Programming Fundamentals Using Java

Learner Manual

Table of Contents

Welcome to the Academic Institute of Excellence

We pride ourselves on our innovative approach to quality education, excellent service delivery, and a modern outlook to technology. AIE creates, develops, supports, and presents innovative progams to create employable, productive, emotionally intelligent, and skilled learners.

Our learning paths are designed by analysing global skills demands and by providing relevant programmes to meet these demands. Our facilitators are qualified subject matter experts with both practical industry experience and tertiary education experience.

The AIE is a revolutionised family of brands, people, and students. We present our programs in a smarter, more efficient, and cost-effective way by utilising innovation and technology, which results in an expanded reach of our learning interventions. We strive towards excellence and on empowering future generations to become problem solvers, critical thinkers and innovators to create the leaders of tomorrow.

We cultivate excellence.

Our VISION

To deliver demand-driven education, built upon the principal of quality education through innovation and technology.

AIE Academic Institute of Excellence

the cad corporation
CK No: 2002/013213/23

learnfast

GREENSIDE DESIGN CENTER

Getting started with Java Language

Module 1

Module Overview

In this module we will look at the basics of Java Language

Topics covered in this module:

- Creating your first Java Program

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

Take 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/

Take 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.

Take 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
```

Take 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!

Take NOTE!

In order for Java commands (java, javac, etc) to be recognized, you will need to make sure of the following

- 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 Strings 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.

Take 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:

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

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.

Take 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
```

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.

Type Conversion

Module 2

Module Overview

In this module we will look at the Type Conversion

Topics covered in this module:

- Numeric primitive casting
- Basic Numeric Promotion
- Non-Numeric primitive casting
- Object Casting
- Testing if an object can be cast using **instanceof**

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.

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.
}
```

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
```

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)
```

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.
}
```

Getters and Setters

Module 3

Module Overview

In this module we will look at Getters and Setters

Topics covered in this module:

- Using a setter or getter to implement a constraint
- Why use Getters and Setters
- Adding Getters and Setters

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

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.

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-backwardcompatible 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.

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.

```java
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
```

Take NOTE!

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

Reference Data Types

Module 4

Module Overview

In this module we will look at Reference Data Types

Topics covered in this module:

- Dereferencing
- Instantiating a reference type

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.

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.

Java Compiler – ‘Javac’

Module 5

Module Overview

In this module we will look at the Java Compiler - 'javac' command

Topics covered in this module:

- The 'javac' command - getting started
- Compiling for a different version of Java

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
```

Take 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.

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
}
```

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 compiletime 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.

Documenting Java Code

Module 6

Module Overview

In this module we will look at the Documenting Java Code command

Topics covered in this module:

- Building Javadocs from the Command Line
- Class Documentation
- Method Documentation
- Package Documentation
- Links
- Code snippets inside documentation
- Field Documentation
- Inline Code Documentation

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.

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 [docsdirectory] is:

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

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.

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 used as a
summary.
* Instead of @throws, @exception can also be used. optional.
* As with other docs, the documentation before the first period
* @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;
}
```

Package Documentation

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**.

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}
*/
```

You can link to the javadoc of an already imported class using ClassName.

You can also use the fully-qualified name, if the class is not already imported: some.other.ClassName

You can link to members (fields or methods) of a class like so:

ClassName.someMethod()

ClassName.someMethodWithParameters(int, String)

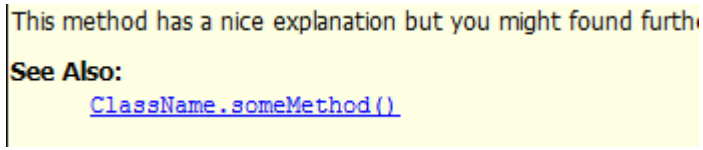
ClassName.someField

someMethodInThisClass() - used to link to members in the current class

You can add a label to a linked javadoc like so: link text

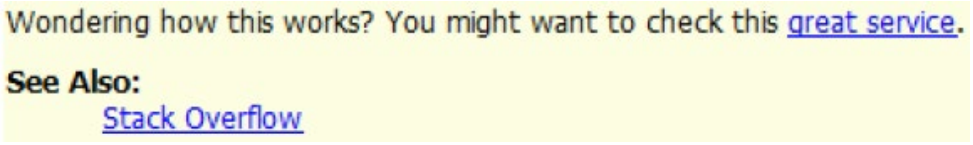
With the @see tag you can add elements to the See also section. Like @param or **@return** the place where they appear is not relevant. The spec says you should write it after **@return**.

```
/**
* 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>
*/
```



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 { }
```

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";
```

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

Command line Argument Processing

Module 7

Module Overview

In this module we will look at the Command line Argument Processing

Topics covered in this module:

- Argument processing using GWT ToolBase
- Processing arguments by hand

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

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 exten
   protected File dir;
   protected int port;
   // getters for dir and port
   ...

   public MyProgramHandler() {
       this.registerHandler(new Arg
            @Override
            public void setDir(File
                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.

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]);
    }
}
```

Take NOTE!

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.

The Java Command – ‘java’ and ‘javaw’

Module 8

Module Overview

In this module we will look at the Java Commands, java’ and ‘javaw

Topics covered in this module:

- Entry point classes
- Troubleshooting the ‘java’ command
- Running a Java application with library dependencies
- Java Options
- Spaces and other special characters in arguments
- Running an executable JAR file
- Running a Java applications via a “main” class

Entry point classes

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

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

Take NOTE!

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

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.

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.)

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".

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.

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.

Take NOTE!

A -cp / -classpath option will be ignored if you use -jar. The application's classpath is determined by the JAR file manifest.

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.)

Literals

Module 9

Module Overview

In this module we will look at Java literals

Topics covered in this module:

- Using underscore to improve readability
- Hexadecimal, Octal and Binary literals
- Boolean literals
- String literals
- The Null literal
- Escape sequences in literals
- Character literals
- Decimal Integer literals
- Floating-point 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, 1, 0.333F, false, 'X' and "Hello world\n".

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)

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
```

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
```

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
```

Take NOTE!

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.

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.

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).

Take NOTE!

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.

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.

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
```

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.

Take NOTE!

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
```

Take NOTE!

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.
```

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:

- Every hexadecimal floating point literal starts with a zero (0) and then an x or X.
- The digits of the number (but not the exponent part!) also include the hexadecimal digits a through f and their uppercase equivalents.
- 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.

Primitive Data Types

Module 10

Module Overview

In this module we will look at the Primitive Data Types in Java

Topics covered in this module:

- The char primitive
- Primitive Types Cheatsheet
- The float primitive
- The int primitive
- Converting Primitives
- Memory consumption of primitives vs. boxed primitives
- The double primitive
- The long primitive
- The boolean primitive
- The byte primitive
- Negative value representation
- The short primitive

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.

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);
}
```

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 -128 to +127	0
Short	16-bit signed	-215 to 215 – 1 -32,768 to +32,767	0
Int	32-bit signed	-231 to 231 – 1 -2,147,483,648 to +2,147,483,647	0
Long	64-bit signed	-263 to 263 – 1 -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0L
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 to 65,535	0

Take NOTE!

1. The Java Language Specification mandates that signed integral types (byte through long) use binary twoscomplement 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).

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.

Take 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.

Take 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
```

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 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
```

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
```

Memory consumption of primitives vs. boxed primitives

Primitive Boxed Type Memory Size of primitive / boxed

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

B 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.

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
```

Take 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
```

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
```

Take 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: ttps://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.

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
```

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
```

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 with value s. The value repesented by those bits is

$x - s * 2^{n-1}$

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 ($2^{n-2} + 2^{n-3} + \ldots + 2^1 + 2^0 = 2^{n-1} - 1$) is subtracted allowing a unique binary representation for each value from -2^{n-1} ($s = 1; x = 0$) to $2^{n-1} - 1$ ($s = 0; x = 2^{n-1} - 1$).

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

$v1 = x1 - s1 * 2^{n-1}$ $v2 = x2 - s2 * 2^{n-1}$

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

Take NOTE!

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)

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
```

Strings

Module 11

Module Overview

In this module we will look at working with Strings. In Java, Strings are immutable, meaning that they cannot be changed.

Topics covered in this module:

- The char primitive
- Primitive Types Cheatsheet
- The float primitive
- The int primitive
- Converting Primitives
- Memory consumption of primitives vs. boxed primitives
- The double primitive
- The long primitive
- The boolean primitive
- The byte primitive
- Negative value representation
- The short primitive

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.

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

Take NOTE!

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").

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

Objects.equals(foo, "baz").

Take 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.compare To 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.

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.

Take 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);
```

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

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:

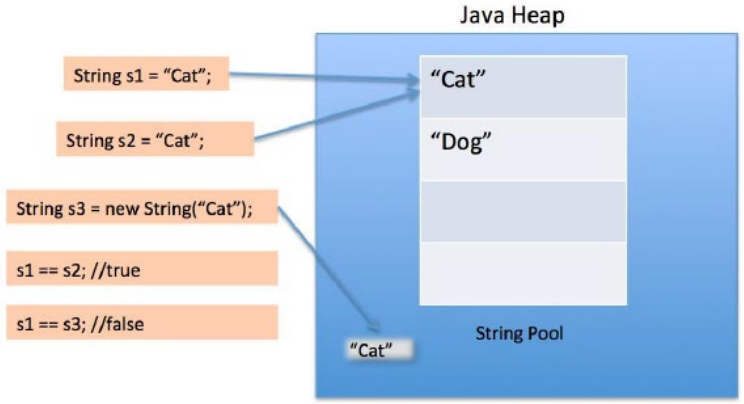
```
class Strings
{
    public static void main (String[] args)
    {
        String a = "alpha";
        String b = "alpha";
        String c = new String("alpha");

        //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.

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.

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.

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 plit(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.

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
```

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}"
```

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");
```

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.

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()
```

Take 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]));
```

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));
```

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.

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()`.

Case insensitive switch

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!

Replacing parts of Strings

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

Take 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

Take 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.
```

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();
```

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.

Take NOTE!

Index n is starting at 0, so the first element is at n=0.

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;
}
```

StringBuffer

Module 12

Module Overview

In this module we will look at introduction to Java StringBuffer class.

Topics covered in this module:

- String Buffer class

Introduction to Java StringBuffer class.

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
 }
}
```

Previous
StringBuffer
StringBuilder
String Tokenizer
Splitting a string into fixed length parts
Date Class
Dates and Time (java.time.*)
Next >>

StringBuilder

Module 13

Module Overview

In this module we will look at the Java StringBuilder class

Topics covered in this module:

- Comparing StringBuffer, StringBuilder, Formatter and StringJoiner
- Repeat a String n times

Module 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.

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.

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() * n2) = 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();
```

Previous
StringBuffer
StringBuilder
String Tokenizer
Splitting a string into fixed length parts
Date Class
Dates and Time (java.time.*)
Next >>

String Tokenizer

Module 14

Module Overview

In this module we will look at the The java.util.StringTokenizer class

Topics covered in this module:

- StringTokenizer Split by space
- StringTokenizer Split by comma

Module 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 pertoken basis.

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

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

Splitting a string into fixed length parts

Module 15

Module Overview

In this module we will look at Splitting a string into fixed length parts

Topics covered in this module:

- Break a string up into substrings all of a known length
- Break a string up into substrings all of variable length

Splitting a string into fixed length parts

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.

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 + "})");
```

Date Class

Module 16

Module Overview

In this module we will look at the Date Class in Java

Topics covered in this module:

- Convert java.util.Date to java.sql.Date
- A basic date output
- Java 8 LocalDate and LocalDateTime objects
- Creating a Specific Date
- Converting Date to a certain String format LocalTime
- Convert formatted string representation of date to Date object
- Creating Date objects
- Comparing Date objects
- Converting String into Date
- Time Zones and java.util.Date

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)

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

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

Previous
StringBuffer
StringBuilder
String Tokenizer
Splitting a string into fixed length parts
Date Class
Dates and Time (java.time.*)
Next >>

```
// 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()));
```

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());
```

Previous
StringBuffer
StringBuilder
String Tokenizer
Splitting a string into fixed length parts
Date Class
Dates and Time (java.time.*)
Next >>

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());
```

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 an 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.

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
```

Take 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.

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.

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;
  }
```

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.

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.

Take NOTE!

Most applications should declare method signatures, fields and variables as LocalDate, not this[ChronoLocalDate] interface.

Previous
StringBuffer
StringBuilder
String Tokenizer
Splitting a string into fixed length parts
Date Class
Dates and Time (java.time.*)
Next >>

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).

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

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();
```

Previous
StringBuffer
StringBuilder
String Tokenizer
Splitting a string into fixed length parts
Date Class
Dates and Time (java.time.*)
Next >>

```
//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
```

Dates and Time (java.time.*)

Module 17

Module Overview

In this module we will look at the Dates and Times in Java

Topics covered in this module:

- Calculate Difference between 2 LocalDates
- Date and time
- Operations on dates and times
- Instant
- Usage of various classes of Date Time API
- Date Time Formatting
- Simple Date Manipulations

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 );
```

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");
```

```
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 time zone information

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");
```

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]"))
```

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
```

Usage of various classes of Date Time API

```
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}
 */
```

Previous
StringBuffer
StringBuilder
String Tokenizer
Splitting a string into fixed length parts
Date Class
Dates and Time (java.time.*)
Next >>

```
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);
}

/**
```

Previous
StringBuffer
StringBuilder
String Tokenizer
Splitting a string into fixed length parts
Date Class
Dates and Time (java.time.*)
Next >>

```
 * 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));
}
```

Previous
StringBuffer
StringBuilder
String Tokenizer
Splitting a string into fixed length parts
Date Class
Dates and Time (java.time.*)
Next >>

```
/**
* 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);

}
}
```

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.

Simple Date Manipulations

Get the current date.

```
LocalDate.now()
```

Get yesterday's date.

```
LocalDate y = LocalDate.now().minusDays(1);
```

Previous
StringBuffer
StringBuilder
String Tokenizer
Splitting a string into fixed length parts
Date Class
Dates and Time (java.time.*)
Next >>

Module 17 – Dates and Time (java.time.*)

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.

LocalTime

Module 18

Module Overview

In this module we will look at the Local Time function in Java

Topics covered in this module:

- Amount of time between two LocalTime
- Intro
- Time Modification
- Time Zones and their time difference

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

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).

```java
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

        // 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);
    }
}
```

Intro to LocalTime

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 timezone.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,
```

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);
```

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)
```

Time Zones and their time difference

```
import java.time.LocalTime;
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");

    }
}
```

BigDecimal

Module 19

Module Overview

In this module we will look at the BigDecimal class in Java

Topics covered in this module:

- Comparing BigDecimals
- Using BigDecimal instead of float
- BigDecimal.valueOf( )
- Mathematical operations with BigDecimal
- Initialization of BigDecimals with value zero, one or ten
- BigDecimal objects are immutable

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.

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
```

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.

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
```

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 BigDecimals 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.

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;
```

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
```

BigInteger

Module 20

Module Overview

In this module we will look at the BigInterger class in Java

Topics covered in this module:

- Initialization
- BigInteger Mathematical Operations Examples
- Comparing BigIntegers
- Binary Logic Operations on BigInteger
- Generating random BigIntegers

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.

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**.

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 12and 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

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

Take 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.
}
```

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

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 2bitCount (exclusive).

This also means that new BigInteger(2147483647, sourceOfRandomness) may return all positive BigIntegers 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:

```
import java.security.SecureRandom;

// somewhere in the code...
new BigInteger(32, new SecureRandom());
```

Take NOTE!

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.

```
new BigInteger(32, new Random() {
    int seed = 0;

    @Override
    protected int next(int bits) {
        seed = ((22695477 * seed) + 1) & 2147483647;
// Values shamelessly stolen from
```

NumberFormat

Module 21

Module Overview

In this module we will look at the different number formats using Locale of java.

Topics covered in this module:

- NumberFormat

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)
```

Bit Manipulation

Module 22

Module Overview

In this module we will look at the Bit Manipulation in Java

Topics covered in this module:

- Checking, setting, clearing, and toggling individual bits. Using long as bit mask
- java.util.BitSet class
- Checking if a number is a power of 2
- Signed vs unsigned shift
- Expressing the power of 2
- Packing / unpacking values as bit fragments

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.

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));
```

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)

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.

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;
}
```

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)
};
```

Arrays

Module 23

Module Overview

In this module we will look at Arrays in Java

Topics covered in this module:

- Creating and Initializing Arrays
- Creating a List from an Array
- Creating an Array from a Collection
- Multidimensional and Jagged Arrays
- ArrayIndexOutOfBoundsException
- Array Covariance
- Arrays to Stream
- Iterating over arrays
- Arrays to a String
- Sorting arrays
- Getting the Length of an Array
- Finding an element in an array
- How do you change the size of an array?
- Converting arrays between primitives and boxed types
- Remove an element from an array
- Comparing arrays for equality
- Copying arrays
- Casting 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.

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

```
// 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 on right).

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:

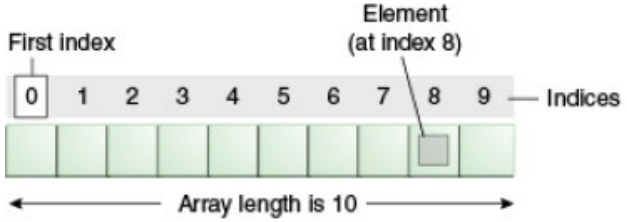
- 0 for primitive numerical types: byte, short, int, long, float, and double.
- '\u0000' (null character) for the char type.
- false for the boolean type.
- null for reference types.

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.
```

Take 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:

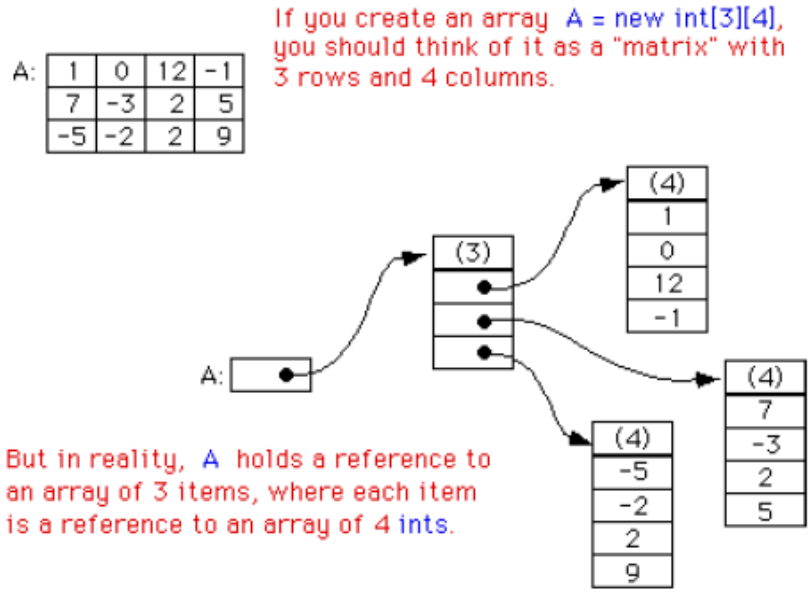
```
int[][] a = { {1, 2}, {3, 4}, {5, 6} };
```

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
```

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

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 };
```

```
// 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

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);
```

Take 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
```

```
(Arrays.asList(stringArray));
```

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.

```
// 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

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.

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.

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

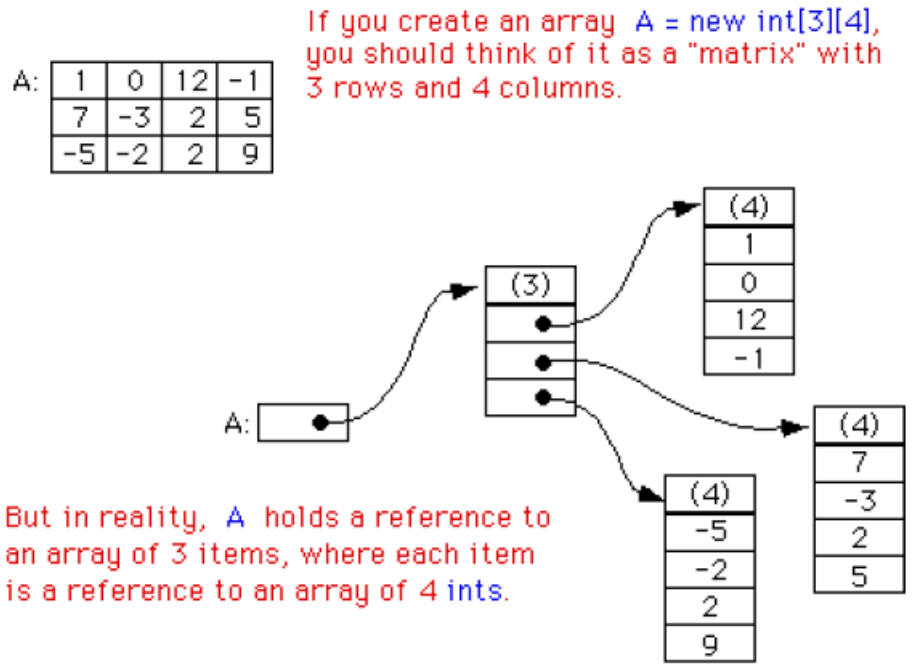


Image source: http://math.hws.edu/eck/cs124/javanotes3/c8/s5.html

Jagged array literal initialization

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}
};
```

Take 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[]
```

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 rrayIndexOutOfBoundsException 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)
```

Take NOTE!

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]);
}
```

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 Integers, and trying to assign a Long element throws a runtime exception.

Take NOTE!

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.
```

Arrays to Stream

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});
```

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.

Take NOTE!

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.

Arrays to a String

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]
```

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]

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
```

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);
```

Take 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);
```

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.)

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
}
```

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]
```

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.

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];
}
```

Take NOTE!

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 Objects in Java, you can use Object.clone().

```
int[] a = {4, 1, 3, 2};
int[] b = Arrays.copyOf(a, a.length); // [4, 1, 3, 2]
```

Take NOTE!

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]
```

Take NOTE!

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( )

Take NOTE!

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]
```

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);
```

Operators

Module 24

Module Overview

In this module we will look at the collections framework in Java

Topics covered in this module:

- The Increment/Decrement Operators (++/--)
- The Conditional Operator (? :)
- The Bitwise and Logical Operators (~, &, |, ^)
- The String Concatenation Operator (+)
- The Arithmetic Operators (+, -, *, /, %)
- The Shift Operators (<<, >> and >>>)
- The Instanceof Operator
- The Assignment Operators (=, +=, -=, *=, /=, %=, <<=, >>= , >>>=, &=, |= and ^=)
- The conditional-and and conditional-or Operators (&& and ||)
- The Relational Operators (<, <=, >, >=)
- The Equality Operators (==, !=)
- The Lambda operator (->)

Operators in Java programming language are special symbols that perform specific operations on one, two, or three operands, and then return a result.

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
```

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))
```

Take NOTE!

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.

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

Take NOTE!

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
```

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 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.

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:

Take NOTE!

The quotient produced for operands n and d is an integer value q whose magnitude is as large as possible while satisfying $|d \cdot q| \leq |n|$. Moreover, q is positive when $|n| \geq |d|$ and n and d have the same sign, but q is negative when $|n| \geq |d|$ 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.

Take 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.

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.

Take NOTE!

Please see notes below related to the Shift Operators:

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

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
```

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.

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.

Take NOTE!

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

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 |.

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.

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.

Take 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-itfalls/16290/using-to-compare-strings.

A similar problem applies to primitive wrapper types; see http://stackoverflow.com/documentation/java/4388/java-itfalls/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"!)

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:

```
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*.

example is included here for completeness. Refer to the Lambda Expressions topic for the full treatment.)

Constructors

Module 25

Module Overview

In this module we will look at the Constructors in Java

Topics covered in this module:

- Default Constructor
- Call parent constructor
- Constructor with Arguments

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.

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 packageprivately 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();
```

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
    }
}
```

Take 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);
    }
}
```

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();
```

Object Class Methods and Constructor

Module 26

Module Overview

In this module we will look at Java class Constructors and Object class methods

Topics covered in this module:

- hashCode() method
- toString() method
- equals() method
- wait() and notify() methods
- getClass() method
- clone() method
- Object constructor
- finalize() method

Module 26 – 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.

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

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});
}
```

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);
}
```

Take 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.

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.

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 ==.

```java
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.

```java
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.

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.

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!
```

getClass() method

The **getClass()** method can be used to find the runtime class type of an object. See the example:

```
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();
```

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);
        }
    }
}
```

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 {

}
```

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.

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.

Credits and References

Module 27

About this Book

AIE has reviewed and modified the content of this book for our training purposes from the Java® Notes for Professionals book.

This book is compiled from Stack Overflow Documentation and the content is written by the beautiful people at Stack Overflow.

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content.

100rabh Chapters 18 and 79
17slim Chapters 28, 40, 72 and 109
1d0m3n30 Chapters 9, 35, 45, 47 and 106
3442 Chapter 23
3751_Creator Chapter 1
4castle Chapters 2 and 57
A Boschman Chapters 42 and 67
A.J. Brown Chapter 11
Aaron Digulla Chapters 47 and 184
Aaron Franke Chapter 56
Aasmund Eldhuset Chapter 46
ABDUL KHALIQ Chapter 90
Abhijeet Chapter 65
Abhishek Jain Chapters 11, 23 and 79
Abubakkar Chapters 11, 23, 57, 67 and 102
acdcjunior Chapters 23 and 57
Ad Infinitum Chapters 23, 24, 33, 43 and 73
Adam Ratzman Chapter 11
Adeel Ansari Chapter 182
Adowrath Chapters 79 and 164
Adrian Krebs Chapters 11, 23, 54, 69, 74 and 111
afzalex Chapter 23
agilob Chapters 11, 23 and 69
agoeb Chapter 54
Aiden Deom Chapter 11
Aimee Borda Chapters 22 and 57
aioobe Chapters 35 and 117
ajablonski Chapter 182
AJNeufeld Chapter 74
akgren_soar Chapter 58
Akhil S K Chapters 1, 66, 69, 117 and 182
alain.janinm Chapters 16, 19 and 138
Alek Mieczkowski Chapters 20, 33 and 78
Alex A Chapter 182
Alex Meiburg Chapters 11 and 47
Alex Shesterov Chapters 11 and 20
Alex T. Chapter 132
Alexandre Grimaud Chapter 181
Alexey Lagunov Chapter 83
alexey semenyuk Chapters 47 and 117
Alexiy Chapter 149
Alon .G. Chapter 43
Alper Fırat Kaya Chapter 77
alphaloop Chapter 144
altomnr Chapters 24 and 182
Amani Kilumanga Chapters 10, 11, 35, 40, 80 and 111
Amit Gujarathi Chapters 12, 29, 30, 31 and 180
Amit Gupta Chapter 73
Anatoly Yakimchuk Chapter 23
Andreas Chapters 40, 99 and 106
Andreas Fester Chapter 35
Andrew Chapters 23, 40, 73 and 79
Andrew Antipov Chapter 88
Andrew Brooke Chapter 74
Andrew Sklyarevsky Chapter 35
Andrii Abramov Chapters 41, 57, 73, 75, 81, 117 and 134
Androbin Chapter 33
Andy Thomas Chapters 11, 80, 85 and 130
Ani Menon Chapters 1, 42, 56 and 182
Anil Chapter 23
ankidaemon Chapter 23
Ankit Katiyar Chapters 73, 122 and 141
Ankur Anand Chapter 1
Anony Chapters 10, 11, 24, 35, 47, 73 and 89
anotherGatsby Chapter 23
Anthony Raymond Chapter 182
Anton Hlinisty Chapter 130
antonio Chapters 1 and 23
anuvab1911 Chapters 42 and 182
ar4ers Chapter 52
Arash Chapter 122
ArcticLord Chapters 103 and 105
arcy Chapters 16 and 68
Arkadiy Chapters 1 and 54
arpit pandey Chapter 21
ArsenArsen Chapter 57
Arthur Chapters 23, 71, 72, 77, 87, 93 and 110
Asaph Chapter 11

AshanPerera Chapter 74
Asiat Chapter 41
assylias Chapters 73 and 126
AstroCB Chapter 23
ata Chapters 28 and 77
Athari Chapters 23 and 57
augray Chapter 42
Aurasphere Chapter 77
Austin Chapter 11
Austin Day Chapter 11
A_Arnold Chapters 11, 16, 18 and 26
Bart Kummel Chapter 57
Batty Chapter 66
bcosynot Chapter 35
ben75 Chapter 97
Bhavik Patel Chapter 69
Bilbo Baggins Chapter 17
Bilesh Ganguly Chapters 10 and 66
Binary Nerd Chapter 28
Blubberguy22 Chapters 25 and 40
bn. Chapter 100
Bob Rivers Chapters 10, 16, 18, 24, 35, 70 and 123
Bobas_Pett Chapter 85
Bohdan Korinnyi Chapter 107
Bohemian Chapters 15, 45 and 54
bowmore Chapters 17, 67 and 126
Božo Stojković Chapter 23
bpoiss Chapters 23 and 57
Brendon Dugan Chapter 129
Brett Kail Chapter 42
Brian Goetz Chapter 73
BrunoDM Chapter 41
Buddy Chapter 74
Burkhard Chapters 1, 6, 7, 11, 23, 54, 66, 69, 77 and 107
bwegs Chapter 23
c.uent Chapter 40
c1phr Chapter 23
Cache Staheli Chapters 11, 13, 23, 47 and 107
CaffeineToCode Chapter 42
Caleb Brinkman Chapters 6 and 74
Caner Balım Chapter 11
carloabelli Chapters 11 and 74
Carlton Chapter 1
CarManuel Chapter 71
Carter Brainerd Chapter 6
Cas Eliëns Chapter 74
Catalina Island Chapter 46
cdm Chapter 70
ced Chapters 24, 25, 26, 41 and 80
charlesreid1 Chapter 85
Charlie H Chapters 1, 23 and 67
Chetya Chapter 126
Chirag Parmar Chapters 24, 26, 78 and 86
Chris Midgley Chapter 40
Christian Chapter 19
Christian Wilkie Chapter 16
Christophe Weis Chapters 84, 103 and 115
Christopher Schneider Chapter 23
Claudia Chapter 40
Claudio Chapter 57
clinomaniac Chapter 24
code11 Chapter 47
Codebender Chapters 24, 41 and 47
coder Chapters 11, 23 and 35
Coffee Ninja Chapter 98
Coffeehouse Coder Chapters 1 and 74
Cold Fire Chapter 23
compuhosny Chapters 151 and 152
Confiqure Chapters 1, 23, 48, 85, 175 and 182
Constantine Chapters 35 and 57
corsiKa Chapter 126
CraftedCart Chapters 5, 74 and 182
cricket_007 Chapter 42
cyberscientist Chapter 11
c□L□s□□□□ Chapter 1
D D Chapter 22
Daniel Chapter 10
Daniel Käfer Chapter 23
Daniel LIn Chapters 48 and 111
Daniel M. Chapters 35, 46, 57, 73, 80, 112 and 177
Daniel Nugent Chapters 81, 114, 130 and 139
Daniel Stradowski Chapters 11, 23, 24, 26 and 57

Daniel Wild Chapter 116
Danilo Guimaraes Chapters 35 and 163
Dariusz Chapters 23, 43, 54, 57, 82, 91, 102, 126, 128 and 138
DarkV1 Chapters 1, 11, 23 and 86
Datagrammar Chapter 95
Dave Ranjan Chapter 48
David Grinberg Chapter 54
David Soroko Chapter 87
DeepCoder Chapter 1
Demon Coldmist Chapter 43
demongolem Chapter 26
desilijic Chapter 176
devguy Chapter 79
devnull69 Chapter 93
DimaSan Chapters 26, 33, 50 and 126
dimo414 Chapters 69, 120 and 131
Display Name Chapters 113 and 170
Dmitriy Kotov Chapter 28
dnup1092 Chapters 10 and 11
Do Nhu Vy Chapters 6 and 10
DonyorM Chapters 54, 57 and 93
dorukayhan Chapters 11, 20, 76, 87, 133 and 134
Draken Chapter 73
Drizzt321 Chapters 43 and 103
Duh Chapter 23
Durgpal Singh Chapters 23 and 28
Dushko Jovanovski Chapters 43, 57 and 130
Dushman Chapters 111, 114 and 173
DVarga Chapters 11, 23, 47, 54, 57 and 79
dwursteisen Chapter 57
Dylan Chapter 41
ebo Chapter 74
Eduard Wirch Chapter 120
Eilit Chapter 23
EJP Chapters 66 and 145
ekaerovets Chapter 97
Elazar Chapter 42
Emil Sierżęga Chapters 23, 42, 115 and 182
Emily Mabrey Chapters 28 and 185
emotionlessbananas Chapters 66 and 120
Emre Bolat Chapters 1 and 23
Enamul Hassan Chapter 56
Eng.Fouad Chapter 23
engineercoding Chapter 33
Enigo Chapters 13, 24, 28, 42 and 77
enrico.bacis Chapters 1, 11, 23 and 57
Enwired Chapter 132
Eran Chapters 1, 23 and 24
erickson Chapter 117
Erik Minarini Chapter 23
Erkan Haspulat Chapter 56
esin88 Chapters 22, 172 and 179
Etki Chapter 23
explv Chapters 16, 23, 54 and 57
F. Stephen Q Chapters 87 and 143
fabian Chapters 10, 13, 40, 43, 48, 74, 80, 85, 91, 103, 117, 122, 136 and 174
faraa Chapter 47
FFY00 Chapter 137
fgb Chapter 13
fikovnik Chapter 67
Fildor Chapters 3, 116, 126 and 127
Filip Smola Chapter 2
FlyingPiMonster Chapters 40, 77 and 122
FMC Chapter 69
foxt7ot Chapter 150
Francesco Menzani Chapters 1, 10 and 97
Freddie Coleman Chapter 126
Friederike Chapter 132
Functino Chapters 1 and 23
futureelite7 Chapter 40
f_puras Chapter 35
Gal Dreiman Chapters 23, 35, 57, 69, 73, 130, 131, 132 and 183
gar Chapters 10 and 73
garg10may Chapter 1
Garreth Golding Chapter 19
Gautam Jose Chapter 182
Gene Marin Chapters 23, 35 and 178
geniushkg Chapter 54
George Bailey Chapters 6 and 41
Gerald Mücke Chapters 6, 22, 77, 130, 136 and 185
GhostCat Chapter 43

Gihan Chathuranga Chapter 86
GingerHead Chapters 1 and 23
giucal Chapter 117
glee8e Chapter 93
gontard Chapter 57
GPI Chapters 24, 28, 63, 73, 81, 108, 126, 135 and 184
GradAsso Chapter 66
granmirupa Chapters 23 and 25
Gray Chapter 11
GreenGiant Chapters 11, 69 and 88
Grexis Chapters 35 and 146
Grzegorz Oledzki Chapter 57
Gubbel Chapter 58
Guilherme Torres Castro Chapter 23
Gustavo Coelho Chapter 23
gwintrob Chapter 67
Gytis Tenovimas Chapters 1 and 23
hamena314 Chapters 11 and 85
Hank D Chapters 57 and 73
Hay Chapter 77
Hazem Farahat Chapter 81
HCarrasko Chapter 116
hellrocker Chapter 126
hexafraction Chapters 47, 69, 73, 126 and 144
hirosht Chapter 54
Holger Chapter 137
HON95 Chapter 11
HTNW Chapters 82 and 130
Hulk Chapter 102
hzpz Chapter 67
Idcmp Chapter 45
iliketocode Chapters 1, 11, 23 and 57
Ilya Chapters 11, 23, 82, 93, 97, 118 and 126
Infuzion Chapter 11
InitializeSahib Chapter 97
inovaovao Chapter 79
intboolstring Chapters 23, 46, 74 and 79
Inzimam Tariq IT Chapter 74
ipsi Chapters 1, 10, 35, 171 and 182
iqbal_cs Chapter 140
Ironcache Chapters 3 and 166
Ivan Vergiliev Chapter 73
J Atkin Chapters 10, 28, 43, 57, 58, 67, 73 and 89
Jérémie Bolduc Chapter 23
J. Pichardo Chapter 150
J.D. Sandifer Chapter 117
Jabir Chapters 11, 16, 24, 28, 69, 72, 86 and 91
Jacob G. Chapter 61
JakeD Chapter 10
James Jensen Chapter 77
james large Chapters 42, 126 and 133
James Oswald Chapter 79
James Taylor Chapters 1 and 23
JamesENL Chapters 42 and 53
Jan Vladimir Mostert Chapters 25, 47 and 79
janos Chapters 25 and 89
Jared Hooper Chapter 35
jatanp Chapter 178
Jatin Balodhi Chapter 5
javac Chapter 11
JAVAC Chapter 81
JavaHopper Chapters 1, 23, 40, 57, 69, 79 and 85
Javant Chapters 23 and 85
Javier Diaz Chapter 28
jayantS Chapter 56
JD9999 Chapters 25 and 59
Jean Chapter 16
Jean Vitor Chapter 1
Jeet Chapter 153
Jeff Coleman Chapter 182
Jeffrey Bosboom Chapters 23, 28, 52 and 54
Jeffrey Lin Chapters 1 and 11
Jens Schauder Chapters 1, 23, 47, 48, 58, 69, 74, 88, 126 and 127
Jeroen Vandevelde Chapter 73
Jeutnarg Chapter 23
Jim Garrison Chapter 23
jitendra varshney Chapter 23
jmattheis Chapter 23
Joe C Chapter 121
Johannes Chapters 23, 35, 79 and 126
John DiFini Chapters 64 and 161
John Fergus Chapter 1

John Nash Chapters 19, 21, 58 and 142
John Slegers Chapter 23
John Starich Chapter 135
johnnyaug Chapter 28
Jojodmo Chapters 10, 11, 23, 40 and 79
Jon Erickson Chapter 57
JonasCz Chapters 11, 69, 74, 78, 86 and 95
Jonathan Chapters 1, 23, 28, 46, 57, 79, 84, 88 and 125
Jonathan Barbero Chapters 96 and 119
Jonathan Lam Chapters 23 and 182
JonK Chapter 88
jopasserat Chapter 25
Jordi Castilla Chapter 11
Jordy Baylac Chapter 77
Jorel Ali Chapter 71
Jorn Vernee Chapters 4, 11, 42, 43, 47, 54, 57, 75, 79, 104, 126 and 135
Joshua Carmody Chapter 2
JStef Chapter 23
Jude Niroshan Chapters 11, 57, 67 and 73
JudgingNotJudging Chapters 13 and 73
juergen d Chapter 74
jwd630 Chapter 177
K'' Chapter 184
k3b Chapter 35
kaartic Chapter 1
Kai Chapters 52, 54, 69 and 79
kajacx Chapter 138
kann Chapter 70
kaotikmynd Chapter 80
Kapep Chapters 11, 43 and 57
KartikKannapur Chapter 28
kasperjj Chapter 97
Kaushal28 Chapters 20, 26 and 86
Kaushik NP Chapter 11
kcoppock Chapter 47
KdgDev Chapter 10
Kelvin Kellner Chapter 159
Ken Y Chapter 122
Kenster Chapters 11, 25, 28 and 63
Kevin DiTraglia Chapter 54
Kevin Raoofi Chapter 73
Kevin Thorne Chapters 6, 23, 40 and 69
Kichiin Chapter 87
kiedysktos Chapter 95
Kineolyan Chapter 22
Kip Chapter 58
KIRAN KUMAR MATAM Chapters 27, 30, 36, 37, 39, 49, 53, 58, 59 and 92
Kirill Sokolov Chapter 89
Kishore Tulsiani Chapter 164
kristyna Chapter 45
Krzysztof Krasoń Chapter 25
kstandell Chapters 11, 42 and 79
KudzieChase Chapter 24
Kuroda Chapter 18
Lachlan Dowding Chapter 86
Lankymart Chapter 56
Laurel Chapters 79, 80 and 86
leaqui Chapter 77
Lernkurve Chapter 111
Li357 Chapter 106
Liju Thomas Chapters 23, 128 and 169
llamositopia Chapter 23
Loris Securo Chapters 19, 23 and 56
Luan Nico Chapters 23, 54, 89 and 103
Lukas Knuth Chapter 102
M M Chapter 14
Maarten Bodewes Chapters 20, 35, 86 and 140
Mac70 Chapter 40
madx Chapters 1, 35 and 103
Makoto Chapters 23, 70 and 74
Makyen Chapter 23
Malav Chapter 11
Malt Chapters 23, 32, 73, 88, 93 and 126
Manish Kothari Chapters 19 and 128
manouti Chapters 109, 156 and 168
Manuel Spigolon Chapter 11
Manuel Vieda Chapter 18
Marc Chapters 1 and 47
Mark Green Chapter 73
Mark Stewart Chapter 5
Mark Yisri Chapters 46 and 71
Maroun Chapter 24
Martin Frank Chapter 24

Marvin Chapters 11 and 23
MasterBlaster Chapters 48, 57 and 87
Matas Vaitkevicius Chapter 23
Matěj Kripner Chapter 126
mateuscb Chapter 77
Matsemann Chapter 88
Matt Chapters 1, 23 and 47
Matt Clark Chapters 11, 16, 58 and 86
matt freake Chapters 20, 43 and 52
Matthew Trout Chapter 73
Matthias Braun Chapter 11
Matthieu Chapter 94
Maxim Kreschishin Chapter 23
Maxim Plevako Chapters 11 and 23
Maximillian Laumeister Chapter 23
mayha Chapter 11
mayojava Chapters 10 and 85
MBorsch Chapter 10
Md. Nasir Uddin Bhuiyan Chapters 20 and 126
Michael Chapters 11 and 58
Michael Myers Chapters 35 and 103
Michael Piefel Chapters 23, 45 and 126
Michael von Wenckstern Chapter 7
Michael Wiles Chapter 67
michaelbahr Chapters 28, 69 and 87
Michał Rybak Chapter 135
Mick Mnemonic Chapter 35
MikeW Chapter 79
Miles Chapters 11, 16 and 17
Miljen Mikic Chapters 23, 42, 69 and 122
Mimouni Chapter 23
Mimyck Chapter 23
Mine_Stone Chapter 93
Minhas Kamal Chapter 23
Miroslav Bradic Chapter 88
Mitch Talmadge Chapters 1 and 23
mnoronha Chapter 1
Mo.Ashfaq Chapter 28
Mohamed Fadhl Chapter 23
Mrunal Pagnis Chapters 107 and 116
Mshnik Chapters 47, 50, 54 and 81
mszymborski Chapter 19
Muhammed Refaat Chapters 23 and 54
Mukund Chapter 1
Murat K. Chapter 126
Mureinik Chapters 57 and 59
Muto Chapter 57
Mykola Yashchenko Chapter 44
Myridium Chapter 69
NageN Chapters 23, 35, 40, 42, 46, 55, 95, 111 and 122
Nagesh Lakinepally Chapter 6
NamshubWriter Chapters 16 and 88
Naresh Kumar Chapter 57
Nathaniel Ford Chapter 28
NatNgs Chapters 47 and 126
Nayuki Chapters 22, 23, 42 and 89
ncmathsadist Chapter 73
Nef10 Chapter 35
neohope Chapter 145
nhahtdh Chapter 80
nicael Chapter 23
Nicholas J Panella Chapter 107
Nick Donnelly Chapter 2
nickguletskii Chapter 126
Nicktar Chapters 16, 42 and 85
Nikhil R Chapters 158 and 160
Nikita Kurtin Chapters 69 and 107
Niklas Rosencrantz Chapter 162
NikolaB Chapter 11
Nishant123 Chapter 16
nishizawa23 Chapter 148
Nithanim Chapters 1, 128 and 182
niyasc Chapter 23
nobeh Chapter 73
Nolequen Chapters 35, 43 and 81
noscreenname Chapters 66 and 127
Nufail Chapter 20
Nuri Tasdemir Chapters 1, 11, 23, 40 and 57
nyarasha Chapter 1
Ocracoke Chapter 23
OldCurmudgeon Chapter 35
OldMcDonald Chapter 54
Oleg Sklyar Chapters 24, 25, 47 and 54
OliPro007 Chapter 35
Omar Ayala Chapter 114

Onur Chapters 11, 23, 47, 66 and 122

orccrusher99 Chapter 23

Ordiel Chapter 157

Ortomala Lokni Chapters 40, 43, 47 and 57

ostrichofevil Chapter 155

OverCoder Chapters 35 and 177

P.J.Meisch Chapters 11, 13, 35, 69, 100, 115 and 130

Pablo Chapter 24

Pace Chapter 42

padippist Chapter 177

paisanco Chapter 47

Panda Chapter 23

ParkerHalo Chapters 9, 10, 40 and 46

Paul Bellora Chapter 47

Pavneet_Singh Chapter 1

Pawan Chapters 57 and 111

Paweł Albecki Chapters 23, 24, 35 and 47

PcAF Chapter 47

Peter Rader Chapter 132

peterh Chapter 72

Petter Friberg Chapters 3, 11, 24, 35, 42, 47, 56, 57, 69, 73, 82, 103, 117 and 162

phant0m Chapter 11

phatfingers Chapter 28

philnate Chapters 47, 52, 73, 74 and 127

Pirate_Jack Chapter 57

Piyush Baderia Chapters 11, 89 and 132

PizzaFrog Chapter 6

Polostor Chapter 24

Pops Chapter 1

Powerlord Chapters 24 and 63

ppeterka Chapters 11, 16, 23, 33, 57, 69, 70, 80, 91, 105, 107, 122 and 135

Prasad Reddy Chapter 24

Prem Singh Bist Chapter 107

Přemysl Šťastný Chapter 11

Pseudonym Patel Chapter 148

PSN Chapter 23

PSo Chapters 11 and 86

Pujan Srivastava Chapter 73

QoP Chapters 11 and 23

qxz Chapter 40

Radek Postołowicz Chapters 10 and 69

Radiodef Chapters 23 and 24

Radouane ROUFID Chapters 1, 11, 23, 35, 47, 57, 69, 73 and 182

Rafael Pacheco Chapter 114

rahul tyagi Chapter 40

rajadilipkolli Chapter 24

Rajesh Chapter 23

Rakitić Chapters 57 and 182

rakwaht Chapter 170

Ralf Kleberhoff Chapters 16 and 41

Ram Chapters 1, 16, 23, 28, 42, 43, 48, 70, 74, 78, 80 and 91

RamenChef Chapters 1, 11, 23, 33, 40, 69, 73, 79, 86, 106, 126, 151 and 165

RAnders00 Chapters 10, 11, 60 and 77

Ravindra babu Chapters 54, 103, 111, 126 and 127

Ravindra HV Chapters 52 and 132

Raviteja Chapter 85

ravthiru Chapters 28, 57 and 82

rd22 Chapters 24, 33, 35, 47 and 126

rdonuk Chapters 24 and 69

Rednivrug Chapter 22

Redterd Chapter 78

Rens van der Heijden Chapter 116

reto Chapter 57

Reut Sharabani Chapters 1, 23, 40 and 57

richersoon Chapter 52

RobAu Chapters 57, 69 and 77

Robert Columbia Chapters 10, 23 and 42

Robin Chapter 75

Rocherlee Chapter 11

Rogério Chapter 47

rokonoid Chapters 66, 77 and 87

rolve Chapters 23, 47, 56 and 73

ronnyfm Chapter 182

Ronon Dex Chapter 35

RudolphEst Chapter 132

Ruslan Bes Chapter 20

RutledgePaulV Chapter 47

Ryan Cocuzzo Chapter 48

Ryan Hilbert Chapter 22
saagarjha Chapters 56 and 89
SachinSarawgi Chapters 1 and 131
Saclyr Barlonium Chapter 73
Sadiq Ali Chapter 71
Saif Chapter 80
Samk Chapters 33 and 35
Sanandrea Chapter 182
Sandeep Chatterjee Chapter 182
sanjaykumar81 Chapter 117
Santhosh Ramanan Chapters 74 and 117
sargue Chapters 6 and 50
Saša Šijak Chapter 69
Saurabh Chapter 23
SaWo Chapter 150
scorpp Chapter 131
screab Chapters 130 and 183
Sergii Bishyr Chapters 23, 57 and 73
sevenforce Chapters 11, 23, 57 and 74
Shaan Chapter 86
Shettyh Chapter 127
shibli049 Chapter 142
ShivBuyya Chapters 11, 40 and 77
shmosel Chapters 23, 35, 43, 57, 67, 88 and 106
Shoe Chapters 11, 23 and 57
Siguza Chapters 1 and 47
Simon Chapter 17
Simulant Chapters 10 and 79
Siva Sainath Reddy Bandi Chapter 66
SjB Chapter 24
skia.heliou Chapter 16
Sky Chapter 11
Skylar Sutton Chapter 73
smichel Chapter 101
Smit Chapter 148
solidcell Chapters 11 and 23
someoneigna Chapter 79
Somnath Musib Chapters 26 and 85
Spina Chapters 35 and 57
SRJ Chapter 57
stackptr Chapters 1, 23 and 57
Stefan Dollase Chapter 57
stefanobaghino Chapter 88
steffen Chapter 135
Stephan Chapter 124
Stephen C Chapters 1, 5, 7, 8, 9, 10, 11, 13, 23, 25, 28, 33, 40, 42, 43, 45, 47, 50, 51, 54, 57, 59, 67, 69, 73, 77, 79, 81, 82, 85, 86, 88, 89, 95, 102, 103, 106, 126, 127, 130, 131, 132, 133, 134, 135, 137, 139, 146, 147, 148, 169, 177, 178, 182, 183, 184 and 185
Stephen Leppik Chapters 1, 23, 47, 69 and 73
Steve K Chapter 57
still_learning Chapters 54, 69, 77, 86 and 107
Stoyan Dekov Chapter 107
Sudhir Singh Chapters 54 and 126
Sugan Chapters 38 and 57
Sujith Niraikulathan Chapter 3
Suketu Patel Chapter 34
Suminda Sirinath S. Dharmasena Chapters 127, 139, 146 and 147
sumit Chapter 85
svsav Chapter 93
GoalKicker.com – Java® Notes for Professionals 956
Snađoшfa□ Chapters 1, 5 and 182
taer Chapter 128
tainy Chapter 47
Tarun Maganti Chapters 17 and 35
TDG Chapter 11
thatguy Chapter 22
The Guy with The Hat Chapter 167
TheLostMind Chapter 11
ThePhantomGamer Chapters 9 and 11
Thisaru Guruge Chapters 3, 25 and 55
Thomas Chapters 23 and 47
Thomas Fritsch Chapter 91
Thomas Gerot Chapters 1 and 79
ThunderStruct Chapter 23
Tim Chapter 107
TMN Chapter 10
TNT Chapters 42 and 162
Tobias Friedinger Chapter 77
Tomasz Bawor Chapter 104

tonirush Chapters 40, 130, 162 and 182
Tony Chapter 154
Tony BenBrahim Chapters 11 and 103
Torsten Chapter 35
Tot Zam Chapters 80, 93 and 162
tpunt Chapters 23 and 57
trashgod Chapter 184
Travis J Chapter 23
Tripta Kiroula Chapter 69
Tunaki Chapters 23, 57 and 73
TuringTux Chapters 76 and 100
Tyler Zika Chapter 48
tynn Chapters 41 and 50
Un3qual Chapter 23
Unihedron Chapters 23, 57, 67, 73, 74, 80 and 89
Universal Electricity Chapters 6 and 103
Uri Agassi Chapters 74 and 167
user1121883 Chapter 102
user1133275 Chapters 1 and 57
user140547 Chapters 67 and 125
user1803551 Chapters 10, 98 and 101
user187470 Chapter 79
user2296600 Chapter 63
user2314737 Chapter 10
user2683146 Chapters 57 and 67
user3105453 Chapters 88, 131 and 132
user6653173 Chapter 23
Uux Chapter 144
uzaif Chapters 1 and 23
vallismortis Chapter 91
Vasilis Vasilatos Chapters 25 and 66
Vasiliy Vlasov Chapters 24, 52 and 68
VatsalSura Chapters 75 and 97
Veedrac Chapter 10
GoalKicker.com – Java® Notes for Professionals 957
Ven Chapter 23
VGR Chapters 11, 72, 114 and 185
Viacheslav Vedenin Chapters 28 and 62
Victor G. Chapters 23 and 35
victorantunes Chapter 104
Vin Chapter 1
Vince Emigh Chapter 79
vincentvanjoe Chapter 73
Vinod Kumar Kashyap Chapter 16
Vivek Anoop Chapter 18
Vlad Chapters 47 and 126
Vladimir Vagaytsev Chapter 89
Vogel612 Chapters 8, 20, 23, 28, 41, 43, 58, 80, 104, 127 and 162
vorburger Chapter 123
vsminkov Chapter 107
vsnyc Chapter 57
Vucko Chapter 54
vvtx Chapter 11
webo80 Chapters 24 and 73
WillShackleford Chapters 102 and 164
Wilson Chapters 1, 11, 23, 47, 57 and 69
Wolfgang Chapter 73
xploreraj Chapters 24, 73 and 88
xTrollxDudex Chapters 126 and 139
xwoker Chapters 19 and 74
yitzih Chapter 123
yiwei Chapter 69
Yogesh Chapter 73
Yohanes Khosiawan 许先
汉Chapter 4
yuku Chapters 11 and 23
Yury Fedorov Chapters 23 and 107
Zachary David Saunders Chapter 1
Ze Rubeus Chapters 57 and 182
Zircon Chapters 79 and 134
ĦěþĐëĐ Chapters 78, 120 and 152
Łukasz Piaszczyk Chapter 19
ΦXocę 웃 Перейра ツChapters 11, 17, 21, 23, 25, 43, 78 and 103
□□R□□□□N Chapters 1 and 25