

Chapter 5 - Introduction to Java

Introduction to Java

“There are only two kinds of languages: the ones people complain about and the ones nobody uses.”

— Bjarne Stroustrup, *The C++ Programming Language*

Java was initially developed by a team of software engineers at Sun Microsystems (now owned by Oracle Corporation) under the leadership of James Gosling in 1991, when he was investigating the development of a hardware-independent software platform using C++. The aim became to develop an alternative to C++, to implement control systems for consumer electronic devices.

C++ was found to be unsatisfactory for this task in many ways, and was dropped in favour of a new language called Oak. This new language consequently renamed Java was a powerful, yet straightforward language waiting for a new application.

In 1994 the World-Wide Web emerged and the Sun Developers used Java as the basis of a web browser, beginning the Java/HotJava project. The name was actually derived from the name of the programmers favourite coffee, during a brainstorming session. HotJava is a WWW browser developed by Sun to demonstrate the power of the Java programming language. Java was perfectly suited for use in WWW applications as the program code is compact, platform neutral, and could be used to generate compact programs called applets that could be embedded in web pages.

In late 1995, Java (beta 2) was released along with the announcement of JavaScript by Sun Microsystems and Netscape Corporation. Support continued, and in late 1995 both Microsoft and IBM requested licensing rights from Sun. In early 1996, Java 1.0 along with JavaScript were officially released on the Internet.

The Java Life-cycle

As you are aware, computer programs are simply lists of instructions to be carried out by the six different logical units of a computer:

- **Input Unit** receives data from the input devices such as the keyboard, mouse and any other peripherals
- **Memory Unit** the primary memory unit (Random Access Memory(RAM)) provides fast access storage of computer programs, data from the input devices and data to be sent to the output devices.
- **Arithmetic and Logic Unit (ALU)** performs the arithmetic calculations on data in memory, such as addition, subtraction, multiplication, division and comparison.
- **The Central Processing Unit (CPU)** manages the other units by sending messages to the input unit to read data into the memory unit, informs the ALU which data to operate on, etc.
- **Storage Unit** stores and reads data and programs in long-term storage (e.g. harddisk drive) to be used at a later time.

- **Output Unit** sends information from the computer to make it available outside of the computer, e.g. printer, network device etc.

Computer programmers write programs to interact with these logical units, either in a form that is directly comprehended by the computer or is comprehended after some form of translation step. Code that is directly understood by a computer is called **machine code**. This language is dependent on the exact type of machine that you are working on, i.e. it will differ if you are working on an Intel PC, or an Apple Macintosh. This code is the lowest level code where an operation to load a number into a particular address could look like: 10110 1100 1011 1001 0000. As it is virtually impossible for humans to write programs in binary code (or decimal/hexadecimal) a higher level language is required.

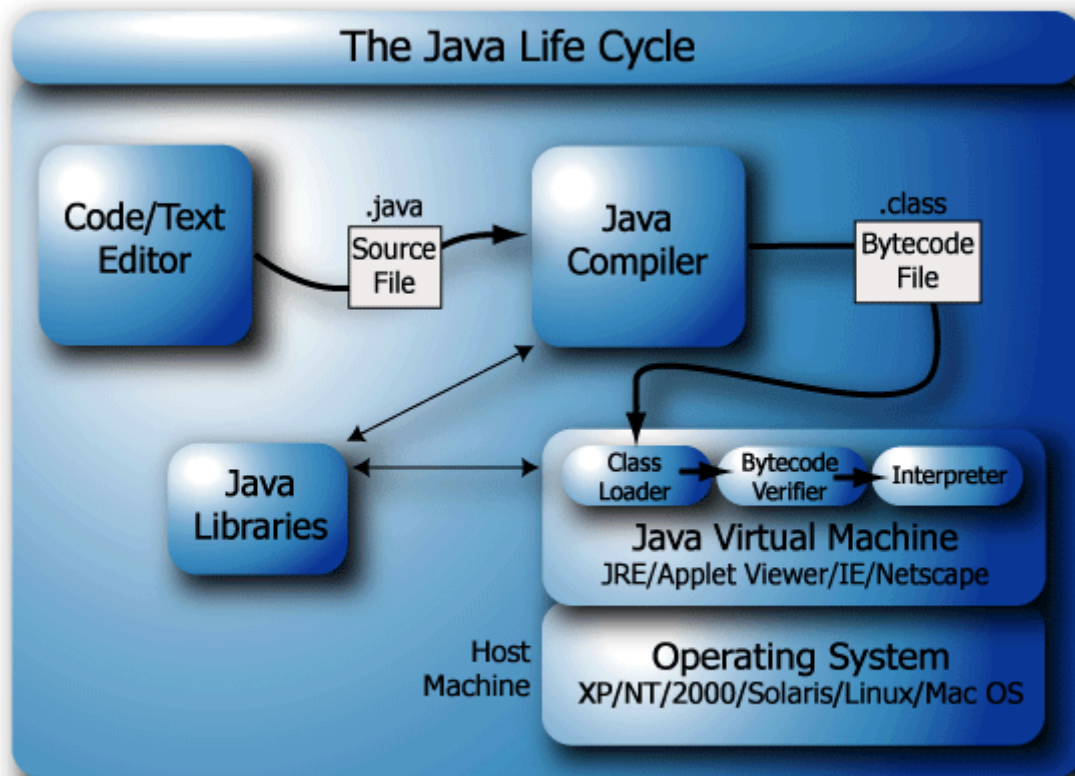
Assembly language improves on this situation as we use sequences of mnemonics to describe the operations that we wish to carry out. For example, `MOV AX,[VarX]` would load the accumulator with the value contained within the variable `VarX`. This code is much clearer to humans, but it still must be translated into machine code so that the computer can understand it.

At the level of assembly language it is possible to write complex computer programs, but they must still be written as low-level instructions. **High-Level languages** were developed to allow a single statement to carry out many tasks. Translator programs called **compilers** then convert the high-level languages into machine code. C, C++ and Java are all examples of high-level languages. Large programs can take significant time to compile from the high-level language form into the low-level machine code form. An alternative to this is to use **interpreters**; programs that execute high-level code directly by translating instructions on demand. These programs do not require compilation time, but the interpreted programs execute much more slowly.

Java programs exist in the form of compiled **bytecode**, that is similar to machine code, except that the platform target is the Java Virtual Machine (JVM). A JVM resides within every Java compatible WWW browser and is indeed stand-alone with the Java Run-time Environment (JRE).

A JVM is, in effect, a bytecode interpreting machine running on a hardware machine. This interpreting stage has an overhead and slows the program execution performance of Java applications. Java bytecode is extremely compact, allowing it to be easily delivered over a network. In theory, the JVM in each Web browser is built to the same specification, requiring that one version of source code should be compatible with many platforms, provided that a Java enabled Web browser exists for that platform. In reality, not all Web browsers implement the exact same JVM specification, introducing minor inconsistencies when the same Java applet is viewed using several different browsers, on different platforms.

The Java application life cycle can be illustrated as in Figure 5.1, "The Java Life Cycle". We can use any text editor to create the high-level Java text file. This file is saved as a `.java` file on the disk. We then compile this text file using the Java compiler, which results in a `.class` file being created on the disk. The `.class` file contains the bytecodes. The file is then loaded into memory by the class loader. The bytecode verifier confirms that the bytecodes are valid and not hostile. Finally, the JVM reads the bytecodes in memory and translates them into machine code.

Figure 5.1. The Java Life Cycle

Just-In-Time Compilation (Dynamic Translation)

In early versions of Java, when an applet was executed for the first time, the speed at which it executes was disappointing. The reason for this is that the Java applet's intermediate bytecode is interpreted by the Java Virtual Machine rather than compiled to native machine instructions as is the case for C++ as discussed previously.

One solution to this performance problem lies in Just-In-Time (JIT) Compilation (also called Dynamic Translation). The JIT compiler reads the bytecode of the Java applet and converts it into native machine instructions for the intended operating system, just after the applet is loaded from the disk. This can happen on a file-by-file basis or even on a method-by-method basis - hence the name, just-in-time. Once the Java applet is converted into native instructions, the application/applet then runs like it was natively compiled. The JIT compiler can, in certain cases, improve the run-time execution speed of applets by a factor of 5-10 times, while still providing portability through retaining the use of intermediate byte-code. Compiled code is kept in memory until the application terminates. In effect, when using just-in-time compilation Java applications are compiled twice, once when the source code is translated into platform portable bytecodes and again when bytecode is being executed on that platform.

The Java Development Kit (JDK)

Introduction

This year, I recommend that you download JDK 15.

We will cover some of the new features of JDKs such as, Metadata, Generics, Enumerated types and Autoboxing of primitive types. However, we will walk before we can run and cover these at the end of the module.

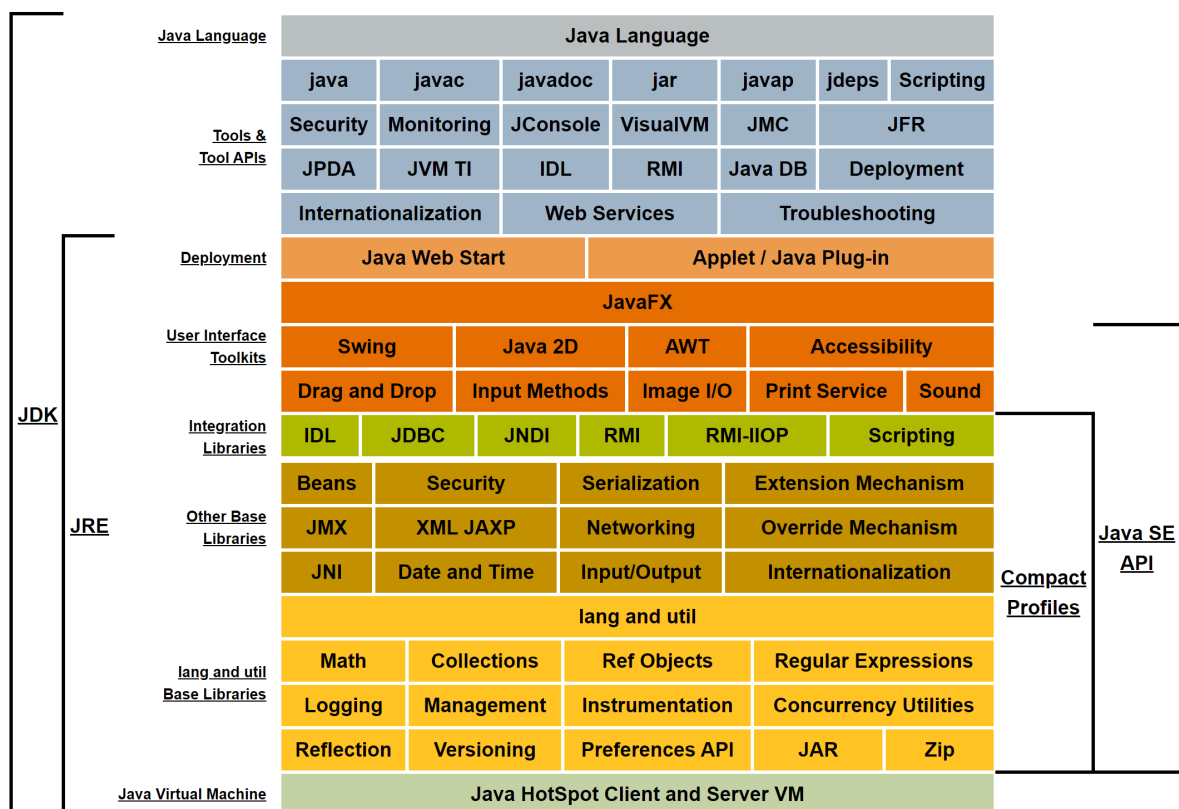
The JDK provides the basic tools of:

- **javac** - The Java programming language compiler.
- **java** - The run-time environment for developed Java applications.
- **appletviewer** - A basic Java enabled browser.
- **javadoc** - Automatic API documentation tool.
- **jdb** - The embedded Java debugging tool.
- **javah** - A native development tool for creating C headers.
- **javap** - A class file disassembler.

It also includes a set of advanced tools for performance analysis, security, Remote Method Invocation (RMI) and internationalisation.

The Java Platform features are illustrated in Figure 5.2, "The Java 2 Platform Overview (Figure from <https://docs.oracle.com/javase/8/docs/>)"¹.

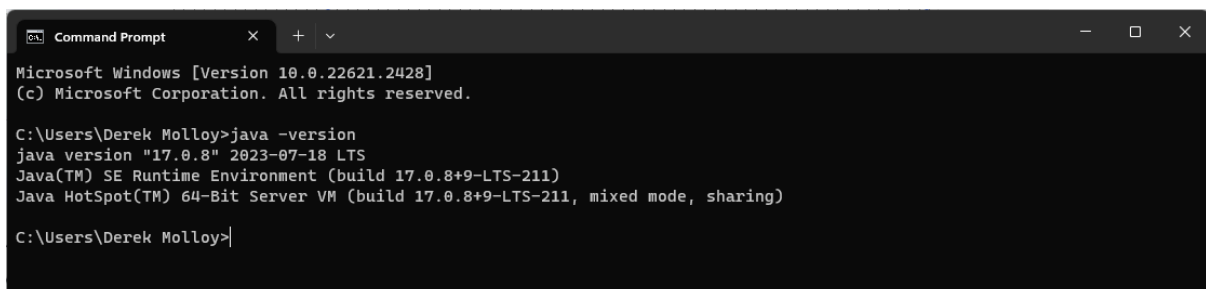
Figure 5.2. The Java Platform Overview (Figure from java.sun.com Version 8)



Installation of the Java Development Kit (JDK)

The latest release of the Java Development Kit (JDK) is Java JDK 21 or 17 -- I recommend Java 17/18 for the 2024/25 academic year. To install this on your home PC, please take the following steps:

- Download the Java Development Kit from <https://www.oracle.com/ie/java/technologies/javase-downloads.html> by following the link that says Oracle JDK -> JDK download -> Windows x64 Installer (or equivalent on macOS/Linux)
- Execute the installation (.exe) file and allow it to complete a full install. Choose the default directory (e.g., c:\Program Files\Java\jdk-17.0.1\)
- Start a Command window and type "java -version". It should appear as below. Once this is working then go to the Further Testing section and then install Eclipse.



```
Microsoft Windows [Version 10.0.22621.2428]
(c) Microsoft Corporation. All rights reserved.

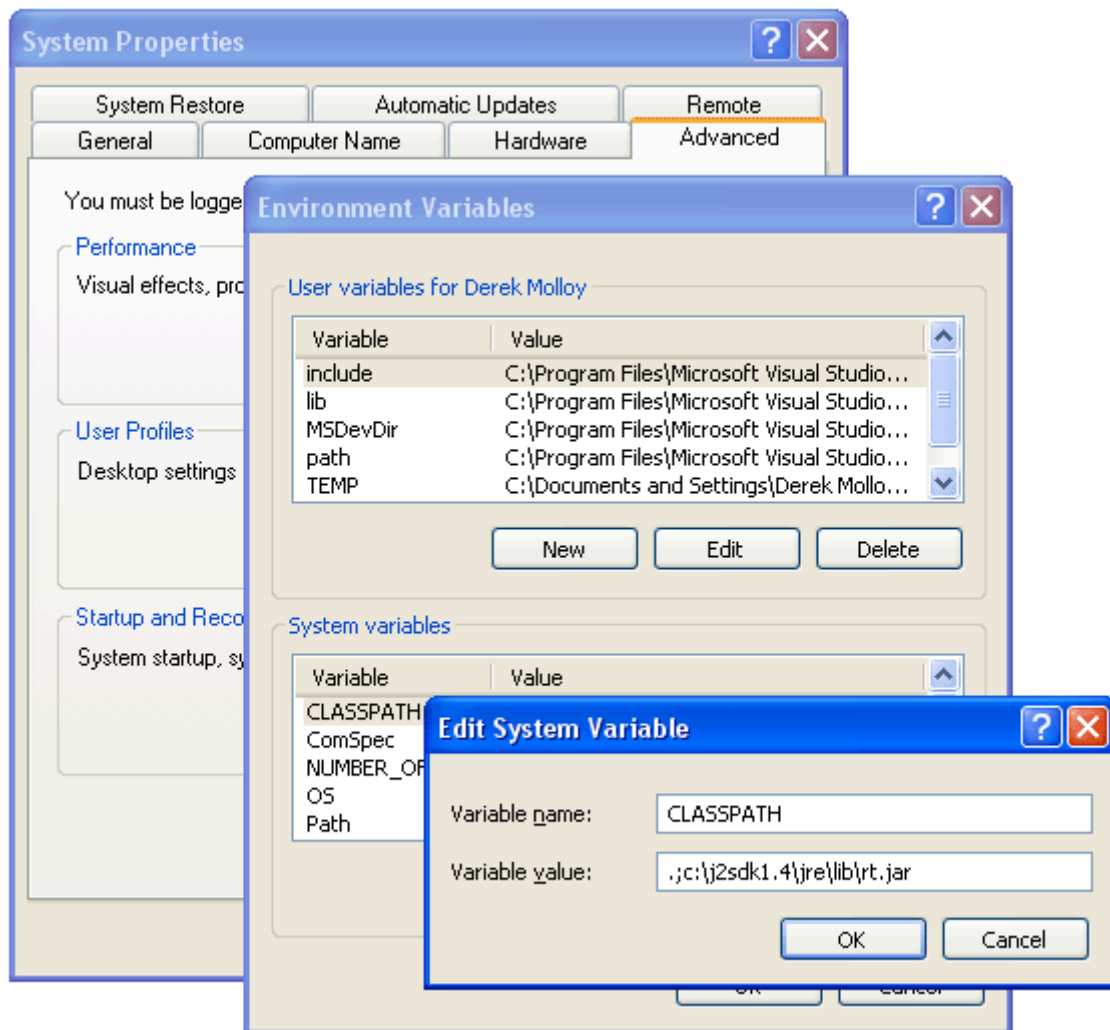
C:\Users\Derek Molloy>java -version
java version "17.0.8" 2023-07-18 LTS
Java(TM) SE Runtime Environment (build 17.0.8+9-LTS-211)
Java HotSpot(TM) 64-Bit Server VM (build 17.0.8+9-LTS-211, mixed mode, sharing)

C:\Users\Derek Molloy>
```

Problems?

- **You should not have to do this step - test first!** Set your **PATH** to include the **bin** directory of the installed folder (e.g. **c:\jdk17\bin** or **C:\Program Files\Java\jdk-17.0.1**). See Figure 5.3, "Setting the Java Environment Variables."
- **You should not have to do this step - test first!** Set a new **CLASSPATH** environment variable to include the file **c:\jdk17\jre\lib\rt.jar** or **C:\Program Files\Java\jdk-17.0.1\jre\lib\rt.jar** and the current directory - so it should have the form **.;c:\jdk17\jre\lib\rt.jar** See Figure 5.3, "Setting the Java Environment Variables.". This is my setting on Windows XP Professional (very similar for Windows 10/11). Do not make this change unless it is absolutely necessary!

Figure 5.3. Setting the Java Environment Variables. (You should not have to do this step unless something goes wrong with the installer.)



Further Testing the JDK

To test the JDK, we will use the Hello World application. Here is the source code:

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

To compile and run this application, perform the following steps:

- Open Notepad, or any other text editor (not MS Word) and paste the code above into the editor.
- Save the file as **HelloWorld.java**. The **HelloWorld** class must be placed in a file of the same name. Note: if you are using notepad, please place inverted commas around the file name (e.g. "HelloWorld.java") or it may be saved as **HelloWorld.java.txt**. Save this file in c:\temp for example.

- Open the command prompt and go to the file location and type **javac -version**. You should see the current version of your installation displayed.
- Compile the (.java) source file, by typing "**javac HelloWorld.java**". If you receive a "command not found" error then your PATH environment variable may be incorrect. Any other errors could mean that your CLASSPATH is incorrect.
- Execute the (.class) bytecode file, by typing "**java HelloWorld**". Note that the file-type extension is omitted and the exact case must be supplied (i.e., capital H and W).

See Figure 5.4, "Testing the JDK." to see this code example working on my PC.

Figure 5.4. Testing the JDK.



```
Command Prompt
C:\Users\Derek Molloy>cd c:\temp
c:\temp>javac -version
javac 17.0.8
c:\temp>javac HelloWorld.java
c:\temp>java HelloWorld
Hello World!
c:\temp>|
```

If you get the same results - you have installed everything correctly - congratulations!

Installing Eclipse

This is very straightforward for Java:

- Go to <https://www.eclipse.org/downloads/> and Download Eclipse x86_64 (or equivalent for a non-Windows platform)
- Choose the Eclipse IDE for Java Developers and choose default values (you should see JDK 17 or the version you just installed appear in the follow-on window as in Figure 5.5.)
- Create a new project: File->new->Java Project called HelloWorld. You can use JavaSE-14 as the execution environment.
- Close the Welcome Screen if it has not disappeared.
- Don't edit the module-info.java file.
- Right-click the src folder and select new->Class. Use een1035for the package name and HelloWorld for the class name. Check the box to add a main() function. Click Finish.
- Add the line `system.out.println("Testing Eclipse");` Press CTRL-S to save and then press the 'play' button to execute (The code is compiled as you type). You should see the same output as in Figure 5.6.

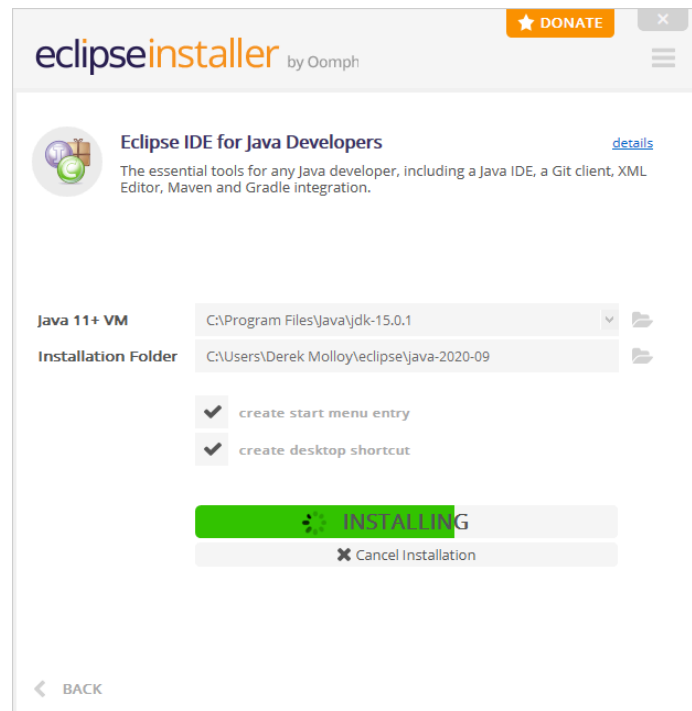


Figure 5.5. The Eclipse Installer

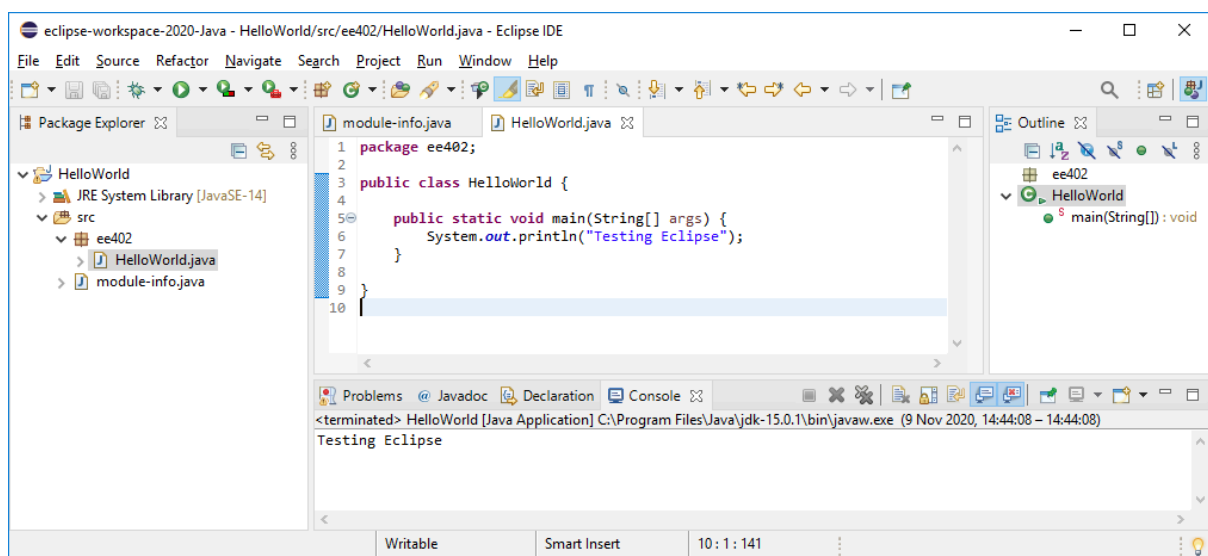


Figure 5.6 Eclipse IDE and Output

The Java Runtime Environment (JRE)

The Java JDK 6 release also included the Java Runtime Environment (JRE), that consists of the JVM, the Java platform core classes, and supporting documentation. The JRE is aimed at developers who wish to distribute their applications with a runtime environment. The JDK is large and licensing also prevents the distribution of the JDK.

New Concepts in Java

Packages

Packages are groups of similar classes that can be, but are not necessarily related to each other through an inheritance structure. Packages are classes organised into directories on a file system.

- Packages are '.' separated words, where the package refers to the directory that the class files are in. For example `java.awt` (the directory `/java/awt/` is a package that stores the classes for creating buttons, text fields, etc.
- A package can also contain more packages in a subfolder. For example, `java.awt.event` contains classes for events, within the awt package.
- Use the `import` keyword to load in packages.
- Multiple classes can be loaded using the `*` character. So, for example, `import java.awt.*;` imports all the classes in the awt package, but please note, it does not include classes in the sub-packages, so, you must explicitly `import java.awt.event.*;`
- In a source file, if no package name is defined on an import e.g. `import SomeClass;` then it is assumed that the class `SomeClass` is in the same directory as the source file.

Packages are very similar to a combination of C++ namespaces and includes, only a lot easier to use.

Java **Modules** were introduced in Java 9. Just like packages they group together classes, but they allow you to define which modules of the Java platform your application requires. This allows you to control the overall size of your deployment (important for embedded devices). The file `module-info.java` describes these relationships.

For example, in `module-info.java` for the HelloWorld project specified earlier you could export `een1035` but perhaps not `een1035.madeupsubpackage`; You can also specify which Java API modules you require.

```
module HelloWorld {  
    exports een1035;  
    requires java.base;  
}
```

Please note that Java modules are not used in the EEN1035 at present as the scale of our developments do not warrant their use.

Garbage Collection

Java does not suffer from memory leaks like C++, due to the addition of an embedded Garbage Collector (GC) and carefully structured reference counting. The GC can be run in three ways:

- The system calls the GC when memory is low.
- The system calls the GC when the CPU is idle.

- The user can call the GC directly, but since the GC runs as a thread in a threaded environment, there is no guarantee that it will run immediately.

super and this predefined variables

In Java we do not have the scope resolution operator `::` that we had in C++. We have the `super` and `this` predefined variables instead:

- `super` - allows us to refer to states and methods of the parent class directly. So, `super.display()` would call the `display()` method, written or inherited by the parent class. If the parent class of the parent class with a `display()` method also had a `display()` method, we do not have access to that method. This `super` predefined variable can also be used in an unusual way to call the parent constructor from within the child constructor, provided it is used as the first line of the child constructor (just like member initialisation lists in Java).
- `this` - allows us to refer to "this object", i.e. to access states or methods for the class that the code is currently within. It allows us to pass a reference to the current object and it also allows us to access the states of the class that are currently out of scope.

So, an example of `super` and `this` would be:

```
public class TestParent {
    protected int y=2;          // example parent state

    public TestParent(int a) { // example parent constructor
        y=a;
    }
}

public class Test extends TestParent { // inheritance
    private int x;

    public Test(int a) {
        super(a); // calls the parent constructor
        x = 5;
    }

    public void someMethod(int x) {
        this.x = x;    // this.x refers to the state of the class
        super.y = 5;   // sets the y state of the parent class to a value of 5.

        someOtherMethod(this); // passes the current object to the
                                // someOtherMethod() method
    }
}
```

Abstract Classes in Java

While abstract classes are not much different to abstract classes in C++, there are some points to note.

Once again, abstract methods are a place-holder that define a set of operations within the class that have not yet been implemented. In Java you can actually set a class to be abstract, even if it has no abstract methods. This would prevent this class from being instantiated. However, if one or more methods in a class are abstract then the class must be defined as abstract. For example:

```
public abstract class Car {
    public abstract void display();
}

public class SportsCar extends Car {
    public void display() {
        System.out.println(" Sports car - etc.");
    }
}
```

Some points to note:

- Not all methods in an abstract class need to be abstract.
- Abstract classes cannot have private abstract methods.
- Abstract classes cannot have static abstract methods.

Interfaces

A Java Interface is a way of grouping methods that describe a particular behaviour. They allow developers to reuse design and they capture similarity, independent of the class hierarchy. We can share both methods and constants using Interfaces, but there can be no states in an interface. Methods in an interface are implicitly public and abstract. Constant states in an interface are implicitly public, static and final.

Interfaces differ from abstract classes in that an interface cannot have an implementation for any method, i.e. all methods are abstract. Classes can implement many interfaces, all unconstrained by a class inheritance structure.

```
interface Demo {
    public static final String demoConst = "hello";
    // public static final can be omitted as is implicit
    void go(); // these methods are public and abstract
    void stop(); // even if public abstract is omitted
}

class SomeClass extends Object implements Demo {
    public void go() {
        // write implementation here
    }
    public void stop() {

```

```

        // write implementation here
    }
}

public class TestApp {
    public static void main(String args[]) {
        SomeClass c = new SomeClass();
        System.out.println(SomeClass.demoConst); // will print out "hello"
    }
}

```

Consider the mouse. We may write many applications that use the mouse, and within these applications, the mouse may be used in many different ways. We can define a common interface for the mouse that each one of these applications must implement. These applications then share a common defined interface, without having to create an IS-A/IS-A-PART-OF relationship.

Interfaces function outside of an inheritance relationship. Since methods and constants are shared through interfaces, and not states, we do not have the difficulties experienced with multiple inheritance when implementing multiple interfaces:

- A class can implement as many interfaces as desired. They are listed one after the other and comma separated.
- An interface should always begin with a capital letter, e.g. `MouseListener`.
- You can define your own interfaces, in the same way that you can define your own classes.

Note: Since the introduction of Java 8. A java interface can have a default implementation using the default keyword. Default methods enable new functionality to be added to existing interfaces without breaking the classes that already implement these interfaces. For example,

```

public interface MyInterface {
    void abstractMethod();

    default void defaultMethod() {
        System.out.println("This is a default method.");
    }
}

```

Strings in Java

Strings in Java are very similar to the strings we discussed in C++, through the use of the `#include<string.h>`.

The `String` class provides a convenient mechanism for working with strings in Java. String objects in Java are immutable - once assigned a value, it cannot be changed. The compiler converts string constants to `String` objects directly.

Some notes on strings:

- `String` object comparisons are achieved using the `compareTo()` and `equals()` methods.
- The `String` class in Java is the only class in Java with overloaded operators of "+" and "=". You should not use "==". So, the "+" operator allows two strings to be concatenated to form a new `String` object.
- Java strings are not null terminated, like C++ strings.

So for example:

```
1
2
3  import java.lang.*;
4
5  public class StringTest
6  {
7      public static void main(String args[])
8      {
9          String x = "Hello ";
10         String y = new String("World!");
11
12         String z = new String(x + y);
13         System.out.println(z);
14         System.out.println("The length of this string is "
15             + z.length() + " characters.");
16
17         String c = "Cat";
18         String d = "Dog";
19         if (c.compareTo(d)<=0)
20         {
21             System.out.println( c + " is less than " + d);
22         }
23
24         if (c.equals("Cat"))
25         {
26             System.out.println("The c object is equal to Cat");
27         }
28
29         String shout = "HELLO!";
30         System.out.println("A bit quieter - " + shout.toLowerCase());
31     }
32 }
33
34
```

The full source code for this example is in **StringTest.java**.

The output of this example is:

```
C:\Temp>javac StringTest.java
C:\Temp>java StringTest
```

```

Hello World!
The length of this string is 12 characters.
Cat is less than Dog
The c object is equal to Cat
A bit quieter - hello!

```

As of J2SE 1.5 an alternative format is available as demonstrated in **StringTest2.java** which demonstrates an alternative form (C Style) outputting, where we can have slightly more control over our output. For example,

```

...
System.out.printf("The length of this string is %d characters.\n", z.length());
...
System.out.printf( "%s is less than %s \n", c, d);
...

```

This is the new `PrintStream` `printf` method. This method allows us to use format specifiers as placeholders for different types, e.g. `%d` for an integer variable and `%s` for a string. Note that in this example `printf` is used instead of `println` for this task. The `%` in the format string denotes the start of a format specifier. The `%s` and `%d` in this example indicates the insertion of a `String` and int value. The use of a `%s` on an object will trigger the call to `toString()`.

A full list of the methods available in the `String` can be found in the Java API documentation under the `java.lang.String` class description. **Note** that the `String` class is hyper-linked in the notes HTML documentation. Where possible I have hyper-linked standard classes to provide direct access to the API documentation.

The `StringBuffer` object can be used instead of a `String` object when more flexibility is required in the object. The `StringBuffer` class is mutable and so can grow in size as the application runs.

We can convert directly between a `String` object and a `StringBuffer` object when required. Some of the functionality available in the `StringBuffer` includes:

Constructors:

```

StringBuffer()
StringBuffer(String s)

```

Methods:

```

int length()
char charAt(int index)
String toString()
StringBuffer append(String s)
StringBuffer insert(int offset, String s)

```

So, for example:

```
1
2
3 import java.lang.*;
4
5 public class StringBufferTest
6 {
7     public static void main(String args[])
8     {
9         String s = new String("Hello World!");
10        StringBuffer buffer = new StringBuffer(s);
11
12        buffer.insert(5, " to the");
13
14        String t = buffer.toString();
15        System.out.println(t);
16    }
17 }
18
19
```

The full source code for this example is in **StringBufferTest.java**

Java Arrays

Java Arrays are quite like those of C++. There is a special length state that returns the size of the array. With arrays in Java, the Virtual machine constantly checks to see if the index goes out of bounds, and throws an exception if it does (we will talk about exceptions later!).

```
char s[];           // a variable s to be an array of char
int[] anArray;      // a variable anArray to be an array of type int
int[][] a2DArray;   // a 2-D array of type int
int initArray[][] = { {1,2,3}, {4,5,6}, {7,8,9} };
    // Creates an initialised 3x3 array of type int.
```

//You cannot write int A[5]; as in C++

```
int myArray[];      // just 4 bytes of memory
myArray = new int[10]; // allocate the room for 10 ints.
```

The rules are similar to C++ when dealing with arrays of objects.

```
Object objArray[];
objArray = new Object[5];

// we have allocated room for 5 objects, but we have
// not created 5 objects.
```



```
for (int i=0; i<objArray.length; i++)
{
    objArray[i] = new Object(); // create the object using the constructor
}
```

The `Object` class

This may be a little confusing, but Java has an `Object` class. By default, every class extends the `Object` class, even if you don't extend any class. The `Object` class provides methods that are inherited by all Java classes, such as `equals()`, `getClass()`, `toString()` and more.

Consider the advantages in a common behaviour of every class:

- Since `Object` is the parent of all classes, then every class must have a `toString()` method. So when we write `System.out.println("Value =" + someObject);` It can immediately be converted to `System.out.println("Value =" + someObject.toString());` by the compiler, as each object must have this method.
- Consider passing data across a network. We could write methods to send `Object` objects (no typo!) across the network. Since every class extends the `Object` class, we can convert any class to an object of the `Object` class, transmit it, and then extract it again on the receiving end. This will prove very useful.

The `Class` class

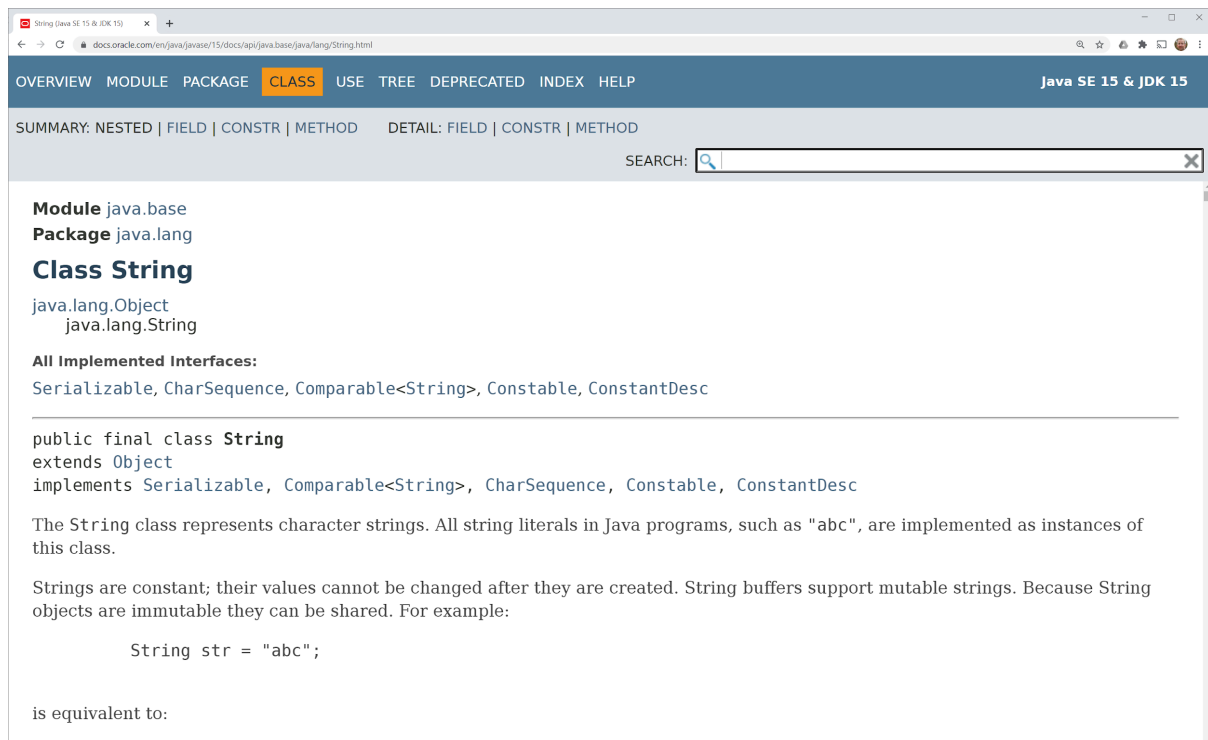
Objects of this class, the `Class` class (called class descriptors) are automatically created and associated with the objects to which they refer. For example, the `getName()` and `toString()` methods return the `String` containing the name of the class or interface. We could use this to compare the classes of objects.

The Java API Documentation

The Java API Documentation is the core reference that we have in the Java programming language and it is vital that you understand how to navigate, comprehend and apply the documentation to your code.

Figure 5.5, "The Java API Documentation" shows a screen grab of the API documentation in use.

Figure 5.5. The Java API Documentation



The screenshot shows the Oracle Java API documentation for the `String` class in Java SE 15 & JDK 15. The page is titled "Class String" and is part of the `java.lang` package. It lists the module as `java.base` and the package as `java.lang`. The class `String` is shown as a `java.lang.Object` subclass. The "All Implemented Interfaces" section lists `Serializable`, `CharSequence`, `Comparable<String>`, `Constable`, and `ConstantDesc`. The class declaration is shown as `public final class String` extending `Object` and implementing the listed interfaces. A description states that the `String` class represents character strings and that all string literals in Java programs are implemented as instances of this class. It also notes that strings are constant and immutable, and can be shared. An example code snippet shows `String str = "abc";`. The page also mentions that this is equivalent to a certain representation.

Below, you will find the API documentation for the `String`, showing a description of the class and the constructors and methods available. This piece of text is much easier to read in the API documentation (`String`).

<https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/lang/String.html>

Differences and Similarities between C++ and Java

Introduction

"Claiming Java is easier than C++ is like saying that K2 is shorter than Everest."

--Larry O'Brien (editor, Software Development)

Now that we have covered C++ in some detail, discussing concepts of object-oriented programming as applied to C++, it makes sense to discuss the similarities and differences between C++ as we have used it so far, and Java as we have begun to use it. The good news is that there are enough similarities between the languages to allow a programmer to have a choice to use the correct language for the application to be developed, without having to enter a whole new thought process.

The General Language

"Arguing that Java is better than C++ is like arguing that grasshoppers taste better than tree bark."

--Thant Tessman

This section will discuss some general differences between the syntax used in the C++ and Java languages.

Data Types

There are some key differences between Java and C++ that must be noted when dealing with a discussion of data types:

- Java has no pointers - all types are passed by value in Java (but non-primitive types appear to be passed by reference) and memory management is automatically taken care of.
- Java has no `struct` or `union` types - These are maintained in Java using classes.
- Java has only 8 supported types - `byte` (1 byte), `boolean` (1 byte), `char` (2 bytes), `short` (2 bytes), `int` (4 bytes), `long` (8 bytes), `float` (4 bytes) and `double` (8 bytes). All are signed and all are initialised to zero when declared - e.g. `int x;` // `x` will have a value of 0.
- Java also natively supports helper or wrapper classes to support these types - `Integer`, `Boolean`, `Double` etc.

In Java the `char` type has 2-bytes to allow support for international character sets as opposed to the single byte in C++. The `boolean` type in Java is a true `boolean` type and not just the `int` value of 0 and non-zero, as in C++. The value stored in a `boolean` type variable must either be `true` or `false` - you cannot perform an assignment like:

```
boolean myFlag = 1; // not allowed - even if you try to use a cast
```

A reference in Java to use objects is similar to the way that we use objects in C++.

```
Account a = new Account(500.0);
```

In C++ the `new` operator returns a pointer, but in Java the `new` operator returns a reference to an object and is the only access that we have to the object. This is the only way that we operate on objects in Java, we never operate on the objects directly, rather we only operate on the reference.

In Java when we create a reference without calling a constructor, it will be initialised to `null` . e.g.,

```
Account a;
```

This account reference `a` will be initialised to `null` and a `NullPointerException` will be thrown if you tried something like `a.display();` as the constructor of `Account` has not yet been called.

Type conversion - Historically C++ casts between pointers and references were unchecked. This has been improved in recent years, preventing us from using a pointer to invoke a method on an object that the pointer "is aware of" but the object "is not aware of". In Java casts that always succeed are allowed, but casts that will always fail are detected at compile time.

In C++ and Java, the compiler performs type checking during the first pass, preventing the incorrect use of arguments in methods, assignments etc. This is called **static type checking**. The Java language includes **dynamic type checking**, where some type checking is also performed at run-time, providing for powerful checks, but leading to a performance decrease.

Comments

As discussed previously, we have the block comment and end of line comment formats in C++. These formats are supported by Java and have the exact same syntax. Java has the addition of a third type of comment beginning with `/**` and ending with `*/` relating to the **javadoc** tool that is in the `/jdk7/bin` directory. This tool allows the generation of automatic code documentation, created following this format. There are several keywords (e.g. `@author`, `@version` etc.) that have meaning within the comment block, and you can even embed HTML tags. For example:

```
/** This is a method to lodge money to the account object
 * @author Derek Molloy
 * @version 1.0
 * @param amount - The amount to be lodged
 * @return void
 */

public void makeLodgement(float amount)
{
    ...
}
```

Once you have written the code, you can run the **javadoc** application on the specified file/package. This will then generate the HTML format documentation for your code. The Java API documentation was created using an internal SUN version of **javadoc**.

Memory management

The Java garbage collector manages all memory in Java. It reclaims all dynamically allocated memory automatically.

C++ developers are required to provide their own memory management routines, otherwise a complex C++ application, running for a long period would simply cause the computer to run out of memory. Implementing memory management correctly in C++ is complicated, and so memory leaks are a very frequent problem with C++ developed

applications. Garbage collection was not added to the C++ language specification, as all developers would be required to deal with a standard garbage collector and every C++ application would incur the overhead. There are many 3rd party C++ garbage collectors available, such as the Boehm-Demers-Weiser ^[10] conservative garbage collector, that replaces the C `malloc()` or C++ `new` calls (It can alternatively be used as a leak detector!).memory for objects once there are no remaining references to that object. When the garbage collector runs, it searches for all objects that have no references and reclaims the memory.

Where are objects stored?

When discussing Memory management and Garbage Collectors it is useful to discuss some aspects of how memory is laid out while the program is running, in particular how memory is arranged. There are six different places to store data:

- **Registers:** Registers are the fastest storage location because they exist inside the processor. However, the number of registers is severely limited, so registers are allocated by the compiler according to its needs. The programmer does not have direct control over the registers.
- **The Stack:** The stack exists in general RAM (Random-Access Memory), but has direct support from the processor via its stack pointer. The stack pointer is moved down to create new memory and moved up to release that memory. This is an extremely fast and efficient way to allocate storage, second only to registers. The Java/C++ compilers must know, while they are creating the program, the exact size and lifetime of all the data that is stored on the stack, because it must generate the code to move the stack pointer up and down. This constraint places limits on the flexibility of our applications. In Java some storage exists on the stack - in particular, object references, but Java objects are not placed on the stack. Variables on the stack are often referred to as automatic (or scoped) variables.
- **The Heap:** The heap is a general-purpose pool of memory in the RAM area (where all Java objects live). The difference between the heap and the stack is that the compiler does not need to know how much storage it needs to allocate from the heap or how long that storage must stay on the heap. There is a great deal of flexibility in using storage on the heap. In Java, whenever you need to create an object, you simply write the code to create it using the `new` keyword and the storage is allocated on the heap when that code is executed. The flexibility associated with the heap has a penalty in that it takes more time to allocate heap storage. In C++ we have the same flexibility, but the programmer must take more responsibility, creating an object on the heap using the `new` keyword, but importantly, also using the `delete` keyword when finished with the object.
- **Constant Storage:** Constant values are often placed directly in the program code, which is safe since they can never change.
- **Static Storage:** Static variables are also stored in RAM. Static storage contains data that is available for the entire time a program is running (Even when the object/reference is not in scope). You can use the `static` keyword to specify that a particular element of an object is static, but Java objects themselves are never placed in static storage. The static storage area is a fixed patch of memory that is allocated before the program begins running.

- **Non-RAM Storage:** If data lives completely outside a program it can exist while the program is not running, outside the control of the program. The two primary examples of this are streamed objects, in which objects are turned into streams of bytes, generally to be sent to another machine, and persistent objects, in which the objects are placed on disk so they will hold their state even when the program is terminated. The trick with these types of storage is turning the objects into something that can exist on the other medium, and yet can be resurrected into a regular RAM-based object when necessary. Java provides some support for persistence and is frequently discussed in the context of database object mapping.

No Pointers

There are generally few problems with this fact as Java provides suitable replacement through the use of well-formatted references.

Global Functions

In Java all methods must be defined within a class. There are no global methods or variables. This might seem unreasonable, but it works well. For example, the mathematical routine square root can be accessed through the `Math` class - a `static` class that allows you to call methods like `Math.sqrt(25.0)`; allowing us to even replace mathematical operations, perhaps with different precision operations, by importing our own mathematical class.

Strings

Native strings in C++ are represented by an array of characters, terminated by `'\0'`. Strings in Java are objects of the `String` class and are immutable, i.e. cannot be modified. There is no operator overloading in Java (as there is in C++) but the `+` operator when applied to strings causes the creation of a new `String` object which is the sum of the two strings to which the operator is applied. The `'+', '=' and '=='` operators are also valid when dealing with the `String` class, however be aware that the `'=='` is used to compare the reference addresses and not the data contained by the objects.

Arrays

In Java, when you declare an array of a class, you could say:

```
Integer[] a = new Integer[5];
```

But this creates only an array of references, you must then instantiate each reference, using

```
a[1] = new Integer();
```

Even for the default constructor (as shown). In C++, when you declare:

```
Integer[5];
```

C++ would call the default constructor for each instance of the array, and there would be no need to explicitly call 'new' again to instantiate each reference of the array.

This form of initialisation only allows you to call the default constructor (or constructor with no parameters). In C++, you can create an array of objects, with each object sitting right next to each other in memory, which lets you move from one object to another, simply by knowing the address of the first object. This is impossible in Java.

In C++ you can also create arrays of pointers or references to objects. This is more closely related to how Java arrays work. Java arrays are always an array of references to objects (except if you create arrays of the basic data types, `int`, `float`, `double` etc.). If you were to use an array of pointers or references, you would have an array of null references (just like Java). The syntax would be like: `Integer **f = new Integer*[10];`

This C++ code segment shows how to create an array of `A` objects `arr[10]` as the state of the `B` class, and have the non-default constructor called for the objects.

```
#include <iostream>
#include <string>

using namespace std;

class A {
private:
    int x;
public:
    A(int);
    A();
    virtual void display();
};

A::A() {}
A::A(int y): x(y) {}

void A::display() {
    cout << "Object has the value " << x << endl;
}

class B {
private:
    A arr[10]; //default A constructor A() is called here (10 times).
public:
    B();
    virtual void display();
};

B::B() {
    for(int i = 0; i < 10; i++)
        arr[i] = A(i); //the non-default constructor
```



```

        // A(int) is called here (10 times).
    }

    void B::display() {
        for(int i = 0; i < 10; i++)
            arr[i].display();
    }

    int main() {
        B b;
        b.display();
    }

```

A Java array is guaranteed to be initialized and cannot be accessed outside of its range. The range checking comes at the price of having a small amount of memory overhead on each array as well as verifying the index at run time, but the assumption is that the safety and increased productivity is worth the expense. Java has run-time bounds checking on arrays, throwing an exception when an operation is performed that is out of bounds. There is a special instance variable called `length` that holds the number of elements in the array. Arrays in Java must be created using the `new` keyword.

When you create an array of objects in Java, you are really creating an array of references, and each of those references is automatically initialized to a special value with its own keyword `null`. When Java sees `null`, it recognizes that the reference in question is not pointing to an object. You must assign an object to each reference before you use it, and if you try to use a reference that is still `null`, the problem will be reported at run-time. Thus, typical array errors are prevented in Java. You can also create an array of primitives. Again, the compiler guarantees initialisation because it zeroes the memory for that array.

Method Parameters

In Java all variables of the standard types (as detailed in the section called "Data Types") and references are passed by value (i.e., never by reference or by pointer). For example:

```

public class Test {
    public void square(int x) { x = x*x; }

    public Test() {
        int y = 5;
        System.out.println(" The value of y is " + y); // outputs 5
        square(y);
        System.out.println(" The value of y is " + y); // outputs 5
    }

    public static void main(String[] args) {
        new Test();
    }
}

```

In Java if you pass an object to a method, you are always passing a reference to the object (NB: NOT passing by reference, you are passing this reference by value). This means that even though you are passing the reference by value, you are always operating on the original object., just as if you had used pass-by-reference in C++. For Example:

```

1
2
3 public class SomeClass {
4     public int x = 2; // just for demonstration - set public
5 }
6
8 public class Test {
10     public void square(SomeClass s) { s.x = s.x * s.x; }
11
12     public Test() {
14         SomeClass y = new SomeClass();
15         System.out.println(" The value of SomeClass x is " + y.x);
16         // outputs 2
17
18         square(y);
19         System.out.println(" The value of SomeClass x is " + y.x);
20         // outputs 4
21     }
22
23     public static void main(String[] args) {
25         new Test();
26     }
27 }
28
29

```

When dealing with assignments in Java we have a very important difference between code written in C++ and code written in Java, that on initial inspection seems exactly the same.

Looking at the C++ version:

```

/* In C++ */
    Account a(600);
    CurrentAccount b(500,5000); //bal = 500, overdraft = 5000
    a = b;
    a.display(); // results in "I am an account" being displayed
                // with no mention of overdraft and a balance
                // of 500. The compiler would have prevented b = a;

```

Looking at the equivalent Java version:

```
/* In Java */
Account a;
CurrentAccount b = new CurrentAccount(500,5000);
a = b;
a.display(); // results in "I am a current account"
              // with an overdraft of 5000 and a balance of 500
```

In C++ when we assign `a=b;` the `CurrentAccount` object is simply sliced into an `Account` object - i.e. "fitting into the box for an `Account` object". In Java we simply have a reference to the original object so that object never changes - there is no slicing performed.

OOP Concepts in C++ and Java

Similar Constructors

Constructors work almost exactly the same way in C++ and Java. If you do not define a constructor for a class then it is allocated a default constructor. If we do define a constructor then we must use it. The only significant difference is that there is no copy constructor in Java.

No Destructors!

There are no destructors in Java, even though there is a `new` keyword, there is no corresponding `delete` keyword, as Java takes care of all of the memory management. Java has a special `finalize()` method that you can add to any class. The strange thing about this method is that you can use it for tidying up (printing a paper record for `Account` etc.), but you do not know when it will be called. In C++ this was easier, as the destructor is called when the object goes out of scope, but in Java the `finalize()` method will be called when the garbage collector destroys the object. We do not know when this will be called! As discussed previously, we can request that the garbage collector should run, but we do not know when it will run! You can help the garbage collector slightly by setting all your references to `null` when you no longer need them, e.g. `a = null;` - will send a hint to the garbage collector that the object is no longer being used.

Access Specifiers

Like C++, in Java we have `public`, `private` and `protected` access specifiers, but we also have another access specifier "package". This is the default access specifier and means that all states and methods are accessible to all classes within the same package. There is no package access specifier keyword, it is simply the default if `public`, `private` or `protected` are not used. Access specifiers in Java are:

- `public` - accessible everywhere, an interface method (same as C++)
- `private` - accessible only in the class where it was defined (same as C++)
- `protected` - accessible in the class where it is defined, the derived classes and in the same package (almost the same as C++ with the exception of the package)

- "package" - default (there is no `package` specifier keyword) and means that it is accessible by any class in the same package. You can equate this to the C++ friendly condition by saying that all of the classes in the same package (directory) are friendly.

In Java there is no direct equivalent to `protected` in C++. In Java version 1.0 there was! it was called "`private protected`", but was deemed complex and unnecessary, and so was removed.

Access Specifiers when Inheriting Classes

Inheritance in Java does not change the protection level of the members in the base class. You cannot specify `public`, `private` or `protected` inheritance in Java, as you can in C++. Overridden methods in a derived class cannot reduce the access of the method in the base class. For example, if a method is public in the base class and you override it, your overridden method must also be public. The compiler will enforce this rule.

Virtual and Non-Virtual Methods

In C++ methods are non-virtual by default, so to replace the behaviour (allow overriding) of a method in the derived class you have to explicitly use the `virtual` keyword in the parent class. You are able to replace the behaviour of a non-virtual method but this causes serious difficulties when the behaviour of the object depends on the static rather than the dynamic type! In Java, methods are virtual by default and we always operate on the dynamic type of the object. You cannot specify a method as non-virtual, but there is a `final` keyword. Once you use the `final` keyword on a method you cannot replace it - so there are no issues with static and dynamic types.

Multiple Inheritance

C++ allows multiple inheritance - Java does not! As discussed previously in the C++ section of the notes, multiple inheritance is complex for the programmer, in that they must make complex decisions about the way that the data of the base class is to be inherited. In Java, we do not have multiple inheritance but we do have the use of Interfaces. Interfaces are a special kind of abstract class that have no data or implemented code. Interfaces might not sound too useful, but they allow a form of multiple inheritance, without having the associated difficulties of dealing with data. This replacement behaviour works very well and does not impact on development, once you get used to the change.

No scope resolution operator

There is no scope resolution operator in Java - such as `Account::display();` as used previously in C++. This is possible because you must define all methods and states within a class so there is no global namespace as we had in C++. We have the `super` and `this` keywords which allow us to access the parent (only through the direct parent) and the states of a class. Packages are used instead of namespaces. When you use the `super()` method in the constructor to call the parent constructor, it must be the first line of the child constructor implementation.

Nested Classes

In Java we have the facility to define a class inside of another class. We refer to this class as an **inner class**. The only issue worth mentioning with inner classes is that they have full access to the private data and methods of the class in which they are nested. Because of this, we do not need to pass state pointers to methods that need callback.

Inline Methods

In C++ we had inline methods, generally short methods that the compiler in effect cuts-and-pastes the method code into the final program, leading to greater efficiency as the assembly language program counter is not jumping around in the final executable. There is of course a trade-off between speed of execution and size of the executable. There are no inline methods in Java - well that's not entirely true. It is up to the compiler to decide whether a method should or should not be inline. There is a keyword in Java called **final** which suggests to the compiler that the method should be inline, but you have no guarantee that the compiler will agree!

The Compiler

General Compiler issues

There are no forward declarations of classes/methods necessary in Java. The compiler takes care of all definitions of methods and does not throw an error until it is sure that the method or class has not been defined elsewhere.

Libraries

The standard Java libraries are supplied by Oracle and are comprehensive. In C++, the programmer uses third party code libraries that do not integrate very well with each vendor. Java provides comprehensive standard libraries from application development to networking and database access providing us with a fully integrated development language.

Exceptions

Exception handling allows run-time errors to be handled directly by the programmer. An exception object is thrown from the line of code where the error occurred and is caught by a suitable exception handler, specially designed for that type of error. Exceptions are in effect control statements that allow a different path of execution when errors occur to the normal path when no errors occur. In the Java and C++ languages exceptions cannot be ignored by the programmer, rather they must deal with them, leading to more robust applications. Exceptions exist outside of object-oriented languages, but in object-oriented languages an object is used to describe the exception.

While we did not cover exceptions in C++, they are available. However, exception specifications in Java are vastly superior to those in C++. Instead of the C++ approach of calling a function at run-time when an exception is thrown, Java exception specifications are checked and enforced at compile-time. In addition, Java overridden methods must conform to the exception specification of the base-class version of that

method: they can throw the specified exceptions or exceptions derived from those. This provides much more robust exception-handling code.

Default Arguments

There is no support for default arguments in Java.

No Preprocessor

There are no compiler directives, no `#define`, no `#includes`

Multithreading

Java has support built into the language for multi-threading, C++ does not.

I will add more to this document as issues arise. If you notice any errors or omissions in the document then please drop me an e-mail.

^[10] See: http://www.hpl.hp.com/personal/Hans_Boehm/ and http://www.hpl.hp.com/personal/Hans_Boehm/gc/index.html