PHAS3459: Scientific Programming Using Object Oriented Languages

Module 7: Building Larger Programs

Dr. B. Waugh and Dr. S. Jolly
Department of Physics & Astronomy, UCL

Contents

1. Aims of Module 7

The aims of this module are to give you an overview of the architecture for compiling and executing Java programs, and to introduce you to some tools and techniques for structuring and documenting your programs and classes.

You will learn about some of the important aspects of developing large programs, including the organisation of code into packages. You will be able to use Javadoc to generate web-based documentation of your own Java code, and you will be aware of the large library of utility classes that are available to aid program development.

While they are particularly important for large and complex software, the topics covered in this module are useful even for smaller projects, and you will be expected to apply them in the remaining coursework and in the final exam.

2. The Java Programming Language

2.1 How Java Works

In this section we will see in more detail what is involved in executing a Java program. This has been happening "behind the scenes" when you run your programs using the Eclipse integrated development environment.

All high-level programming languages are in some sense highly portable. A program written in Fortran or C++ can be run on many kinds of computers. All that is required is a compiler appropriate to the given machine and operating system. The compiler takes the code and translates it into an *executable*, a file of low-level instructions that are understood by the computer in question. This kind of portability is more than adequate for many purposes, including most scientific applications.

The architects of the Java language have taken things one step further, following the paradigm of "write once - run anywhere" (although perhaps "compile once - run anywhere" would be more appropriate). Not only is the Java code portable, but the compiled result — called "byte-code" — can be run anywhere that has a suitable environment installed. There are two components required to achieve this, as illustrated in Fig. 7.1:

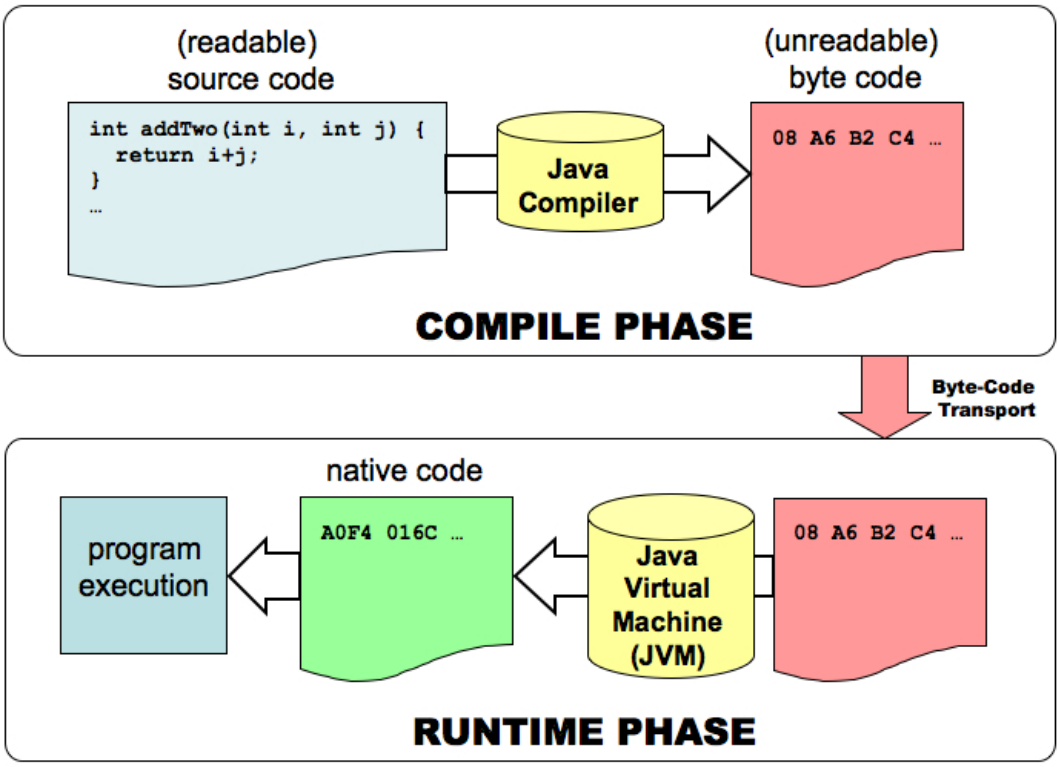

Figure 7.1: The phases involved in running a Java program. First, the human-readable Java code is compiled using the Java compiler that is part of the Java Development Kit (JDK) into a set of low-level instructions called "byte-code". The byte-code can be sent to any other machine that has the Java Runtime Environment (JRE) installed. The byte-code is written for a Java Virtual Machine, which in effect translates it into native instructions understood by the particular computer in question.

- **Compile Phase**. The "`xxx.java`" files are compiled to produce "`xxx.class`" byte-code files. Regardless of the machine on which the Java compiler runs, or where the eventual program will be executed, the same byte-code file is produced. The `.class` files are machine independent instructions. In fact they can be thought of as instructions for a hypothetical ("virtual") computer, the architecture of which which has been specified by the Java language developers. The compiler, along with libraries of useful classes and other utilities, form part of the "Java Development Kit" (JDK) that must be installed on any computer being used for Java program development.
- **Runtime Phase**. The byte-code created in the above step can be transported to any other machine that has the "Java Runtime Environment" (JRE) installed. The "Java Virtual Machine" (JVM) can read the byte-code file, translating the contained instructions into native instructions for the particular computer hardware and operating system on which it resides. Typically this translation is done "on-the-fly", in other words the byte-code is "interpreted" as it is run[1].

How does this achieve the required Java program portability? The crucial thing is that the compilation need only be done once to generate a portable byte-code file which can then be run on any Java-enabled computer or device. A single downloadable object (e.g. via the

web) can run on any number of client computers. By contrast, program portability with another language such as C++ could only be achieved by either (a) downloading the code and compiling each time (this would be slow and very error prone) or (b) having available for download a library of pre-compiled executables for all the different platforms that the program may run on (requiring a lot of storage space, and being difficult to maintain).

There are JVMs for all types of computer and operating system, as well as non-PC devices such as mobile phones, set-top boxes, etc. JVMs are often incorporated into application software such as web browsers.

2.2 The History of Java

2.2.1 The Origins of Java

After several years of development under different names, the first generally available version, Java 1.0, was released by Sun in 1996. A large part of its early success was due to its use in *applets*: Java programs that would run within a web browser, providing more sophisticated functionality than that available using HTML alone. Since then, the Java language has evolved significantly and additional specialised editions are also available for developing applications to run on web servers and on small devices such as mobile phones.

2.2.2 Java Versions

The Java language contains many built-in classes. When new classes are added either by Sun developers or popular demand, or if extensions to the language are added, then Sun release a new Java version. New classes or language features often supersede previous implementations. The previous implementations can still be used (in other words, "backwards-compatibility" is maintained) but the user is encouraged to migrate code to the newer and better implementation. The older implementations are said to be "deprecated" and the compiler will warn the user if they try to use them.

The history of Java versions is approximately as follows:

- **1.0**: January 1996.
- **1.1**: February 1997.
- **1.2**: December 1998. There were significant changes between **1.1** and **1.2** (for example the graphics classes were re-written); so much so that releases from 1.1 to 5.0 are often referred to as **Java 2**.
- **1.3**: May 2000.
- **1.4**: February 2002.
- **5.0**: September 2004. (also known as version 1.5). It includes some significant features not present in earlier versions, such as generic types and automatic conversion between primitive types (e.g. `int`) and objects (e.g. `Integer`), as well as the newer syntax for looping over collections, e.g. `for (int element : array) ...`
- **6**: December 2006. The language differences from version 5.0 are not significant for the purposes of this course, but the performance was improved.
- **7**: July 2011. Some additional language features were introduced, including:

    - new exception-handling capabilities
    - a new framework for performing operations on files
    - a simpler way of ensuring files and similar objects are closed correctly after use

  These are not currently used in this course because until quite recently Java version 6 was still widely used. Changes were also made to the JVM to provides better support for languages other than Java, especially *dynamic languages* where the type of a variable may not be known at compile time but only during execution of the program.
- **8**: March 2014. Includes *lambda expressions* for functional programming.
- **9**: Planned for 2016. Expected to include an improved approach to modularity, and possibly support for GPU programming using OpenCL.
- **10**: Possibly 2018. No firm commitments yet about what will be included, but possibilities include *value types*, which would allow programmers to define types that could be used much more efficiently and quickly than normal Java objects, by doing without some of the sophisticated features of objects to become more like the built-in *primitive* types.

---

[1] It is also possible for the byte-code to be further compiled into a file comprising machine dependent instructions using a "Just-In-Time" (JIT) compiler, increasing the performance of Java applications. ↩

---

3. Software Development in Java

While the projects you have so far developed are small in scale, for large-scale scientific or commercial programming enterprises the effective organisation of the code is critical. This section discusses a few of the tools and techniques you can used to make your programs more manageable.

3.1 Java Software Deployment

There are four file types or structures for organising Java code:

- `classname.java`: the definition and implementation of each Java class is contained in such a file.
- `classname.class`: the compiled byte-code for each class is stored in such a file.
- `name.jar`: many class files can be packaged together (e.g. for distribution) in a single jar (= **j**ava **ar**chive) file.
- package: related source code is organised into directories called packages.

3.2 Java Packages

For large projects it becomes unwieldy to have all the code in one logical location. Java has the concept of *packages* for grouping together related code files; for example all the Java classes relating to file manipulation might reside in one package while all those that form part of a physics simulation might reside in another.

Packages are nothing more than directories (or folders) in a hierarchy and indeed package names must match corresponding directory names. If a Java package is referred to as "`aaa.bbb.ccc`" then there will be a corresponding directory "`aaa/bbb/ccc`" (or "`aaa\bbb\ccc`" on a Windows machine) containing the corresponding Java files. Package names should be descriptive and unique.

We take advantage of Java classes in a different package by "importing" the code. You have already done this many times at this point in the course. One thing to note is that if two packages contain class definitions with the same name, one must use the fully qualified name to distinguish them:

```
package module10;

// This package contains a "physics"
// vector class. A single class in the
// package "ucl.physics.oo.example"
// is imported:

import ucl.physics.oo.example.Vector;

// This package contains a "container"
// vector class. All the classes in the
// package "java.util" are imported
// with the use of the "*" wildcard:

import java.util.*;

// ...

// In the code, make clear with fully
// qualified package names which one
// you want to use.

// A "vector" to represent a particle's
// 3-momentum:
ucl.physics.oo.example.Vector particleMomentum =
  new ucl.physics.oo.example.Vector();

// A "vector" to store a list of particle
// 3-momenta:
java.util.Vector particleMomenta =
  new java.util.Vector();

// ...
```

An important point to note is that if the code we are developing needs to use code in the same package then an "import" is not necessary. Code in the same package is automatically imported, as is code in the package `java.lang`.

3.3 Javadoc

You should by now be familiar with browsing the online Java API documentation, as shown in Fig. 7.2. In fact it is particularly easy to generate similar web-based documentation for your own code using a standard application called "Javadoc". As an example, take the following class definition:

```
package module10;

/** A class representing an electron.
 *  Inherits from Lepton, and also implements
 *  relevant interfaces.
 *  @author D.Waters
 *  @author B Waugh
 *  @version 1.3 (19/11/08)
 */
public class Electron extends Lepton
  implements SimulableParticle, ReconstructableParticle {

  /** Constructs an Electron with given momentum
   *  @param p three-momentum of electron
   */
  public Electron(ThreeVector p) {
    // ...
  }

  /** Get generation of lepton
   * @return generation (1 for electron)
   */
  public int generation() {
    return 1;
  }

  // ...
}
```

Note the comments preceding the class name and individual method declarations, beginning with "`/**`" and ending with "`*/`", that will be extracted by Javadoc. Comments in the body of the methods will be ignored. Note also the "`@author`", "`@version`", "`@param`" and "`@return`" tags, which will be recognised by Javadoc. You should get used to using these tags to document your classes and methods. Every public method should have at least a brief Javadoc comment stating what it does. The return value and any arguments should be documented using the `@return` and `@param` tags, except in cases where it is *very* obvious what they mean.

To run Javadoc within Eclipse, select the class, package or project of interest in the package explorer. Click on the "Project" menu and select "Generate Javadoc". The first time you do this you will have to tell Eclipse where to find the Javadoc application. You can browse the filesystem to find this within the JDK installation directory. As of 19th November 2013 on the Desktop@UCL machines it is at `C:\Program Files\Java \jdk1.7.0_40\bin\javadoc.exe`. When you click on "Finish", Javadoc will process the selected class and output the results in the following location (or similar, depending on your exact setup): "`N:\Eclipse\workspace\PHAS3459\doc\moduleX\YYY.html`".

The output from the code fragment above is displayed in Fig. 7.3. Note the way that the comment tags and the special fields were recognised by the Javadoc processor and displayed in the resulting web page. Note how the superclasses of `Electron` are displayed at the top of the web page, along with a list of the interfaces that have been implemented.

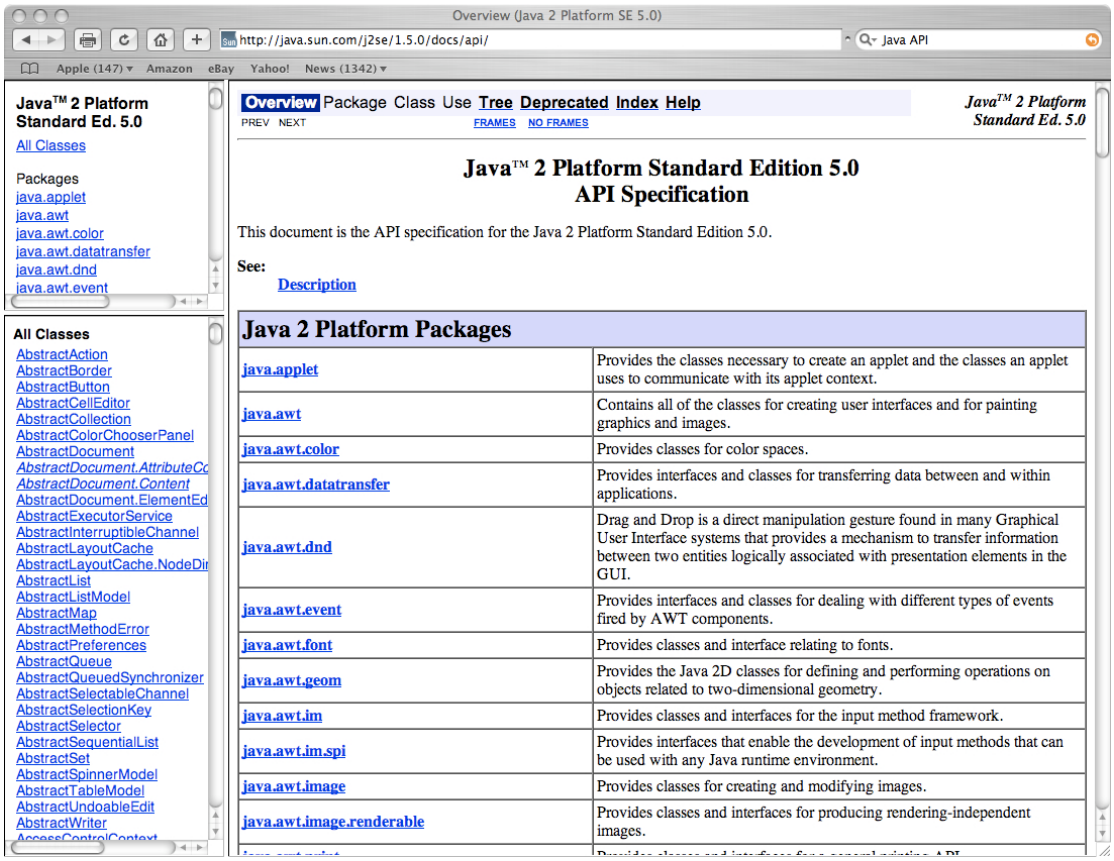

Figure 7.2: Online Java API documentation. Note the list of packages in the top left-hand corner. Note the layout of the documentation when a particular class is selected, containing information on the package membership, superclass inheritance, etc.

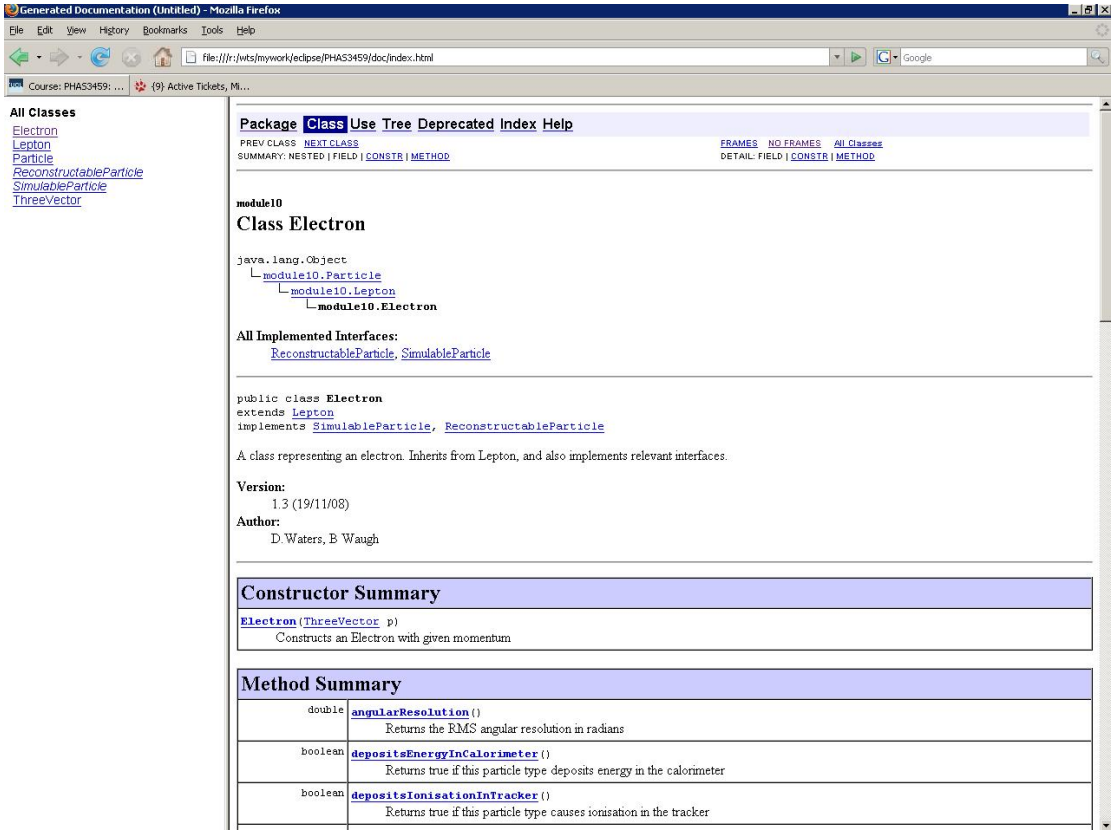

Figure 7.3: Javadoc generated web page for user-defined class "`Electron`". Note the information provided on the inheritance hierarchy and the interfaces implemented by this class. See text for further details.

Try running Javadoc on one of your own classes, suitably commented, and look through the resulting documentation using a web browser.

3.4 Using the Java API

Java is now a mature language. It supports in a rigorous and extremely well-tested way almost any generic operation that you need in your programming task such as I/O, networking, graphics, database access, security, mathematics, XML, threads, etc. A good programmer always takes the time to learn what already exists before re-inventing the wheel: using existing classes and infrastructure, especially those "officially" provided in the Java API, *dramatically* reduces the incidence of bugs.

Fig. 7.4 shows in outline the components of the Java **7** platform, including many utility packages and utilities we have already used. In the next module we will learn in much more detail about the "AWT" and "Swing" packages, and create applets that can run in a web browser using the Java plug-in.

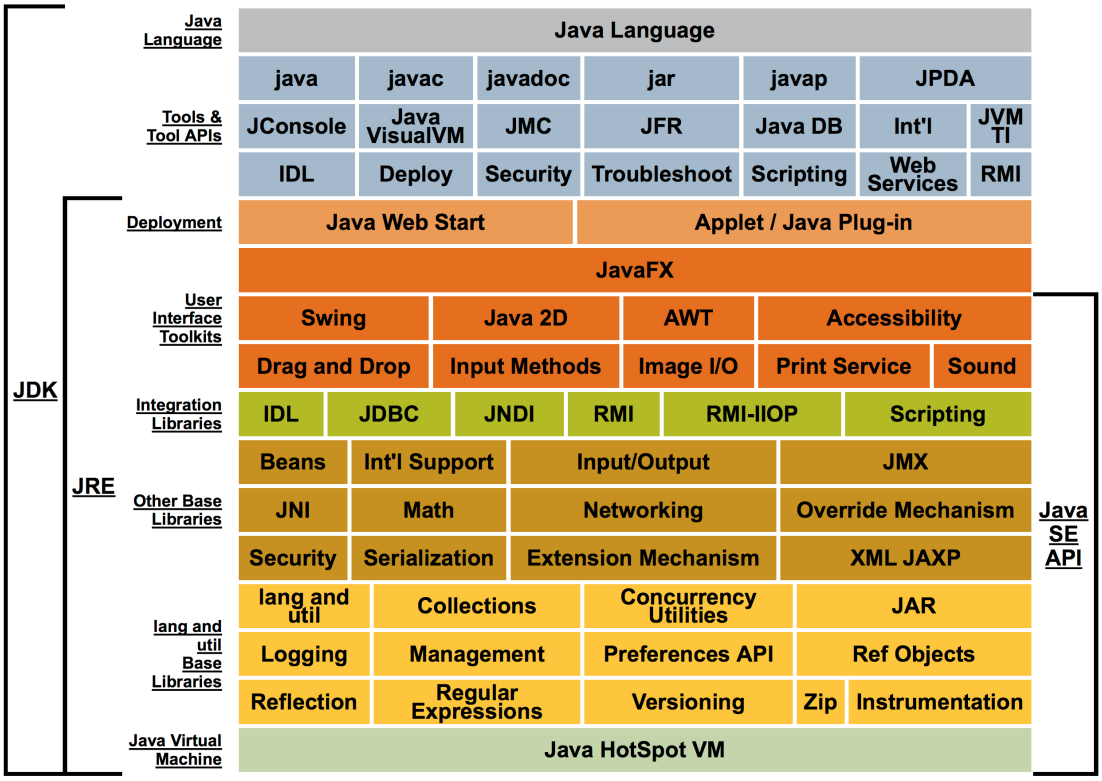

Figure 7.4: Organisation of the Java **7** platform, including many utility packages.

4. The Software Development Process

Many large books are available on the subject of developing software, and there are many *methodologies* for doing so, some more rigid than others. It is beyond the scope of this course to explore any of these, but we will give you a few general guidelines that may be helpful.

The object-oriented approach to programming places great importance on producing *modular* code. This is related to the topic of *encapsulation*, which we have already discussed. The principle is that it is easier to construct, debug and maintain a complex system if it is made up of smaller, simpler components each of which has well defined behaviour. Your choice of classes and methods, and which members are public, is therefore very important.

Even these relatively simple components may still contain bugs or design errors that cause the program as a whole to function incorrectly, so it is important to *test* your code by, for example, making sure it gives the correct output for a range of inputs. This may be done by adding a `main` method to a class such as `ThreeVector`, containing code to test each of the methods of the class, or by creating a separate test program. There are frameworks (such as *JUnit*) available to do this *unit testing* in a consistent way.

Even if all the components work correctly, it is still possible for them to interact in ways that the programmer has not foreseen, so testing of the program as a whole is also necessary.

Incremental development is also a useful approach. Trying to design and write a complex system in one go, with all the desired functionality, risks producing a program that does not work and is almost impossible to fix because it has so many (often subtle) bugs. In an incremental approach, the aim is to start by creating a simple system that does only a small amount of what is required, but which works. Then the software is extended a bit at a time to add further functionality, with a working but incomplete program produced at each stage. Testing at each stage is vital to ensure that the foundations are firm before building the next level, and one school of thought (*test-driven development* even holds that you should write code to test your classes before you start writing the classes themselves.

To create our `ThreeVector` class, for example, we might start with a very simple class that simply holds three `double` variables and has a `toString()` method so that we can see what their values are. Then (or even beforehand) we would create a separate class, or a `main` method, that creates a vector and prints it to the screen. Next we might add one or two constructors and some `get` and `set` methods, then `add` and `subtract`, moving on to `scalarProduct` and `vectorProduct`, writing suitable tests at each stage. The end result should be a `ThreeVector` class where every method has been well tested, and which we can use with confidence in other code.

One caveat is that incremental development is not a substitute for planning: if you have no idea what the completed program will look like you may end up with convoluted code and not be able to change it to do everything required. However, you should be prepared to adapt your design as you go along.

Another caveat is that this approach may need to be adapted for use in an exam! In this course you will get more marks for showing that you have made a reasonable attempt at each part of a problem, and know more or less how to solve it, than for a beautifully written, well tested program that does only a small part of what is asked.

5. Summary

In this module you have learned more about the basic architecture of Java and some of the ways to effectively organise and document your programs. Together with a knowledge of the classes already available to you in standard Java utility packages, this will greatly enhance your ability to execute large and complicated programming tasks using Java.