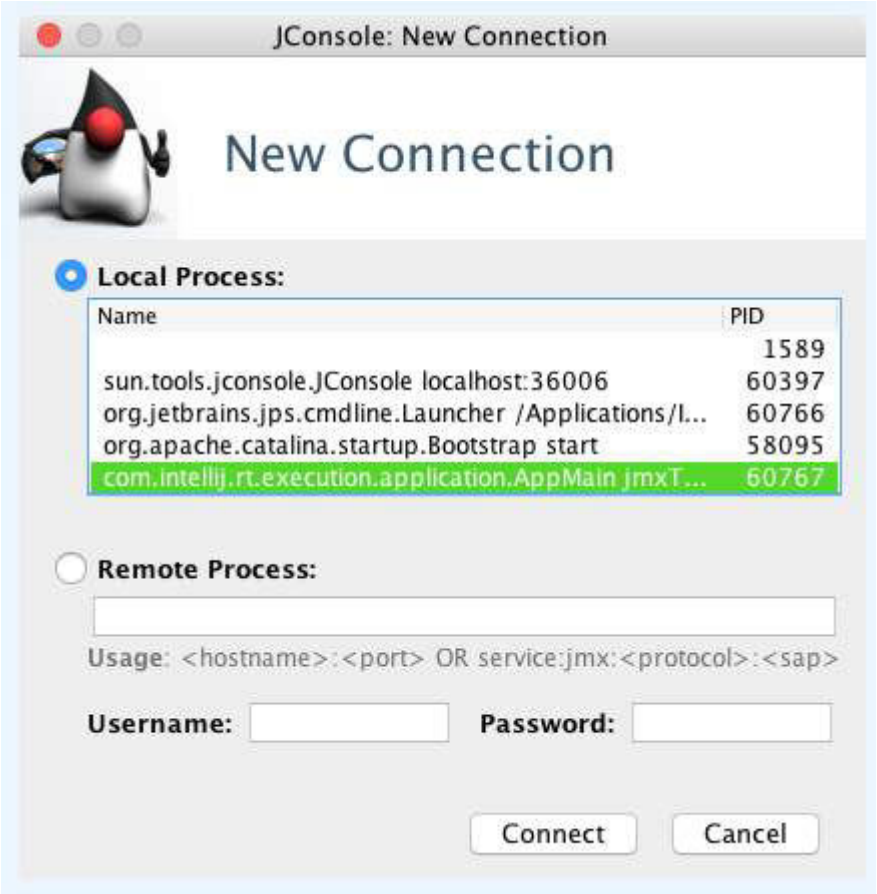
```
public class UserCounter implements UserCounterMBean, Runnable {
    private AtomicLong sleepTime = new AtomicLong(10000);
    private AtomicInteger userCount = new AtomicInteger(0);
    private AtomicReference<String> greetingString = new AtomicReference<>("welcome");
    private AtomicBoolean interrupted = new AtomicBoolean(false);

    @Override
    public long getSleepTime() {
        return sleepTime.get();
    }

    @Override
    public void setSleepTime(long sleepTime) {
        this.sleepTime.set(sleepTime);
    }

    @Override
    public int getUserCount() {
        return userCount.get();
    }

    @Override
    public void setUserCount(int userCount) {
        this.userCount.set(userCount);
    }
```

```
    @Override
    public String getGreetingString() {
        return greetingString.get();
    }

    @Override
    public void setGreetingString(String greetingString) {
        this.greetingString.set(greetingString);
    }

    @Override
    public void stop() {
        this.interrupted.set(true);
    }

    @Override
    public void run() {
        while (!interrupted.get()) {
            try {
                System.out.printf("User %d, %s%n", userCount.incrementAndGet(),
greetingString.get());
                Thread.sleep(sleepTime.get());
            } catch (InterruptedException ignored) {
            }
        }
    }
}
```

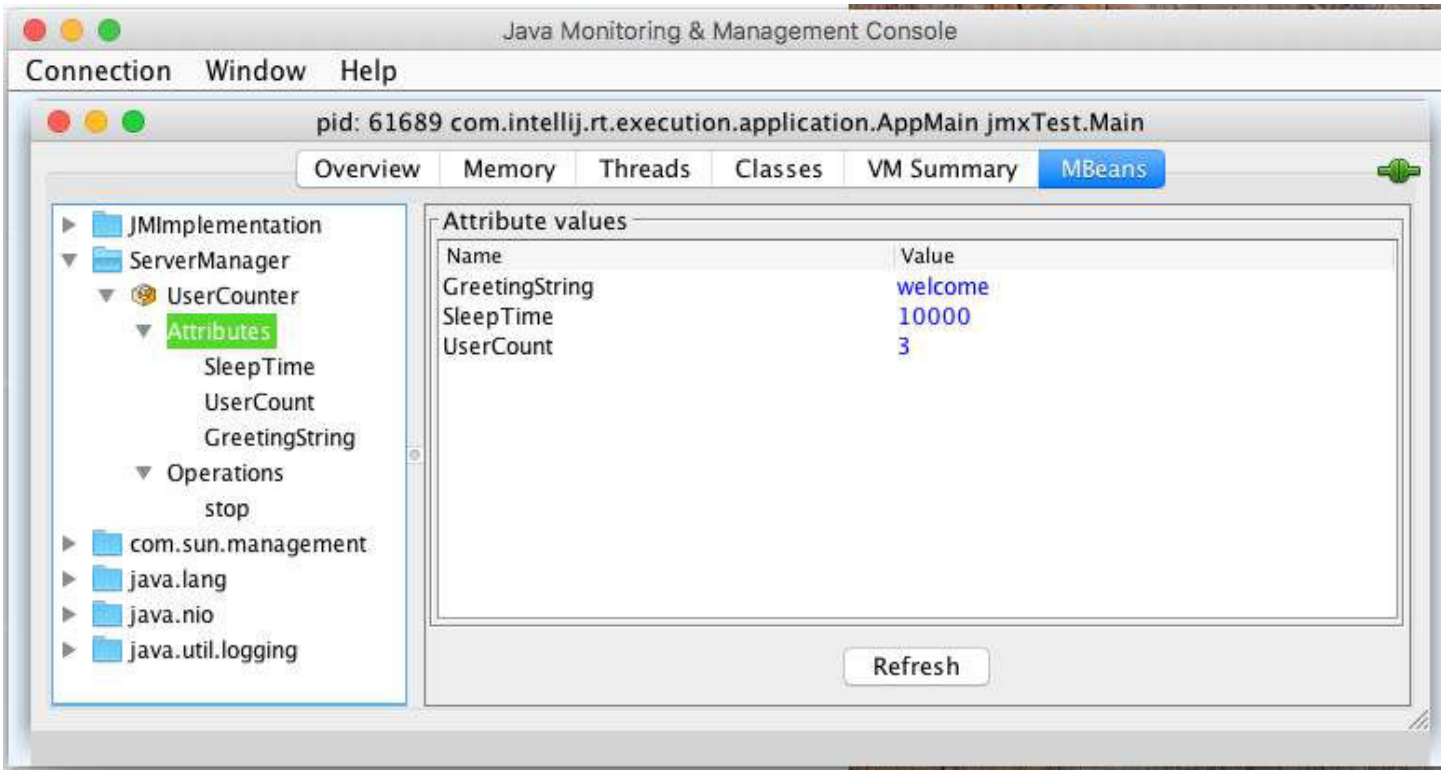
For simple example with local or remote management, we need to register our MBean:

```
import javax.management.InstanceAlreadyExistsException;
import javax.management.MBeanRegistrationException;
import javax.management.MBeanServer;
import javax.management.MalformedObjectNameException;
import javax.management.NotCompliantMBeanException;
import javax.management.ObjectName;
import java.lang.management.ManagementFactory;

public class Main {
    public static void main(String[] args) throws MalformedObjectNameException,
NotCompliantMBeanException, InstanceAlreadyExistsException, MBeanRegistrationException,
InterruptedException {
        final UserCounter userCounter = new UserCounter();
        final MBeanServer mBeanServer = ManagementFactory.getPlatformMBeanServer();
        final ObjectName objectName = new ObjectName("ServerManager:type=UserCounter");
        mBeanServer.registerMBean(userCounter, objectName);

        final Thread thread = new Thread(userCounter);
        thread.start();
        thread.join();
    }
}
```

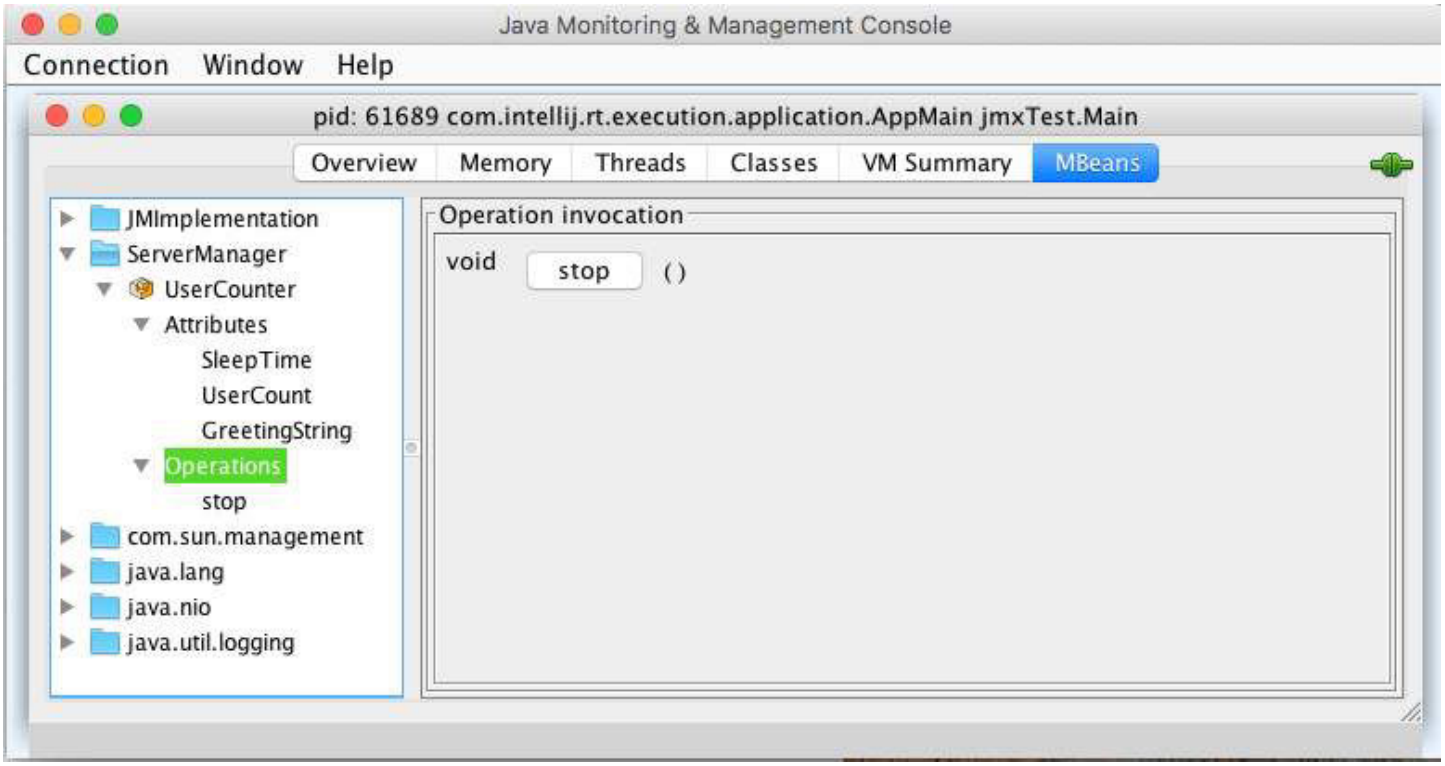
After that we can run our application and connect to it via jConsole, which can be found in your `$JAVA_HOME/bin` directory. First, we need to find our local java process with our application



then switch to MBeans tab and find that MBean that we used in our Main class as an `ObjectName` (in the example above it's `ServerManager`). In `Attributes` section we can see out attributes. If you specified get method only, attribute will be readable but not writeable. If you specified both get and set methods, attribute would be readable and writeable.



Specified methods can be invoked in `Operations` section.



If you want to be able to use remote management, you will need additional JVM parameters, like:

```
-Dcom.sun.management.jmxremote=true //true by default
-Dcom.sun.management.jmxremote.port=36006
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
```

These parameters can be found in Chapter 2 of JMX guides. After that you will be able to connect to your application via jConsole remotely with `jconsole host:port` or with specifying `host:port` or `service:jmx:rmi:///jndi/rmi://hostName:portNum/jmxrmi` in jConsole GUI.

Useful links:

- JMX guides
- JMX Best practices

Chapter 173: Java Virtual Machine (JVM)

Section 173.1: These are the basics

JVM is an **abstract computing machine** or **Virtual machine** that resides in your RAM. It has a platform-independent execution environment that interprets Java bytecode into native machine code. (Javac is Java Compiler which compiles your Java code into Bytecode)

Java program will be running inside the JVM which is then mapped onto the underlying physical machine. It is one of programming tool in JDK.

(`Byte code` is platform-independent code which is run on every platform and `Machine code` is platform-specific code which is run in only specific platform such as windows or linux; it depend on execution.)

Some of the components:

- Class Loder - load the .class file into RAM.
- Bytecode verifier - check whether there are any access restriction violations in your code.
- Execution engine - convert the byte code into executable machine code.
- JIT(just in time) - JIT is part of JVM which used to improves the performance of JVM.It will dynamically compile or translate java bytecode into native machine code during execution time.

(Edited)

Chapter 174: XJC

Parameter	Details
schema file	The xsd schema file to convert to java

XJC is a Java SE tool that compiles an XML schema file into fully annotated Java classes.

It is distributed within the JDK package and is located at `/bin/xjc` path.

Section 174.1: Generating Java code from simple XSD file

XSD schema (schema.xsd)

The following xml schema (xsd) defines a list of users with attributes `name` and `reputation`.

```
<?xml version="1.0"?>

<xs:schema version="1.0"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:ns="http://www.stackoverflow.com/users"
           elementFormDefault="qualified"
           targetNamespace="http://www.stackoverflow.com/users">
    <xs:element name="users" type="ns:Users"/>

    <xs:complexType name="Users">
        <xs:sequence>
            <xs:element type="ns:User" name="user" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="User">
        <xs:attribute name="name" use="required" type="xs:string"/>
        <xs:attribute name="reputation" use="required">
            <xs:simpleType>
                <xs:restriction base="xs:int">
                    <xs:minInclusive value="1"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:attribute>
    </xs:complexType>
</xs:schema>
```

Using xjc

This requires the path to the xjc tool (JDK binaries) to be in the OS path variable.

The code generation can be started using

```
xjc schema.xsd
```

This will generate java files in the working directory.

Result files

There will be some additional comments, but basically the java files generated look like this:

```
package com.stackoverflow.users;
```

```java
import java.util.ArrayList;
import java.util.List;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlType;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "Users", propOrder = {
    "user"
})
public class Users {

    protected List<User> user;

    public List<User> getUser() {
        if (user == null) {
            user = new ArrayList<User>();
        }
        return this.user;
    }

}
```

```java
package com.stackoverflow.users;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlType;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "User")
public class User {

    @XmlAttribute(name = "name", required = true)
    protected String name;
    @XmlAttribute(name = "reputation", required = true)
    protected int reputation;

    public String getName() {
        return name;
    }

    public void setName(String value) {
        this.name = value;
    }

    public int getReputation() {
        return reputation;
    }

    public void setReputation(int value) {
        this.reputation = value;
    }

}
```

```java
package com.stackoverflow.users;

import javax.xml.bind.JAXBElement;
import javax.xml.bind.annotation.XmlElementDecl;
import javax.xml.bind.annotation.XmlRegistry;
import javax.xml.namespace.QName;
```

```java
@XmlRegistry
public class ObjectFactory {

    private final static QName _Users_QNAME = new QName("http://www.stackoverflow.com/users",
"users");

    public ObjectFactory() {
    }

    public Users createUsers() {
        return new Users();
    }

    public User createUser() {
        return new User();
    }

    @XmlElementDecl(namespace = "http://www.stackoverflow.com/users", name = "users")
    public JAXBElement<Users> createUsers(Users value) {
        return new JAXBElement<Users>(_Users_QNAME, Users.class, null, value);
    }

}
```

package-info.java

```java
@javax.xml.bind.annotation.XmlSchema(namespace = "http://www.stackoverflow.com/users",
elementFormDefault = javax.xml.bind.annotation.XmlNsForm.QUALIFIED)
package com.stackoverflow.users;
```

Chapter 175: JVM Flags

Section 175.1: -XXaggressive

`-XXaggressive` is a collection of configurations that make the JVM perform at a high speed and reach a stable state as soon as possible. To achieve this goal, the JVM uses more internal resources at startup; however, it requires less adaptive optimization once the goal is reached. We recommend that you use this option for long-running, memory-intensive applications that work alone.

Usage:

```
-XXaggressive:<param>
```

<param>	Description
`opt`	Schedules adaptive optimizations earlier and enables new optimizations, which are expected to be the default in future releases.
`memory`	Configures the memory system for memory-intensive workloads and sets an expectation to enable large amounts of memory resources to ensure high throughput. JRockit JVM will also use large pages, if available.

Section 175.2: -XXallocClearChunks

This option allows you to clear a TLA for references and values at TLA allocation time and pre-fetch the next chunk. When an integer, a reference, or anything else is declared, it has a default value of 0 or null (depending upon type). At the appropriate time, you will need to clear these references and values to free the memory on the heap so Java can use- or reuse- it. You can do either when the object is allocated or, by using this option, when you request a new TLA.

Usage:

```
-XXallocClearChunks
-XXallocClearChunks=<true | false>
```

The above is a boolean option and is generally recommended on IA64 systems; ultimately, its use depends upon the application. If you want to set the size of chunks cleared, combine this option with `-XXallocClearChunkSize`. If you use this flag but do not specify a boolean value, the default is **`true`**.

Section 175.3: -XXallocClearChunkSize

When used with `-XXallocClearChunkSize`, this option sets the size of the chunks to be cleared. If this flag is used but no value is specified, the default is 512 bytes.

Usage:

```
-XXallocClearChunks -XXallocClearChunkSize=<size>[k|K][m|M][g|G]
```

Section 175.4: -XXcallProfiling

This option enables the use of call profiling for code optimizations. Profiling records useful runtime statistics specific to the application and can—in many cases—increase performance because JVM can then act on those statistics.

> Note: This option is supported with the JRockit JVM R27.3.0 and later version. It may become default in future versions.

Usage:

```
java -XXcallProfiling myApp
```

This option is disabled by default. You must enable it to use it.

Section 175.5: -XXdisableFatSpin

This option disables the fat lock spin code in Java, allowing threads that block trying to acquire a fat lock go to sleep directly.

Objects in Java become a lock as soon as any thread enters a synchronized block on that object. All locks are held (that is, stayed locked) until released by the locking thread. If the lock is not going to be released very fast, it can be inflated to a “fat lock.” “Spinning” occurs when a thread that wants a specific lock continuously checks that lock to see if it is still taken, spinning in a tight loop as it makes the check. Spinning against a fat lock is generally beneficial although, in some instances, it can be expensive and might affect performance. `-XXdisableFatSpin` allows you to turn off spinning against a fat lock and eliminate the potential performance hit.

Usage:

```
-XXdisableFatSpin
```

Section 175.6: -XXdisableGCHeuristics

This option disables the garbage collector strategy changes. Compaction heuristics and nursery size heuristics are not affected by this option. By default, the garbage collection heuristics are enabled.

Usage:

```
-XXdisableFatSpin
```

Section 175.7: -XXdumpSize

This option causes a dump file to be generated and allows you to specify the relative size of that file (that is, small, medium, or large).

Usage:

```
-XXdumpsize:<size>
```

<size>	**Description**
`none`	Does not generate a dump file.
`small`	On Windows, a small dump file is generated (on Linux a full core dump is generated). A small dump only include the thread stacks including their traces and very little else. This was the default in the JRockit JVM 8.1 with service packs 1 and 2, as well as 7.0 with service pack 3 and higher).
`normal`	Causes a normal dump to be generated on all platforms. This dump file includes all memory except the java heap. This is the default value for the JRockit JVM 1.4.2 and later.
`large`	Includes everything that is in memory, including the Java heap. This option makes `-XXdumpSize` equivalent to `-XXdumpFullState`.

Section 175.8: -XXexitOnOutOfMemory

This option makes JRockit JVM exit on the first occurrence of an out of memory error. It can be used if you prefer restarting an instance of JRockit JVM rather than handling out of memory errors. Enter this command at startup to force JRockit JVM to exit on the first occurrence of an out of memory error.

Usage:

```
-XXexitOnOutOfMemory
```

Chapter 176: JVM Tool Interface

Section 176.1: Iterate over objects reachable from object (Heap 1.0)

```
#include <vector>
#include <string>


#include "agent_util.hpp"
//this file can be found in Java SE Development Kit 8u101 Demos and Samples
//see http://download.oracle.com/otn-pub/java/jdk/8u101-b13-demos/jdk-8u101-windows-x64-demos.zip
//jdk1.8.0_101.zip!\demo\jvmti\versionCheck\src\agent_util.h

/*
*   Struct used for jvmti->SetTag(object, <pointer to tag>);
*   http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#SetTag
*
*/
typedef struct Tag
{
    jlong referrer_tag;
    jlong size;
    char* classSignature;
    jint hashCode;
} Tag;

/*
* Utility function: jlong -> Tag*
*/
static Tag* pointerToTag(jlong tag_ptr)
{
    if (tag_ptr == 0)
    {
        return new Tag();
    }
    return (Tag*)(ptrdiff_t)(void*)tag_ptr;
}

/*
* Utility function: Tag* -> jlong
*/
static jlong tagToPointer(Tag* tag)
{
    return (jlong)(ptrdiff_t)(void*)tag;
}

/*
*  Heap 1.0 Callback
*  http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#jvmtiObjectReferenceCallback
*/
static jvmtiIterationControl JNICALL heabObjectReferencesCallback(
    jvmtiObjectReferenceKind reference_kind,
    jlong class_tag,
    jlong size,
    jlong* tag_ptr,
    jlong referrer_tag,
    jint referrer_index,
    void* user_data)
{
```

```
    //iterate only over reference field
    if (reference_kind != JVMTI_HEAP_REFERENCE_FIELD)
    {
        return JVMTI_ITERATION_IGNORE;
    }
    auto tag_ptr_list = (std::vector<jlong>*)(ptrdiff_t)(void*)user_data;
    //create and assign tag
    auto t = pointerToTag(*tag_ptr);
    t->referrer_tag = referrer_tag;
    t->size = size;
    *tag_ptr = tagToPointer(t);
    //collect tag
    (*tag_ptr_list).push_back(*tag_ptr);

    return JVMTI_ITERATION_CONTINUE;
}

/*
*  Main function for demonstration of Iterate Over Objects Reachable From Object
*
 http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#IterateOverObjectsReachableFromObject
*
*/
void iterateOverObjectHeapReferences(jvmtiEnv* jvmti, JNIEnv* env, jobject object)
{
    std::vector<jlong> tag_ptr_list;

    auto t = new Tag();
    jvmti->SetTag(object, tagToPointer(t));
    tag_ptr_list.push_back(tagToPointer(t));

    stdout_message("tag list size before call callback:  %d\n", tag_ptr_list.size());
    /*
    *    Call Callback for every reachable object reference
    *    see
http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#IterateOverObjectsReachableFromObject
    */
    jvmti->IterateOverObjectsReachableFromObject(object, &heabObjectReferencesCallback,
(void*)&tag_ptr_list);
    stdout_message("tag list size after call callback:  %d\n", tag_ptr_list.size());

    if (tag_ptr_list.size() > 0)
    {
        jint found_count = 0;
        jlong* tags = &tag_ptr_list[0];
        jobject* found_objects;
        jlong* found_tags;

        /*
        *    collect all tagged object (via *tag_ptr = pointer to tag )
        *    see http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#GetObjectsWithTags
        */
        jvmti->GetObjectsWithTags(tag_ptr_list.size(), tags, &found_count, &found_objects,
&found_tags);
        stdout_message("found %d objects\n", found_count);

        for (auto i = 0; i < found_count; ++i)
        {
            jobject found_object = found_objects[i];

            char* classSignature;
            jclass found_object_class = env->GetObjectClass(found_object);
```

```
            /*
            *    Get string representation of found_object_class
            *    see http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#GetClassSignature
            */
            jvmti->GetClassSignature(found_object_class, &classSignature, nullptr);

            jint hashCode;
            /*
            *    Getting hash code for found_object
            *    see http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#GetObjectHashCode
            */
            jvmti->GetObjectHashCode(found_object, &hashCode);

            //save all it in Tag
            Tag* t = pointerToTag(found_tags[i]);
            t->classSignature = classSignature;
            t->hashCode = hashCode;
        }

        //print all saved information
        for (auto i = 0; i < found_count; ++i)
        {
            auto t = pointerToTag(found_tags[i]);
            auto rt = pointerToTag(t->referrer_tag);

            if (t->referrer_tag != 0)
            {
                stdout_message("referrer object %s#%d --> object %s#%d (size: %2d)\n",
                    rt->classSignature, rt->hashCode, t->classSignature, t->hashCode, t->size);
            }
        }
    }
}
```

Section 176.2: Get JVMTI environment

Inside Agent_OnLoad method:

```
jvmtiEnv* jvmti;
/* Get JVMTI environment */
vm->GetEnv(reinterpret_cast<void **>(&jvmti), JVMTI_VERSION);
```

Section 176.3: Example of initialization inside of Agent_OnLoad method

```
/* Callback for JVMTI_EVENT_VM_INIT */
static void JNICALL vm_init(jvmtiEnv* jvmti, JNIEnv* env, jthread thread)
{
    jint runtime_version;
    jvmti->GetVersionNumber(&runtime_version);
    stdout_message("JVMTI Version: %d\n", runtime_verision);
}

/* Agent_OnLoad() is called first, we prepare for a VM_INIT event here. */
JNIEXPORT jint JNICALL
Agent_OnLoad(JavaVM* vm, char* options, void* reserved)
{
    jint rc;
    jvmtiEventCallbacks callbacks;
```

```
    jvmtiCapabilities capabilities;
    jvmtiEnv* jvmti;

    /* Get JVMTI environment */
    rc = vm->GetEnv(reinterpret_cast<void **>(&jvmti), JVMTI_VERSION);
    if (rc != JNI_OK)
    {
        return -1;
    }

    /* Immediately after getting the jvmtiEnv* we need to ask for the
    *  capabilities this agent will need.
    */
    jvmti->GetCapabilities(&capabilities);
    capabilities.can_tag_objects = 1;
    jvmti->AddCapabilities(&capabilities);

    /* Set callbacks and enable event notifications */
    memset(&callbacks, 0, sizeof(callbacks));
    callbacks.VMInit = &vm_init;

    jvmti->SetEventCallbacks(&callbacks, sizeof(callbacks));
    jvmti->SetEventNotificationMode(JVMTI_ENABLE, JVMTI_EVENT_VM_INIT, nullptr);

    return JNI_OK;
}
```

Chapter 177: Java Memory Management

Section 177.1: Setting the Heap, PermGen and Stack sizes

When a Java virtual machine starts, it needs to know how big to make the Heap, and the default size for thread stacks. These can be specified using command-line options on the `java` command. For versions of Java prior to Java 8, you can also specify the size of the PermGen region of the Heap.

Note that PermGen was removed in Java 8, and if you attempt to set the PermGen size the option will be ignored (with a warning message).

If you don't specify Heap and Stack sizes explicitly, the JVM will use defaults that are calculated in a version and platform specific way. This may result in your application using too little or too much memory. This is typically OK for thread stacks, but it can be problematic for a program that uses a lot of memory.

Setting the Heap, PermGen and default Stack sizes:

The following JVM options set the heap size:

- `-Xms<size>` - sets the initial heap size
- `-Xmx<size>` - sets the maximum heap size
- `-XX:PermSize<size>` - sets the initial PermGen size
- `-XX:MaxPermSize<size>` - sets the maximum PermGen size
- `-Xss<size>` - sets the default thread stack size

The **`<size>`** parameter can be a number of bytes, or can have a suffix of `k`, `m` or `g`. The latter specify the size in kilobytes, megabytes and gigabytes respectively.

Examples:

```
$ java -Xms512m -Xmx1024m JavaApp
$ java -XX:PermSize=64m -XX:MaxPermSize=128m JavaApp
$ java -Xss512k JavaApp
```

Finding the default sizes:

The `-XX:+printFlagsFinal` option can be used to print the values of all flags before starting the JVM. This can be used to print the defaults for the heap and stack size settings as follows:

- For Linux, Unix, Solaris and Mac OSX

  $ java -XX:+PrintFlagsFinal -version | grep -iE 'HeapSize|PermSize|ThreadStackSize'

- For Windows:

  > java -XX:+PrintFlagsFinal -version | findstr /i "HeapSize PermSize ThreadStackSize"

The output of the above commands will resemble the following:

```
uintx InitialHeapSize                          := 20655360        {product}
uintx MaxHeapSize                              := 331350016       {product}
uintx PermSize                                  = 21757952        {pd product}
uintx MaxPermSize                               = 85983232        {pd product}
intx ThreadStackSize                           = 1024            {pd product}
```

The sizes are given in bytes.

Section 177.2: Garbage collection

The C++ approach - new and delete

In a language like C++, the application program is responsible for managing the memory used by dynamically allocated memory. When an object is created in the C++ heap using the `new` operator, there needs to be a corresponding use of the `delete` operator to dispose of the object:

- If program forgets to `delete` an object and just "forgets" about it, the associated memory is lost to the application. The term for this situation is a *memory leak*, and it too much memory leaks an application is liable to use more and more memory, and eventually crash.
- On the other hand, if an application attempts to `delete` the same object twice, or use an object after it has been deleted, then the application is liable to crash due to problems with memory corruption

In a complicated C++ program, implementing memory management using `new` and `delete` can be time consuming. Indeed, memory management is a common source of bugs.

The Java approach - garbage collection

Java takes a different approach. Instead of an explicit `delete` operator, Java provides an automatic mechanism known as garbage collection to reclaim the memory used by objects that are no longer needed. The Java runtime system takes responsibility for finding the objects to be disposed of. This task is performed by a component called a *garbage collector*, or GC for short.

At any time during the execution of a Java program, we can divide the set of all existing objects into two distinct subsets1:

- Reachable objects are defined by the JLS as follows:

  > A reachable object is any object that can be accessed in any potential continuing computation from any live thread.

  In practice, this means that there is a chain of references starting from an in-scope local variable or a `static` variable by which some code might be able to reach the object.

- Unreachable objects are objects that *cannot possibly* be reached as above.

Any objects that are unreachable are *eligible* for garbage collection. This does not mean that they *will* be garbage collected. In fact:

- An unreachable object *does not* get collected immediately on becoming unreachable1.
- An unreachable object *may not ever* be garbage collected.

The Java language Specification gives a lot of latitude to a JVM implementation to decide when to collect unreachable objects. It also (in practice) gives permission for a JVM implementation to be conservative in how it detects unreachable objects.

The one thing that the JLS guarantees is that no *reachable* objects will ever be garbage collected.

What happens when an object becomes unreachable

First of all, nothing specifically happens when an object *becomes* unreachable. Things only happen when the garbage collector runs *and* it detects that the object is unreachable. Furthermore, it is common for a GC run to not detect all unreachable objects.

When the GC detects an unreachable object, the following events can occur.

1. If there are any `Reference` objects that refer to the object, those references will be cleared before the object is deleted.
2. If the object is *finalizable*, then it will be finalized. This happens before the object is deleted.
3. The object can be deleted, and the memory it occupies can be reclaimed.

Note that there is a clear sequence in which the above events *can* occur, but nothing requires the garbage collector to perform the final deletion of any specific object in any specific time-frame.

Examples of reachable and unreachable objects

Consider the following example classes:

```
// A node in simple "open" linked-list.
public class Node {
    private static int counter = 0;

    public int nodeNumber = ++counter;
    public Node next;
}

public class ListTest {
    public static void main(String[] args) {
        test();                    // M1
        System.out.prinln("Done"); // M2
    }

    private static void test() {
        Node n1 = new Node();      // T1
        Node n2 = new Node();      // T2
        Node n3 = new Node();      // T3
        n1.next = n2;              // T4
        n2 = null;                 // T5
        n3 = null;                 // T6
    }
}
```

Let us examine what happens when `test()` is called. Statements T1, T2 and T3 create `Node` objects, and the objects are all reachable via the `n1`, `n2` and `n3` variables respectively. Statement T4 assigns the reference to the 2nd `Node` object to the `next` field of the first one. When that is done, the 2nd `Node` is reachable via two paths:

```
n2 -> Node2
```

```
n1 -> Node1, Node1.next -> Node2
```

In statement T5, we assign `null` to `n2`. This breaks the first of the reachability chains for `Node2`, but the second one remains unbroken, so `Node2` is still reachable.

In statement T6, we assign `null` to `n3`. This breaks the only reachability chain for `Node3`, which makes `Node3` unreachable. However, `Node1` and `Node2` are both still reachable via the `n1` variable.

Finally, when the `test()` method returns, its local variables `n1`, `n2` and `n3` go out of scope, and therefore cannot be accessed by anything. This breaks the remaining reachability chains for `Node1` and `Node2`, and all of the `Node` objects are nor unreachable and *eligible* for garbage collection.

1 - This is a simplification that ignores finalization, and `Reference` classes. 2 - Hypothetically, a Java implementation could do this, but the performance cost of doing this makes it impractical.

Section 177.3: Memory leaks in Java

In the Garbage collection example, we implied that Java solves the problem of memory leaks. This is not actually true. A Java program can leak memory, though the causes of the leaks are rather different.

Reachable objects can leak

Consider the following naive stack implementation.

```
public class NaiveStack {
    private Object[] stack = new Object[100];
    private int top = 0;

    public void push(Object obj) {
        if (top >= stack.length) {
            throw new StackException("stack overflow");
        }
        stack[top++] = obj;
    }

    public Object pop() {
        if (top <= 0) {
            throw new StackException("stack underflow");
        }
        return stack[--top];
    }

    public boolean isEmpty() {
        return top == 0;
    }
}
```

When you `push` an object and then immediately `pop` it, there will still be a reference to the object in the `stack` array.

The logic of the stack implementation means that that reference cannot be returned to a client of the API. If an object has been popped then we can prove that it cannot *"be accessed in any potential continuing computation from any live thread"*. The problem is that a current generation JVM cannot prove this. Current generation JVMs do not consider the logic of the program in determining whether references are reachable. (For a start, it is not practical.)

But setting aside the issue of what *reachability* really means, we clearly have a situation here where the `NaiveStack` implementation is "hanging onto" objects that ought to be reclaimed. That is a memory leak.

In this case, the solution is straightforward:

```
    public Object pop() {
        if (top <= 0) {
            throw new StackException("stack underflow");
        }
        Object popped = stack[--top];
        stack[top] = null;              // Overwrite popped reference with null.
        return popped;
    }
```

Caches can be memory leaks

A common strategy for improving service performance is to cache results. The idea is that you keep a record of common requests and their results in an in-memory data structure known as a cache. Then, each time a request is made, you lookup the request in the cache. If the lookup succeeds, you return the corresponding saved results.

This strategy can be very effective if implemented properly. However, if implemented incorrectly, a cache can be a memory leak. Consider the following example:

```
public class RequestHandler {
    private Map<Task, Result> cache = new HashMap<>();

    public Result doRequest(Task task) {
        Result result = cache.get(task);
        if (result == null) {
            result == doRequestProcessing(task);
            cache.put(task, result);
        }
        return result;
    }
}
```

The problem with this code is that while any call to `doRequest` could add a new entry to the cache, there is nothing to remove them. If the service is continually getting different tasks, then the cache will eventually consume all available memory. This is a form of memory leak.

One approach to solving this is to use a cache with a maximum size, and throw out old entries when the cache exceeds the maximum. (Throwing out the least recently used entry is a good strategy.) Another approach is to build the cache using `WeakHashMap` so that the JVM can evict cache entries if the heap starts getting too full.

Section 177.4: Finalization

A Java object may declare a `finalize` method. This method is called just before Java releases the memory for the object. It will typically look like this:

```
public class MyClass {

    //Methods for the class

    @Override
    protected void finalize() throws Throwable {
        // Cleanup code
    }
}
```

However, there some important caveats on the behavior of Java finalization.

- Java makes no guarantees about when a `finalize()` method will called.
- Java does not even guarantee that a `finalize()` method will be called some time during the running application's lifetime.
- The only thing that is guaranteed is that the method will be called before the object is deleted ... if the object is deleted.

The caveats above mean that it is a bad idea to rely on the `finalize` method to perform cleanup (or other) actions that must be performed in a timely fashion. Over reliance on finalization can lead to storage leaks, memory leaks and other problems.

In short, there are very few situation where finalization is actually a good solution.

Finalizers only run once

Normally, an object is deleted after it has been finalized. However, this doesn't happen all of the time. Consider the following example1:

```
public class CaptainJack {
    public static CaptainJack notDeadYet = null;

    protected void finalize() {
        // Resurrection!
        notDeadYet = this;
    }
}
```

When an instance of `CaptainJack` becomes unreachable and the garbage collector attempts to reclaim it, the `finalize()` method will assign a reference to the instance to the `notDeadYet` variable. That will make the instance reachable once more, and the garbage collector won't delete it.

Question: Is Captain Jack immortal?

Answer: No.

The catch is the JVM will only run a finalizer on an object once in its lifetime. If you assign **`null`** to `notDeadYet` causing a resurected instance to be unreachable once more, the garbage collector won't call `finalize()` on the object.

1 - See https://en.wikipedia.org/wiki/Jack_Harkness.

Section 177.5: Manually triggering GC

You can manually trigger the Garbage Collector by calling

```
System.gc();
```

However, Java does not guarantee that the Garbage Collector has run when the call returns. This method simply "suggests" to the JVM (Java Virtual Machine) that you want it to run the garbage collector, but does not force it to do so.

It is generally considered a bad practice to attempt to manually trigger garbage collection. The JVM can be run with the `-XX:+DisableExplicitGC` option to disable calls to `System.gc()`. Triggering garbage collection by calling `System.gc()` can disrupt normal garbage management / object promotion activities of the specific garbage collector implementation in use by the JVM.

Chapter 178: Java Performance Tuning

Section 178.1: An evidence-based approach to Java performance tuning

Donald Knuth is often quoted as saying this:

> "Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. *We should forget about small efficiencies, say about 97% of the time*: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%."

source

Bearing that sage advice in mind, here is the recommended procedure for optimizing programs:

1. First of all, design and code your program or library with a focus on simplicity and correctness. To start with, don't spend much effort on performance.
2. Get it to a working state, and (ideally) develop unit tests for the key parts of the codebase.
3. Develop an application level performance benchmark. The benchmark should cover the performance critical aspects of your application, and should perform a range of tasks that are typical of how the application will be used in production.
4. Measure the performance.
5. Compare the measured performance against your criteria for how fast the application needs to be. (Avoid unrealistic, unattainable or unquantifiable criteria such as "as fast as possible".)
6. If you have met the criteria, STOP. You job is done. (Any further effort is probably a waste of time.)
7. Profile the application while it is running your performance benchmark.
8. Examine the profiling results and pick the biggest (unoptimized) "performance hotspots"; i.e. sections of the code where the application seems to be spending the most time.
9. Analyse the hotspot code section to try to understand why it is a bottleneck, and think of a way to make it faster.
10. Implement that as a proposed code change, test and debug.
11. Rerun the benchmark to see if the code change has improved the performance:
    - If Yes, then return to step 4.
    - If No, then abandon the change and return to step 9. If you are making no progress, pick a different hotspot for your attention.

Eventually you will get to a point where the application is either fast enough, or you have considered all of the significant hotspots. At this point you need to stop this approach. If a section of code is consuming (say) 1% of the overall time, then even a 50% improvement is only going to make the application 0.5% faster overall.

Clearly, there is a point beyond which hotspot optimization is a waste of effort. If you get to that point, you need to

take a more radical approach. For example:

- Look at the algorithmic complexity of your core algorithms.
- If the application is spending a lot of time garbage collection, look for ways to reduce the rate of object creation.
- If key parts of the application are CPU intensive and single-threaded, look for opportunities for parallelism.
- If the application is already multi-threaded, look for concurrency bottlenecks.

But wherever possible, rely on tools and measurement rather than instinct to direct your optimization effort.

Section 178.2: Reducing amount of Strings

In Java, it's too "easy" to create many String instances which are not needed. That and other reasons might cause your program to have lots of Strings that the GC is busy cleaning up.

Some ways you might be creating String instances:

```
myString += "foo";
```

Or worse, in a loop or recursion:

```
for (int i = 0; i < N; i++) {
    myString += "foo" + i;
}
```

The problem is that each + creates a new String (usually, since new compilers optimize some cases). A possible optimization can be made using `StringBuilder` or `StringBuffer`:

```
StringBuffer sb = new StringBuffer(myString);
for (int i = 0; i < N; i++) {
    sb.append("foo").append(i);
}
myString = sb.toString();
```

If you build long Strings often (SQLs for example), use a String building API.

Other things to consider:

- Reduce usage of `replace`, `substring` etc.
- Avoid `String.toArray()`, especially in frequently accessed code.
- Log prints which are destined to be filtered (due to log level for example) should not be generated (log level should be checked in advance).
- Use libraries like this if necessary.
- StringBuilder is better if the variable is used in a non-shared manner (across threads).

Section 178.3: General approach

The internet is packed with tips for performance improvement of Java programs. Perhaps the number one tip is awareness. That means:

- Identify possible performance problems and bottlenecks.
- Use analyzing and testing tools.
- Know good practices and bad practices.

The first point should be done during the design stage if speaking about a new system or module. If speaking about legacy code, analyzing and testing tools come into the picture. The most basic tool for analyzing your JVM performance is JVisualVM, which is included in the JDK.

The third point is mostly about experience and extensive research, and of course raw tips that will show up on this page and others, like this.

Chapter 179: Benchmarks

Writing performance benchmarks in java is not as simple as getting `System.currentTimeMillis()` in the beginning and in the end and calculating the difference. To write valid performance benchmarks, one should use proper tools.

Section 179.1: Simple JMH example

One of the tools for writing proper benchmark tests is JMH. Let's say we want to compare performance of searching an element in `HashSet` vs `TreeSet`.

The easiest way to get JHM into your project - is to use maven and shade plugin. Also you can see `pom.xml` from JHM examples.

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
            <version>3.0.0</version>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>shade</goal>
                    </goals>
                    <configuration>
                        <finalName>/benchmarks</finalName>
                        <transformers>
                            <transformer

 implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                                <mainClass>org.openjdk.jmh.Main</mainClass>
                            </transformer>
                        </transformers>
                        <filters>
                            <filter>
                                <artifact>*:*</artifact>
                                <excludes>
                                    <exclude>META-INF/*.SF</exclude>
                                    <exclude>META-INF/*.DSA</exclude>
                                    <exclude>META-INF/*.RSA</exclude>
                                </excludes>
                            </filter>
                        </filters>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>

<dependencies>
    <dependency>
        <groupId>org.openjdk.jmh</groupId>
        <artifactId>jmh-core</artifactId>
        <version>1.18</version>
    </dependency>
    <dependency>
        <groupId>org.openjdk.jmh</groupId>
```

```
        <artifactId>jmh-generator-annprocess</artifactId>
        <version>1.18</version>
    </dependency>
</dependencies>
```

After this you need to write benchmark class itself:

```
package benchmark;

import org.openjdk.jmh.annotations.*;
import org.openjdk.jmh.infra.Blackhole;

import java.util.HashSet;
import java.util.Random;
import java.util.Set;
import java.util.TreeSet;
import java.util.concurrent.TimeUnit;

@State(Scope.Thread)
public class CollectionFinderBenchmarkTest {
    private static final int SET_SIZE = 10000;

    private Set<String> hashSet;
    private Set<String> treeSet;

    private String stringToFind = "8888";

    @Setup
    public void setupCollections() {
        hashSet = new HashSet<>(SET_SIZE);
        treeSet = new TreeSet<>();

        for (int i = 0; i < SET_SIZE; i++) {
            final String value = String.valueOf(i);
            hashSet.add(value);
            treeSet.add(value);
        }

        stringToFind = String.valueOf(new Random().nextInt(SET_SIZE));
    }

    @Benchmark
    @BenchmarkMode(Mode.AverageTime)
    @OutputTimeUnit(TimeUnit.NANOSECONDS)
    public void testHashSet(Blackhole blackhole) {
        blackhole.consume(hashSet.contains(stringToFind));
    }

    @Benchmark
    @BenchmarkMode(Mode.AverageTime)
    @OutputTimeUnit(TimeUnit.NANOSECONDS)
    public void testTreeSet(Blackhole blackhole) {
        blackhole.consume(treeSet.contains(stringToFind));
    }
}
```

Please keep in mind this `blackhole.consume()`, we'll get back to it later. Also we need main class for running benchmark:

```
package benchmark;
```

```java
import org.openjdk.jmh.runner.Runner;
import org.openjdk.jmh.runner.RunnerException;
import org.openjdk.jmh.runner.options.Options;
import org.openjdk.jmh.runner.options.OptionsBuilder;

public class BenchmarkMain {
    public static void main(String[] args) throws RunnerException {
        final Options options = new OptionsBuilder()
                .include(CollectionFinderBenchmarkTest.class.getSimpleName())
                .forks(1)
                .build();

        new Runner(options).run();
    }
}
```

And we're all set. We just need to run `mvn package` (it will create `benchmarks.jar` in your `/target` folder) and run our benchmark test:

```
java -cp target/benchmarks.jar benchmark.BenchmarkMain
```

And after some warmup and calculation iterations, we will have our results:

```
# Run complete. Total time: 00:01:21

Benchmark                                  Mode  Cnt   Score    Error  Units
CollectionFinderBenchmarkTest.testHashSet  avgt   20   9.940 ±  0.270  ns/op
CollectionFinderBenchmarkTest.testTreeSet  avgt   20  98.858 ± 13.743  ns/op
```

About that `blackhole.consume()`. If your calculations do not change the state of your application, java will most likely just ignore it. So, in order to avoid it, you can either make your benchmark methods return some value, or use `Blackhole` object to consume it.

You can find more information about writing proper benchmarks in Aleksey Shipilëv's blog, in Jacob Jenkov's blog and in java-performance blog: 1, 2.

Chapter 180: FileUpload to AWS

Upload File to AWS s3 bucket using spring rest API.

Section 180.1: Upload file to s3 bucket

Here we will create a rest API which will take file object as a multipart parameter from front end and upload it to S3 bucket using java rest API.

Requirement: secrete key and Access key for s3 bucket where you wanna upload your file.

code: DocumentController.java

```
@RestController
@RequestMapping("/api/v2")
public class DocumentController {

    private static String bucketName = "pharmerz-chat";
    //   private static String keyName        = "Pharmerz"+ UUID.randomUUID();

    @RequestMapping(value = "/upload", method = RequestMethod.POST, consumes =
MediaType.MULTIPART_FORM_DATA)
    public URL uploadFileHandler(@RequestParam("name") String name,
                                 @RequestParam("file") MultipartFile file) throws IOException {

/******* Printing all the possible parameter from @RequestParam *************/

        System.out.println("*****************************");

        System.out.println("file.getOriginalFilename() " + file.getOriginalFilename());
        System.out.println("file.getContentType()" + file.getContentType());
        System.out.println("file.getInputStream() " + file.getInputStream());
        System.out.println("file.toString() " + file.toString());
        System.out.println("file.getSize() " + file.getSize());
        System.out.println("name " + name);
        System.out.println("file.getBytes() " + file.getBytes());
        System.out.println("file.hashCode() " + file.hashCode());
        System.out.println("file.getClass() " + file.getClass());
        System.out.println("file.isEmpty() " + file.isEmpty());

        /*************Parameters to b pass to s3 bucket put Object **************/
        InputStream is = file.getInputStream();
        String keyName = file.getOriginalFilename();


// Credentials for Aws
        AWSCredentials credentials = new BasicAWSCredentials("AKIA*************",
"zr**********************");

        /****************** DocumentController.uploadfile(credentials); *************************/


        AmazonS3 s3client = new AmazonS3Client(credentials);
        try {
            System.out.println("Uploading a new object to S3 from a file\n");
            //File file = new File(awsuploadfile);
            s3client.putObject(new PutObjectRequest(
                    bucketName, keyName, is, new ObjectMetadata()));
```

```
            URL url = s3client.generatePresignedUrl(bucketName, keyName,
Date.from(Instant.now().plus(5, ChronoUnit.MINUTES)));
            // URL url=s3client.generatePresignedUrl(bucketName,keyName,
Date.from(Instant.now().plus(5, ChronoUnit.)));
            System.out.println("************************************");
            System.out.println(url);

            return url;

        } catch (AmazonServiceException ase) {
            System.out.println("Caught an AmazonServiceException, which " +
                    "means your request made it " +
                    "to Amazon S3, but was rejected with an error response" +
                    " for some reason.");
            System.out.println("Error Message:    " + ase.getMessage());
            System.out.println("HTTP Status Code: " + ase.getStatusCode());
            System.out.println("AWS Error Code:   " + ase.getErrorCode());
            System.out.println("Error Type:       " + ase.getErrorType());
            System.out.println("Request ID:       " + ase.getRequestId());
        } catch (AmazonClientException ace) {
            System.out.println("Caught an AmazonClientException, which " +
                    "means the client encountered " +
                    "an internal error while trying to " +
                    "communicate with S3, " +
                    "such as not being able to access the network.");
            System.out.println("Error Message: " + ace.getMessage());
        }

        return null;

    }


}
```

Front end Function

```
var form = new FormData();
form.append("file", "image.jpeg");

var settings = {
 "async": true,
 "crossDomain": true,
 "url": "http://url/",
 "method": "POST",
 "headers": {
   "cache-control": "no-cache"
 },
 "processData": false,
 "contentType": false,
 "mimeType": "multipart/form-data",
 "data": form
}

$.ajax(settings).done(function (response) {
 console.log(response);
});
```

Chapter 181: AppDynamics and TIBCO BusinessWorks Instrumentation for Easy Integration

As AppDynamics aims to provide a way to measure application performance, speed of development, delivery (deployment) of applications is an essential factor in making DevOps efforts a true success. Monitoring a TIBCO BW application with AppD is generally simple and not time consuming but when deploying large sets of applications rapid instrumentation is key. This guide shows how to instrument all of your BW applications in a single step without modifying each application before deploying.

Section 181.1: Example of Instrumentation of all BW Applications in a Single Step for Appdynamics

1. Locate and open your TIBCO BW bwengine.tra file typlically under TIBCO_HOME/bw/5.12/bin/bwengine.tra (Linux environment)
2. Look for the line that states:

***** Common variables. Modify these only. *****

3. Add the following line right after that section tibco.deployment=%tibco.deployment%
4. Go to the end of the file and add (replace ? with your own values as needed or remove the flag that does not apply): java.extended.properties=-javaagent:/opt/appd/current/appagent/javaagent.jar -Dappdynamics.http.proxyHost=? -Dappdynamics.http.proxyPort=? -Dappdynamics.agent.applicationName=? -Dappdynamics.agent.tierName=? -Dappdynamics.agent.nodeName=%tibco.deployment% -Dappdynamics.controller.ssl.enabled=? -Dappdynamics.controller.sslPort=? -Dappdynamics.agent.logs.dir=? -Dappdynamics.agent.runtime.dir=? -Dappdynamics.controller.hostName=? -Dappdynamics.controller.port=? -Dappdynamics.agent.accountName=? -Dappdynamics.agent.accountAccessKey=?
5. Save file and redeploy. All your applications should now be instrumented automatically at deployment time.

Appendix A: Installing Java (Standard Edition)

This documentation page gives access to instructions for installing `java standard edition` on `Windows`, `Linux`, and `macOS` computers.

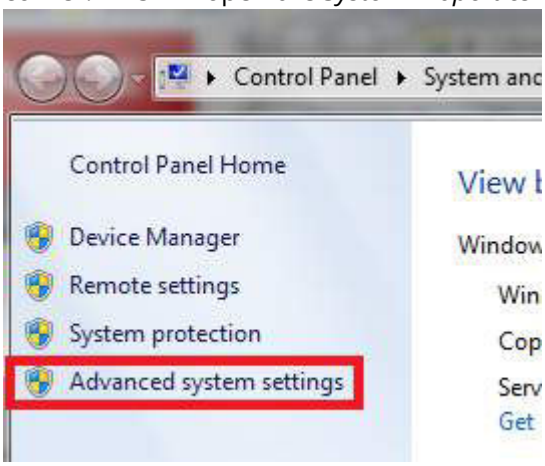
Section A.1: Setting %PATH% and %JAVA_HOME% after installing on Windows

Assumptions:

- An Oracle JDK has been installed.
- The JDK was installed to the default directory.

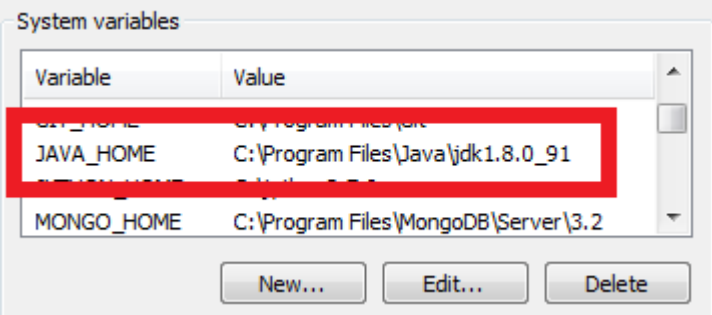
Setup steps

1. Open Windows Explorer.
2. In the navigation pane on the left right click on *This PC* (or *Computer* for older Windows versions). There is a shorter way without using the explorer in actual Windows versions: Just press `Win`+`Pause`
3. In the newly opened Control Panel window, left click *Advanced System Settings* which should be in the top left corner. This will open the *System Properties* window.

   Alternatively, type `SystemPropertiesAdvanced` (case insensitive) in the *Run* (`Win`+`R`), and hit `Enter`.

4. In the *Advanced* tab of *System Properties* select the `Environment Variables...` button in the lower right corner of the window.
5. Add a **New System Variable** by clicking the `New...` button in *System Variables* with the name `JAVA_HOME` and whose value is the path to the directory where the JDK was installed. After entering these values, press `OK`.



6. Scroll down the list of *System Variables* and select the `Path` variable.

> 7. **CAUTION:** Windows relies on `Path` to find important programs. If any or all of it is removed, Windows may not be able to function properly. It must be modified to allow Windows to run the JDK. With this in mind ,click the "Edit..." button with the `Path` variable selected. Add `%JAVA_HOME%\bin;` to the beginning of the `Path` variable.

It is better to append at the beginning of the line because Oracle's software used to register their own version of Java in `Path` - This will cause your version to be ignored if it occurs after Oracle's declaration.

Check your work

1. Open the command prompt by clicking Start then typing `cmd` and pressing `Enter`.
2. Enter `javac -version` into the prompt. If it was successful, then the version of the JDK will be printed to the screen.

Note: If you have to try again, close the prompt before checking your work. This will force windows to get the new version of `Path`.

Section A.2: Installing a Java JDK on Linux

Using the Package Manager

JDK and/or JRE releases for OpenJDK or Oracle can be installed using the package manager on most mainstream Linux distribution. (The choices that are available to you will depend on the distro.)

As a general rule, the procedure is to open terminal window and run the commands shown below. (It is assumed that you have sufficient access to run commands as the "root" user ... which is what the `sudo` command does. If you do not, then please talk to your system's administrators.)

Using the package manager is recommended because it (generally) makes it easier to keep your Java installation up to date.

`apt-get`, Debian based Linux distributions (Ubuntu, etc)

The following instructions will install Oracle Java 8:

```
$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt-get update
$ sudo apt-get install oracle-java8-installer
```

Note: To automatically set up the Java 8 environment variables, you can install the following package:

```
$ sudo apt-get install oracle-java8-set-default
```

Creating a `.deb` file

If you prefer to create the `.deb` file yourself from the `.tar.gz` file downloaded from Oracle, do the following (assuming you've downloaded the `.tar.gz` to `./<jdk>.tar.gz`):

```
$ sudo apt-get install java-package # might not be available in default repos
$ make-jpkg ./<jdk>.tar.gz          # should not be run as root
$ sudo dpkg -i *j2sdk*.deb
```

Note: This expects the input to be provided as a ".tar.gz" file.

`slackpkg`, Slackware based Linux distributions

```
sudo slapt-get install default-jdk
```

`yum`, RedHat, CentOS, etc

```
sudo yum install java-1.8.0-openjdk-devel.x86_64
```

`dnf`, Fedora

On recent Fedora releases, `yum` has been superseded by `dnf`.

```
sudo dnf install java-1.8.0-openjdk-devel.x86_64
```

In recent Fedora releases, there are no packages for installing Java 7 and earlier.

`pacman`, Arch based Linux distributions

```
sudo pacman -S jdk8-openjdk
```

Using `sudo` is not required if you're running as the root user.

Gentoo Linux

The Gentoo Java guide is maintained by the Gentoo Java team and keeps an updated wiki page including the correct portage packages and USE flags needed.

Installing Oracle JDKs on Redhat, CentOS, Fedora

Installing JDK from an Oracle JDK or JRE `tar.gz` file.

1. Download the appropriate Oracle archive ("tar.gz") file for the desired release from the Oracle Java downloads site.
2. Change directory to the place where you want to put the installation;
3. Decompress the archive file; e.g.

```
tar xzvf jdk-8u67-linux-x64.tar.gz
```

Installing from an Oracle Java RPM file.

1. Retrieve the required RPM file for the desired release from the Oracle Java downloads site.
2. Install using the `rpm` command. For example:

```
$ sudo rpm -ivh jdk-8u67-linux-x644.rpm
```

Section A.3: Installing a Java JDK on macOS

Oracle Java 7 and Java 8

Java 7 and Java 8 for macOS are available from Oracle. This Oracle page answers a lot of questions about Java for Mac. Note that Java 7 prior to 7u25 have been disabled by Apple for security reasons.

In general, Oracle Java (Version 7 and later) requires an Intel-based Mac running macOS 10.7.3 or later.

Installation of Oracle Java

Java 7 & 8 JDK and JRE installers for macOS can be downloaded from Oracle's website:

- Java 8 - Java SE Downloads
- Java 7 - Oracle Java Archive.

After downloading the relevant package, double click on the package and go through the normal installation process. A JDK should get installed here:

```
/Library/Java/JavaVirtualMachines/<version>.jdk/Contents/Home
```

where corresponds to the installed version.

Command-Line Switching

When Java is installed, the installed version is automatically set as the default. To switch between different, use:

```
export JAVA_HOME=/usr/libexec/java_home -v 1.6 #Or 1.7 or 1.8
```

The following functions can be added to the ~/`.bash_profile` (If you use the default Bash shell) for ease of use:

```
function java_version {
   echo 'java -version';
}

function java_set {
   if [[ $1 == "6" ]]
   then
       export JAVA_HOME='/usr/libexec/java_home -v 1.6';
       echo "Setting Java to version 6..."
       echo "$JAVA_HOME"
   elif [[ $1 == "7" ]]
   then
       export JAVA_HOME='/usr/libexec/java_home -v 1.7';
       echo "Setting Java to version 7..."
       echo "$JAVA_HOME"
   elif [[ $1 == "8" ]]
   then
```

```
        export JAVA_HOME='/usr/libexec/java_home -v 1.8';
        echo "Setting Java to version 8..."
        echo "$JAVA_HOME"
    fi
}
```

Apple Java 6 on macOS

On older versions of macOS (10.11 El Capitan and earlier), Apple's release of Java 6 comes pre-installed. If installed, it can be be found at this location:

```
/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
```

Note that Java 6 passed its end-of-life long ago, so upgrading to a newer version is recommended. There is more information on reinstalling Apple Java 6 on the Oracle website.

Section A.4: Installing a Java JDK or JRE on Windows

Only Oracle JDKs and JREs are available for Windows platforms. The installation procedure is straight-forward:

1. Visit the Oracle Java Downloads page:
2. Click on either the JDK button, the JRE button or the Server JRE button. Note that to develop using Java you need JDK. To know the difference between JDK and JRE, see here
3. Scroll down to the version you want to download. (Generally speaking, the most recent one is recommended.)
4. Select the "Accept License Agreement" radio button.
5. Download the Windows x86 (32 bit) or Windows x64 (64 bit) installer.
6. Run the installer ... in the normal way for your version of Windows.

An alternate way to install Java on Windows using the command prompt is to use Chocolately:

1. Install Chocolately from https://chocolatey.org/
2. Open a cmd instance, for example hit `Win` + `R` and then type "cmd" in the "Run" window followed by an enter.
3. In your cmd instance, run the following command to download and install a Java 8 JDK:

```
C:\> choco install jdk8
```

Getting up and running with portable versions

There are instances where you might want to install JDK/JRE on a system with limited privileges like a VM or you might want to install and use multiple versions or architectures (x64/x86) of JDK/JRE. The steps remain same till the point you download the installer (.EXE). The steps after that are as follows (The steps are applicable for JDK/JRE 7 and above, for older versions they are slightly different in the names of folders and files):

1. Move the file to an appropriate location where you would want your Java binaries to reside permanently.
2. Install 7-Zip or its portable version if you have limited privileges.
3. With 7-Zip, extract the files from the Java installer EXE to the location.
4. Open up command prompt there by holding `Shift` and `Right-Click`ing in the folder in explorer or navigate to that location from anywhere.

5. Navigate to the newly created folder. Let's say the folder name is `jdk-7u25-windows-x64`. So type `cd jdk-7u25-windows-x64`. Then type the following commands in order :

   ```
   cd .rsrc\JAVA_CAB10
   ```

   ```
   extrac32 111
   ```

6. This will create a `tools.zip` file in that location. Extract the `tools.zip` with 7-Zip so that the files inside it are now created under `tools` in the same directory.
7. Now execute these commands on the already opened command prompt :

   ```
   cd tools
   ```

   ```
   for /r %x in (*.pack) do .\bin\unpack200 -r "%x" "%~dx%~px%~nx.jar"
   ```

8. Wait for the command to complete. Copy the contents of `tools` to the location where you want your binaries to be.

This way, you can install any versions of JDK/JRE you need to be installed simultaneously.

Original post : http://stackoverflow.com/a/6571736/1448252

Section A.5: Configuring and switching Java versions on Linux using alternatives

Using Alternatives

Many Linux distributions use the `alternatives` command for switching between different versions of a command. You can use this for switching between different versions of Java installed on a machine.

1. In a command shell, set $JDK to the pathname of a newly installed JDK; e.g.

   ```
   $ JDK=/Data/jdk1.8.0_67
   ```

2. Use `alternatives --install` to add the commands in the Java SDK to alternatives:

   ```
   $ sudo alternatives --install /usr/bin/java java $JDK/bin/java 2
   $ sudo alternatives --install /usr/bin/javac javac $JDK/bin/javac 2
   $ sudo alternatives --install /usr/bin/jar jar $JDK/bin/jar 2
   ```

   And so on.

Now you can switch between different versions of a Java command as follows:

```
$ sudo alternatives --config javac

There is 1 program that provides 'javac'.

  Selection    Command
-----------------------------------------------
*+ 1           /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.101-1.b14.fc23.x86_64/bin/javac
```

```
   2           /Data/jdk1.8.0_67/bin/javac

Enter to keep the current selection[+], or type selection number: 2
$
```

For more information on using `alternatives`, refer to the alternatives(8) manual entry.

Arch based installs

Arch Linux based installs come with the command `archlinux-java` to switch java versions.

Listing installed environments

```
$ archlinux-java status
Available Java environments:
  java-7-openjdk (default)
  java-8-openjdk/jre
```

Switching current environment

```
# archlinux-java set <JAVA_ENV_NAME>
```

Eg:

```
# archlinux-java set java-8-openjdk/jre
```

More information can be found on the Arch Linux Wiki

Section A.6: What do I need for Java Development

A JDK installation and a text editor are the bare minimum for Java development. (It is nice to have a text editor that can do Java syntax highlighting, but you can do without.)

However for serious development work it is recommended that you also use the following:

- A Java IDE such as Eclipse, Intellij IDEA or NetBeans
- A Java build tool such as Ant, Gradle or Maven
- A version control system for managing your code base (with appropriate backups, and off-site replication)
- Test tools and CI (continuous integration) tools

Section A.7: Selecting an appropriate Java SE release

There have been many releases of Java since the original Java 1.0 release in 1995. (Refer to Java version history for a summary.) However most releases have passed their official End Of Life dates. This means that the vendor (typically Oracle now) has ceased new development for the release, and no longer provides public / free patches for any bugs or security issues. (Private patch releases are typically available for people / organizations with a support contract; contact your vendor's sales office.)

In general, the recommended Java SE release for use will be the latest update for the latest public version. Currently, this means the latest available Java 8 release. Java 9 is due for public release in 2017. (Java 7 has passed its End Of Life and the last public release was in April 2015. Java 7 and earlier releases are not recommended.)

This recommendation applies for all new Java development, and anyone learning Java. It also applies to people who just want to run Java software provided by a third-party. Generally speaking, well-written Java code will work on a newer release of Java. (But check the software's release notes, and contact the author / supplier / vendor if you have doubts.)

If you are working on an older Java codebase, you would be advised to ensure that your code runs on the latest release of Java. Deciding when to start using the features of newer Java releases is more difficult, as this will impact your ability to support customers who are unable or unwilling their Java installation.

Section A.8: Java release and version naming

Java release naming is a little confusing. There are actually two systems of naming and numbering, as shown in this table:

JDK version	Marketing name
jdk-1.0	JDK 1.0
jdk-1.1	JDK 1.1
jdk-1.2	J2SE 1.2
...	...
jdk-1.5	J2SE 1.5 rebranded Java SE 5
jdk-1.6	Java SE 6
jdk-1.7	Java SE 7
jdk-1.8	Java SE 8
jdk-91	Java SE 9 (not released yet)

1 - It appears that Oracle intends to break from their previous practice of using a "semantic version number" scheme in the Java version strings. It remains to be seen if they will follow through with this.

The "SE" in the marketing names refers to Standard Edition. This is the base release for running Java on most laptops, PCs and servers (apart from Android).

There are two other official editions of Java: "Java ME" is the Micro Edition, and "Java EE" is the Enterprise Edition. The Android flavor of Java is also significantly different from Java SE. Java ME, Java EE and Android Java are outside of the scope of this Topic.

The full version number for a Java release looks like this:

```
1.8.0_101-b13
```

This says JDK 1.8.0, Update 101, Build #13. Oracle refers to this in the release notes as:

```
Java™ SE Development Kit 8, Update 101 (JDK 8u101)
```

The update number is important -- Oracle regularly issue updates to a major release with security patches, bug fixes and (in some cases) new features. The build number is usually irrelevant. Note that Java 8 and Java 1.8 *refer to the same thing*; Java 8 is just the "marketing" name for Java 1.8.

Section A.9: Installing Oracle Java on Linux with latest tar file

Follow the below steps to install Oracle JDK from the latest tar file:

1. Download the latest tar file from here - Current latest is Java SE Development Kit 8u112.
2. You need sudo privilages:

```
sudo su
```

3. Create a dir for jdk install:

```
mkdir /opt/jdk
```

4. Extract downloaded tar into it:

```
tar -zxf jdk-8u5-linux-x64.tar.gz -C /opt/jdk
```

5. Verify if the files are extracted:

```
ls /opt/jdk
```

6. Setting Oracle JDK as the default JVM:

```
update-alternatives --install /usr/bin/java java /opt/jdk/jdk1.8.0_05/bin/java 100
```

and

```
update-alternatives --install /usr/bin/javac javac /opt/jdk/jdk1.8.0_05/bin/javac 100
```

7. Check Java version:

java -version

Expected output:

```
java version "1.8.0_111"
Java(TM) SE Runtime Environment (build 1.8.0_111-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.111-b14, mixed mode)
```

Section A.10: Post-installation checking and configuration on Linux

After installing a Java SDK, it is advisable to check that it is ready to use. You can do this by running these two commands, using your normal user account:

```
$ java -version
$ javac -version
```

These commands print out the version information for the JRE and JDK (respectively) that are on your shell's command search path. Look for the JDK / JRE version string.

- If either of the above commands fails, saying "command not found", then the JRE or JDK is not on the search path at all; go to **Configuring PATH directly** below.
- If either of the above commands displays a different version string to what you were expecting, then either your search path or the "alternatives" system needs adjusting; go to **Checking Alternatives**
- If the correct version strings are displayed, you are nearly done; skip to **Checking JAVA_HOME**

Configuring PATH directly

If there is no `java` or `javac` on the search path at the moment, then the simple solution is to add it to your search

path.

First, find where you installed Java; see **Where was Java installed?** below if you have doubts.

Next, assuming that `bash` is your command shell, use a text editor to add the following lines to the end of either `~/.bash_profile` or `~/.bashrc` (If you use Bash as your shell).

```
JAVA_HOME=<installation directory>
PATH=$JAVA_HOME/bin:$PATH

export JAVA_HOME
export PATH
```

... replacing `<installation directory>` with the pathname for your Java installation directory. Note that the above assumes that the installation directory contains a `bin` directory, and the `bin` directory contains the `java` and `javac` commands that you are trying to use.

Next, source the file that you just edited, so that the environment variables for your current shell are updated.

```
$ source ~/.bash_profile
```

Next, repeat the `java` and `javac` version checks. If there are still problems, use `which java` and `which javac` to verify that you have updates the environment variables correctly.

Finally, logout and login again so that the updated environment variables ptopagate to all of your shells. You should now be done.

Checking Alternatives

If `java -version` or `javac -version` worked but gave an unexpected version number, you need to check where the commands are coming from. Use `which` and `ls -l` to find this out as follows:

```
$ ls -l `which java`
```

If the output looks like this, :

```
lrwxrwxrwx. 1 root root 22 Jul 30 22:18 /usr/bin/java -> /etc/alternatives/java
```

then the `alternatives` version switching is being used. You needs to decide whether to continue using it, or simply override it by setting the `PATH` directly.

- Configuring and Switching Java versions on Linux using alternatives
- See "Configuring PATH directly" above.

Where was Java installed?

Java can be installed in a variety of places, depending on the installation method.

- The Oracle RPMs put the Java installation in "/usr/java".
- On Fedora, the default location is "/usr/lib/jvm".
- If Java was installed by hand from ZIP or JAR files, the installation could be anywhere.

If you are having difficultly finding the installation directory, We suggest that you use `find` (or `slocate`) to find the command. For example:

```
$ find / -name java -type f 2> /dev/null
```

This gives you the pathnames for all files called `java` on your system. (The redirection of standard error to "/dev/null" suppresses messages about files and directories that you can't access.)

Appendix B: Java Editions, Versions, Releases and Distributions

Section B.1: Differences between Java SE JRE or Java SE JDK distributions

Sun / Oracle releases of Java SE come in two forms: JRE and JDK. In simple terms, JREs support running Java applications, and JDKs also support Java development.

Java Runtime Environment

Java Runtime Environment or JRE distributions consist of the set of libraries and tools needed to run and manage Java applications. The tools in a typical modern JRE include:

- The `java` command for running a Java program in a JVM (Java Virtual Machine)
- The `jjs` command for running the Nashorn Javascript engine.
- The `keytool` command for manipulating Java keystores.
- The `policytool` command for editing security sandbox security policies.
- The `pack200` and `unpack200` tools for packing and unpacking "pack200" file for web deployment.
- The `orbd`, `rmid`, `rmiregistry` and `tnameserv` commands that support Java CORBA and RMI applications.

"Desktop JRE" installers include a Java plugin suitable for some web browser. This is deliberately left out of "Server JRE" installers.linux syscall read benchmarku

From Java 7 update 6 onwards, JRE installers have included JavaFX (version 2.2 or later).

Java Development Kit

A Java Development Kit or JDK distribution includes the JRE tools, and additional tools for developing Java software. The additional tools typically include:

- The `javac` command, which compiles Java source code (".java") to bytecode files (".class").
- The tools for creating JAR files such as `jar` and `jarsigner`
- Development tools such as:
    - `appletviewer` for running applets
    - `idlj` the CORBA IDL to Java compiler
    - `javah` the JNI stub generator
    - `native2ascii` for character set conversion of Java source code
    - `schemagen` the Java to XML schema generator (part of JAXB)
    - `serialver` generate Java Object Serialization version string.
    - the `wsgen` and `wsimport` support tools for JAX-WS
- Diagnostic tools such as:
    - `jdb` the basic Java debugger
    - `jmap` and `jhat` for dumping and analysing a Java heap.
    - `jstack` for getting a thread stack dump.
    - `javap` for examining ".class" files.
- Application management and monitoring tools such as:
    - `jconsole` a management console,
    - `jstat`, `jstatd`, `jinfo` and `jps` for application monitoring

A typical Sun / Oracle JDK installation also includes a ZIP file with the source code of the Java libraries. Prior to Java 6, this was the only publicly available Java source code.

From Java 6 onwards, the complete source code for OpenJDK is available for download from the OpenJDK site. It is typically not included in (Linux) JDK packages, but is available as a separate package.

Section B.2: Java SE Versions

Java SE Version History

The following table provides the timeline for the significant major versions of the Java SE platform.

Java SE Version1	Code Name	End-of-life (free2)	Release Date
Java SE 10 (Early Access)	*None*	future	2018-03-20 (estimated)
Java SE 9	*None*	future	2017-07-27)
Java SE 8	*None*	future	2014-03-18
Java SE 7	Dolphin	2015-04-14	2011-07-28
Java SE 6	Mustang	2013-04-16	2006-12-23
Java SE 5	Tiger	2009-11-04	2004-10-04
Java SE 1.4.2	Mantis	prior to 2009-11-04	2003-06-26
Java SE 1.4.1	Hopper / Grasshopper	prior to 2009-11-04	2002-09-16
Java SE 1.4	Merlin	prior to 2009-11-04	2002-02-06
Java SE 1.3.1	Ladybird	prior to 2009-11-04	2001-05-17
Java SE 1.3	Kestrel	prior to 2009-11-04	2000-05-08
Java SE 1.2	Playground	prior to 2009-11-04	1998-12-08
Java SE 1.1	Sparkler	prior to 2009-11-04	1997-02-19
Java SE 1.0	Oak	prior to 2009-11-04	1996-01-21

Footnotes:

1. The links are to online copies of the respective releases documentation on Oracle's website. The documentation for many older releases no longer online, though it typically can be downloaded from the Oracle Java Archives.
2. Most historical versions of Java SE have passed their official "end of life" dates. When a Java version passes this milestone, Oracle stop providing free updates for it. Updates are still available to customers with support contracts.

Source:

- JDK release dates by Roedy Green of Canadian Mind Products

Java SE Version Highlights

Java SE Version	Highlights
Java SE 8	Lambda expressions and MapReduce-inspired Streams. The Nashorn Javascript engine. Annotations on types and repeating annotations. Unsigned arithmetic extensions. New Date and Time APIs. Statically linked JNI libraries. JavaFX launcher. Removal of PermGen.
Java SE 7	String switches, *try-with-resource*, the diamond (`<>`), numeric literal enhancements and exception handling / rethrowing improvements. Concurrency library enhancements. Enhanced support for native file systems. Timsort. ECC crypto algorithms. Improved 2D graphics (GPU) support. Pluggable annotations.
Java SE 6	Significant performance enhancements to JVM platform and Swing. Scripting language API and Mozilla Rhino Javascript engine. JDBC 4.0. Compiler API. JAXB 2.0. Web Services support (JAX-WS)

Java SE 5	Generics, annotations, auto-boxing, `enum` classes, varargs, enhanced `for` loops and static imports. Specification of the Java Memory Model. Swing and RMI enhancements. Addition of `java.util.concurrent.*` package and `Scanner`.
Java SE 1.4	The `assert` keyword. Regular expression classes. Exception chaining. NIO APIs - non-blocking I/O, `Buffer` and `Channel`. `java.util.logging.*` API. Image I/O API. Integrated XML and XSLT (JAXP). Integrated security and cryptography (JCE, JSSE, JAAS). Integrated Java Web Start. Preferences API.
Java SE 1.3	HotSpot JVM included. CORBA / RMI integration. Java Naming and Directory Interface (JNDI). Debugger framework (JPDA). JavaSound API. Proxy API.
Java SE 1.2	The `strictfp` keyword. Swing APIs. The Java plugin (for web browsers). CORBA interoperability. Collections framework.
Java SE 1.1	Inner classes. Reflection. JDBC. RMI. Unicode / character streams. Internationalization support. Overhaul of AWT event model. JavaBeans.

Source:

- Wikipedia: Java version history

Section B.3: Differences between Java EE, Java SE, Java ME and JavaFX

Java technology is both a programming language and a platform. The Java programming language is a high-level object-oriented language that has a particular syntax and style. A Java platform is a particular environment in which Java programming language applications run.

There are several Java platforms. Many developers, even long-time Java programming language developers, do not understand how the different platforms relate to each other.

The Java Programming Language Platforms

There are four platforms of the Java programming language:

- Java Platform, Standard Edition (Java SE)
- Java Platform, Enterprise Edition (Java EE)
- Java Platform, Micro Edition (Java ME)
- Java FX

All Java platforms consist of a Java Virtual Machine (VM) and an application programming interface (API). The Java Virtual Machine is a program, for a particular hardware and software platform, that runs Java technology applications. An API is a collection of software components that you can use to create other software components or applications. Each Java platform provides a virtual machine and an API, and this allows applications written for that platform to run on any compatible system with all the advantages of the Java programming language: platform-independence, power, stability, ease-of-development, and security.

Java SE

When most people think of the Java programming language, they think of the Java SE API. Java SE's API provides the core functionality of the Java programming language. It defines everything from the basic types and objects of the Java programming language to high-level classes that are used for networking, security, database access, graphical user interface (GUI) development, and XML parsing.

In addition to the core API, the Java SE platform consists of a virtual machine, development tools, deployment technologies, and other class libraries and toolkits commonly used in Java technology applications.

Java EE

The Java EE platform is built on top of the Java SE platform. The Java EE platform provides an API and runtime environment for developing and running large-scale, multi-tiered, scalable, reliable, and secure network applications.

Java ME

The Java ME platform provides an API and a small-footprint virtual machine for running Java programming language applications on small devices, like mobile phones. The API is a subset of the Java SE API, along with special class libraries useful for small device application development. Java ME applications are often clients of Java EE platform services.

Java FX

Java FX technology is a platform for creating rich internet applications written in Java FX ScriptTM. Java FX Script is a statically-typed declarative language that is compiled to Java technology bytecode, which can then be run on a Java VM. Applications written for the Java FX platform can include and link to Java programming language classes, and may be clients of Java EE platform services.

- Taken from the Oracle documentation

Appendix C: The Classpath

The classpath lists places where the Java runtime should look for classes and resources. The classpath is also used by the Java compiler to find previously compiled and external dependencies.

Section C.1: Different ways to specify the classpath

There are three ways to set the classpath.

1. It can be set using the `CLASSPATH` environment variable :

   ```
   set CLASSPATH=...          # Windows and csh
   export CLASSPATH=...       # Unix ksh/bash
   ```

2. It can be set on the command line as follows

   ```
   java -classpath ...
   javac -classpath ...
   ```

   Note that the `-classpath` (or `-cp`) option takes precedence over the `CLASSPATH` environment variable.

3. The classpath for an executable JAR file is specified using the `Class-Path` element in `MANIFEST.MF`:

   ```
   Class-Path: jar1-name jar2-name directory-name/jar3-name
   ```

   Note that this only applies when the JAR file is executed like this:

   ```
   java -jar some.jar ...
   ```

   In this mode of execution, the `-classpath` option and the CLASSPATH environment variable will be ignored, even if the JAR file has no `Class-Path` element.

If no classpath is specified, then the default classpath is the selected JAR file when using `java -jar`, or the current directory otherwise.

Related:

- https://docs.oracle.com/javase/tutorial/deployment/jar/downman.html
- http://docs.oracle.com/javase/7/docs/technotes/tools/windows/classpath.html

Section C.2: Adding all JARs in a directory to the classpath

If you want to add all the JARs in directory to the classpath, you can do this concisely using classpath wildcard syntax; for example:

```
someFolder/*
```

This tells the JVM to add all JAR and ZIP files in the `someFolder` directory to the classpath. This syntax can be used in a `-cp` argument, a `CLASSPATH` environment variable, or a `Class-Path` attribute in an executable JAR file's manifest file.See *Setting the Class Path: Class Path Wild Cards* for examples and caveats.

Notes:

1. Classpath wildcards were first introduced in Java 6. Earlier versions of Java do not treat "*" as a wildcard.
2. You cannot put other characters before or after the "*"; *e.g. "someFolder/*.jar" is not a wildcard.
3. A wildcard matches only files with the suffix ".jar" or ".JAR". ZIP files are ignored, as are JAR files with a different suffixes.
4. A wildcard matches only JAR files in the directory itself, not in its subdirectories.
5. When a group of JAR files is matched by a wildcard entry, their relative order on the classpath is not specified.

Section C.3: Load a resource from the classpath

It can be useful to load a resource (image, text file, properties, KeyStore, ...) that is packaged inside a JAR. For this purpose, we can use the `Class` and `ClassLoader`s.

Suppose we have the following project structure:

```
program.jar
|
\-com
\-project
|
|-file.txt
\-Test.class
```

And we want to access the contents of `file.txt` from the `Test` class. We can do so by asking the classloader:

```
InputStream is = Test.class.getClassLoader().getResourceAsStream("com/project/file.txt");
```

By using the classloader, we need to specify the fully qualified path of our resource (each package).

Or alternatively, we can ask the Test class object directly

```
InputStream is = Test.class.getResourceAsStream("file.txt");
```

Using the class object, the path is relative to the class itself. Our `Test.class` being in the `com.project` package, the same as `file.txt`, we do not need to specify any path at all.

We can, however, use absolute paths from the class object, like so:

```
is = Test.class.getResourceAsStream("/com/project/file.txt");
```

Section C.4: Classpath path syntax

The classpath is a sequence of entries which are directory pathnames, JAR or ZIP file pathnames, or JAR / ZIP wildcard specifications.

- For a classpath specified on the command line (e.g. `-classpath`) or as an environment variable, the entries must be separated with `;` (semicolon) characters on Windows, or `:` (colon) characters on other platforms (Linux, UNIX, MacOSX and so on).
- For the `Class-Path` element in a JAR file's `MANIFEST.MF`, use a single space to separate the entries.

Sometimes it is necessary to embed a space in a classpath entry

- When the classpath is specified on the command line, it is simply a matter of using the appropriate shell quoting. For example:

```
export CLASSPATH="/home/user/My JAR Files/foo.jar:second.jar"
```

  (The details may depend on the command shell that you use.)

- When the classpath is specified in a JAR file's a "MANIFEST.MF" file, URL encoding must be used.

```
Class-Path: /home/user/My%20JAR%20Files/foo.jar second.jar
```

Section C.5: Dynamic Classpath

Sometimes, just adding all the JARs from a folder isn't enough, for example when you have native code and need to select a subset of JARs. In this case, you need two `main()` methods. The first one builds a classloader and then uses this classloader to call the second `main()`.

Here is an example which selects the correct SWT native JAR for your platform, adds all your application's JARs and then invokes the real `main()` method: Create cross platform Java SWT Application

Section C.6: Mapping classnames to pathnames

The standard Java toolchain (and 3rd-party tools designed to interoperate with them) have specific rules for mapping the names of classes to the pathnames of files and other resources that represent them.

The mappings are as follows

- For classes in the default package, the pathnames are simple filenames.
- For classes in a named package, the package name components map to directories.
- For named nested and inner classes, the filename component is formed by joining the class names with a `$` character.
- For anonymous inner classes, numbers are used in place of names.

This is illustrated in the following table:

Classname	Source pathname	Classfile pathname
`SomeClass`	`SomeClass.java`	`SomeClass.class`
`com.example.SomeClass`	`com/example/SomeClass.java`	`com/example/SomeClass.class`
`SomeClass.Inner`	(in `SomeClass.java`)	`SomeClass$Inner.class`
`SomeClass` anon inner classes	(in `SomeClass.java`)	`SomeClass$1.class`, `SomeClass$2.class`, etc

Section C.7: The bootstrap classpath

The normal Java classloaders look for classes first in the bootstrap classpath, before checking for extensions and the application classpath. By default, the bootstrap classpath consists of the "rt.jar" file and some other important JAR files that are supplied by the JRE installation. These provide all of the classes in the standard Java SE class library, along with various "internal" implementation classes.

Under normal circumstances, you don't need to concern yourself with this. By default, commands like `java`, `javac` and so on will use the appropriate versions of the runtime libraries.

Very occasionally, it is necessary to override the normal behavior of the Java runtime by using an alternative version of a class in the standard libraries. For example, you might encounter a "show stopper" bug in the runtime libraries that you cannot work around by normal means. In such a situation, it is possible to create a JAR file containing the altered class and then add it to the bootstrap classpath which launching the JVM.

The `java` command provides the following `-X` options for modifying the bootstrap classpath:

- `-Xbootclasspath:<path>` replaces the current boot classpath with the path provided.
- `-Xbootclasspath/a:<path>` appends the provided path to the current boot classpath.
- `-Xbootclasspath/p:<path>` prepends the provided path to the current boot classpath.

Note that when use the bootclasspath options to replace or override a Java class (etcetera), you are technically modifying Java. There *may be* licensing implications if you then distribute your code. (Refer to the terms and conditions of the Java Binary License ... and consult a lawyer.)

Section C.8: What the classpath means: how searches work

The purpose of the classpath is to tell a JVM where to find classes and other resources. The meaning of the classpath and the search process are intertwined.

The classpath is a form of search path which specifies a sequence of *locations* to look for resources. In a standard classpath, these places are either, a directory in the host file system, a JAR file or a ZIP file. In each cases, the location is the root of a *namespace* that will be searched.

The standard procedure for searching for a class on the classpath is as follows:

1. Map the class name to a relative classfile pathname `RP`. The mapping for class names to class filenames is described elsewhere.
2. For each entry `E` in the classpath:
    - If the entry is a filesystem directory:
        - Resolve `RP` relative to `E` to give an absolute pathname `AP`.
        - Test if `AP` is a path for an existing file.
        - If yes, load the class from that file
    - If the entry is a JAR or ZIP file:
        - Lookup `RP` in the JAR / ZIP file index.
        - If the corresponding JAR / ZIP file entry exists, load the class from that entry.

The procedure for searching for a resource on the classpath depends on whether the resource path is absolute or relative. For an absolute resource path, the procedure is as above. For a relative resource path resolved using `Class.getResource` or `Class.getResourceAsStream`, the path for the classes package is prepended prior to searching.

(Note these are the procedures implemented by the standard Java classloaders. A custom classloader might perform the search differently.)

Appendix D: Resources (on classpath)

Java allows the retrieval of file-based resources stored inside of a JAR alongside compiled classes. This topic focuses on loading those resources and making them available to your code.

Section D.1: Loading default configuration

To read default configuration properties:

```
package com.example;

public class ExampleApplication {
    private Properties getDefaults() throws IOException {
        Properties defaults = new Properties();

        try (InputStream defaultsStream =
            ExampleApplication.class.getResourceAsStream("config.properties")) {

            defaults.load(defaultsStream);
        }

        return defaults;
    }
}
```

Section D.2: Loading an image from a resource

To load a bundled image:

```
package com.example;

public class ExampleApplication {
    private Image getIcon() throws IOException {
        URL imageURL = ExampleApplication.class.getResource("icon.png");
        return ImageIO.read(imageURL);
    }
}
```

Section D.3: Finding and reading resources using a classloader

Resource loading in Java comprises the following steps:

1. Finding the `Class` or `ClassLoader` that will find the resource.
2. Finding the resource.
3. Obtaining the byte stream for the resource.
4. Reading and processing the byte stream.
5. Closing the byte stream.

The last three steps are typically accomplished by passing the URL to a library method or constructor to load the resource. You will typically use a `getResource` method in this case. It is also possible to read the resource data in application code. You will typically use `getResourceAsStream` in this case.

Absolute and relative resource paths

Resources that can be loaded from the classpath are denoted by a *path*. The syntax of the path is similar to a UNIX / Linux file path. It consists of simple names separated by forward slash (`/`) characters. A *relative path* starts with a name, and an *absolute path* starts with a separator.

As the Classpath examples describe, a JVM's classpath defines a namespace by overlaying the namespaces of the directories and JAR or ZIP files in the classpath. When an absolute path is resolved, it the classloaders interpret the initial `/` as meaning the root of the namespace. By contrast, a relative path *may be* resolved relative to any "folder" in the namespace. The folder used will depend on the object that you use to resolve the path.

Obtaining a Class or Classloader

A resource can be located using either a **`Class`** object or a `ClassLoader` object. A **`Class`** object can resolve relative paths, so you will typically use one of these if you have a (class) relative resource. There are a variety of ways to obtain a **`Class`** object. For example:

- A *class literal* will give you the **`Class`** object for any class that you can name in Java source code; e.g. `String.class` gives you the **`Class`** object for the `String` type.
- The `Object.getClass()` will give you the **`Class`** object for the type od any object; e.g. `"hello".getClass()` is another way to get **`Class`** of the `String` type.
- The **`Class`**`.forName(String)` method will (if necessary) dynamically load a class and return its **`Class`** object; e.g. **`Class`**`.forName("java.lang.String")`.

A `ClassLoader` object is typically obtained by calling `getClassLoader()` on a **`Class`** object. It is also possible to get hold of the JVM's default classloader using the static `ClassLoader.getSystemClassLoader()` method.

The `get` methods

Once you have a **`Class`** or `ClassLoader` instance, you can find a resource, using one of the following methods:

Methods	Description
`ClassLoader.getResource(path)` `ClassLoader.getResources(path)`	Returns a URL which represents the location of the resource with the given path.
`ClassLoader.getResources(path)` **`Class`**`.getResources(path)`	Returns an `Enumeration<URL>` giving the URLs which can be used to locate the `foo.bar` resource; see below.
`ClassLoader.getResourceAsStream(path)` **`Class`**`.getResourceStream(path)`	Returns an `InputStream` from which you can read the contents of the `foo.bar` resource as a sequence of bytes.

Notes:

- The main difference between the `ClassLoader` and **`Class`** versions of the methods is in the way that relative paths are interpreted.
    - The **`Class`** methods resolve a relative path in the "folder" that corresponds to the classes package.
    - The `ClassLoader` methods treat relative paths as if they were absolute; i.e. the resolve them in the "root folder" of the classpath namespace.
- If the requested resource (or resources) cannot be found, the `getResource` and getResourceAsStream`methods` **`return`**`null, and the`getResources`methods` **`return`**` an empty`Enumeration`.
- The URLs returned will be resolvable using `URL.toStream()`. They could be `file:` URLs or other conventional URLs, but if the resource resides in a JAR file, they will be `jar:` URLs that identify the JAR file and a specific resource within it.

- If your code uses a `getResourceAsStream` method (or `URL.toStream()`) to obtain an `InputStream`, it is responsible for closing the stream object. Failure to close the stream could lead to a resource leak.

Section D.4: Loading same-name resource from multiple JARs

Resource with same path and name may exist in more than one JAR file on the classpath. Common cases are resources following a convention or that are part of a packaging specification. Examples for such resources are

- META-INF/MANIFEST.MF
- META-INF/beans.xml (CDI Spec)
- ServiceLoader properties containing implementation providers

To get access to *all* of these resources in different jars, one has to use a ClassLoader, which has a method for this. The returned `Enumeration` can be conveniently converted to a `List` using a Collections function.

```
Enumeration<URL> resEnum = MyClass.class.getClassLoader().getResources("META-INF/MANIFEST.MF");
ArrayList<URL> resources = Collections.list(resEnum);
```