OOPJAV – Object-Oriented Programming in Java

Seminar 1 – Introduction to Java

Welcome to Object-Oriented Programming in Java. This module has two main goals:

1. To learn how to program effectively in Java.

2. To learn how to take advantage of the object-oriented approach to software development.

Java is a powerful and yet flexible language that is fully object-oriented. It also has the additional advantage that it can operate over the WWW. In this introductory seminar we will look at the origins of Java and the concept of Object-Oriented Programming (OOP).

1. Origins of Java

The Java language came to life in 1990 when a Sun Microsystems team headed by James Gosling designed a new programming language, known as Oak, for the development of consumer electronics software. It was not until the introduction of the WWW that the Java language came into being. With the introduction of the WWW, the Oak development team quickly realised that Oak-style programs (i.e. Java programs), could be invoked over the Internet using what has become known as a *Java applet*. The significance of this is that the language would be completely platform independent, a major milestone in an era where the operating system platform running on a machine dictated what tasks and programs were able to be created and run on that machine. For example, in order to run a C program on a PC platform, the machine needed a PC compiler for the C program, and to run the same program on a UNIX platform you needed a UNIX compiler, etc. To demonstrate this new platform independence, Sun developed the world's first *Java enabled* web browser and called it *HotJava*.

The term *Java enabled* means that inside the browser there is a hypothetical machine known as a *Java Virtual Machine* (JVM) which can run Java for any particular computer. The Java that comes over the net when an applet is invoked is encoded in something known as *Java Byte Code* (JBC). A JVM resident on a particular computer can interpret JBC and consequently "run" the applet on that computer.

This platform independence sets Java apart from any other language. However, Java programs do not have to be run from a Java enabled WWW browser; they can be run independently as *applications* in the traditional manner.

As a relatively young language (compared to languages such as Ada or C), Java is still evolving. The core of the language is small (with respect to other OO languages such as C++), yet more and more packages (libraries) are constantly being added. Version 1.0 of the language was launched by Sun in 1995. Version 1.1 was launched in 1997 and included minor modifications to the core language and major additions to the libraries. Version 1.2 was released in 1998 and built on Version 1.1. The current version is version 1.6, although many code generator programs are still running version 1.4.

2. Programming in Java

From the above introduction, we see that we can write two different kinds of Java programs:

- Applications: Stand-alone programs similar in nature to the kinds of programs that might be written in any other high level language (C, C++, Ada, etc.).

- Applets (small web-based applications): Programs that are executed from a web browser such as Internet Explorer. Applets typically allow the user to control execution through a Graphical User Interface (GUI) - a collection of components such as buttons and scroll bars within a window. An applet can be executed from any Java enabled browser.

In addition to these two programmatic approaches to Java, the language is also used to create remote computing bridges so that machines can communicate over distance. Applications of Java in remote computing include:

- Servlets: Java programs that extend the flexibility and functionality of Web servers. Servlets may be used to query databases, provide dynamic custom content for Web-based users, or track session information for clients.

- JSPs (Java Server Pages): JSPs are an extension of servlets and allow Web designers to further customize a user's experience by providing dynamically generated Web content.

- J2EE solutions: Java Enterprise solutions use a wide variety of Java-based technologies to fully integrate several aspects of a networked system.

In this module, we will focus exclusively on application development. We will, however, be learning the Java language basics, programming concepts, and graphic user interface elements – all of which may be used to develop the other types of Java applications listed above. This means that with further independent study, you will find that the Java basics presented in this course will provide you with a firm background so that you may continue your personal study and development in your Java area of choice.

3. Downloading the Java SDK

We will be using the suite of programs known as the JDK 6.0 with a text editor (textPad from www.textpad.com). You should use Java version (JDK 6.0) so that commutative problems with running code do not occur.

Detailed installation instructions are provided by Sun on the download site.

The site is located at: http://java.sun.com/javase/downloads/index.jsp Please refer to chapter "before you begin" in your textbook for further information.

4. The Mechanics of Creating Java Programs

A. Source Code

As with most other high level languages, a Java program is written as an ordinary *text file* created using a *text editor* like textPad. The text contained in such a text file is referred to as the *source code*. All Java source code text files must have the suffix **.java**.

B. Compilation

Once a program exists as a text file, the next stage is to compile (translate) this text into an executable form. Generally, most compilers (e.g. C, C++ and Ada compilers) translate source code into machine code. The disadvantage of this is that different compilers will be required for different languages and machines (machine code is machine specific), which is why languages such as C and Ada are not platform independent. Java, on the other hand, ***is*** platform independent because it compiles the source code into Java Byte Code (JBC).

JBC comprises a set of instructions written for the *Java Virtual Machine* (JVM). In this manner the Java language achieves its *platform independence*. We say that Java code is *portable* in that a Java program will run (without modification) on many types of machines. Portability is an important issue in software development and is an often-quoted advantage of the Java high-level languages.

Therefore, given a Java source code file called **MyPro.java** the Java compiler is invoked at the MS-DOS prompt as follows:

```
javac MyPro.java
```

The 'javac' command is the call to the Java compiler, and the file name following the command lists the file you wish to compile. This action will produce a Java byte code file called **MyPro.class** in the same directory where you had saved the MyPro.java file.

NOTE that the document name must be appended with .java and that Java is case sensitive, so you must enter the file name after the 'javac' command exactly as it is saved on your machine. The resulting .class file will also be created with the same case sensitive name.

C. Program Execution

In the case of languages such as C, C++ and Ada, once code has been compiled, the result may be *run* (executed) through a command to the operating system that includes the name of the program without any suffixes. However, in the case of Java, to achieve the desired platform independence, the Java byte code must be interpreted using an interpreter. Assuming the existence of a JBC file called **MyPro.class**, this interpretation can be achieved by typing:

```
java MyPro
```

at the MS-DOS prompt. NOTE that the DOS prompt must be pointing to the folder where the .java file is in order to compile the file using 'javac' and that it must be pointing to the folder where the .class file is in order to interpret and run the .class file using the 'java' command. Also note that the javac command is used with the file name plus the file extension (such as MyPro.java), while the java command only uses the file name of the .class file – not the extension (such as java MyPro).

Java can thus be viewed as a cross between a compiled high level language and an interpreted one.

We suggest that you practice compiling one or two short programs from the book using the DOS window so that you understand how the compilation process works.

5. The Object-Oriented (OO) Methodology

A systematic approach to software engineering (i.e. a methodology or a way of doing things) simplifies the process and results in software which is understandable, verifiable and reliable without stifling the creativity of the software engineer. One well-tried and tested methodology is o*bject-oriented* software which dovetails nicely with the use of an object-oriented language such as Java. Object-oriented languages encourage (or enforce) the OOP paradigm.

Note: *The correct terminology is "Object-Oriented," NOT "Object-Orientated." Please bear this in mind as we go through this class!!!*

Unlike other programming language paradigms, the OO paradigm is both a programming style as well as an approach to software engineering.

The principal features of the methodology are:

1. Flexibility and adaptability
2. Software reuse and extension
3. Information hiding
4. Consistent notation and integration of the various software engineering phases
5. Reliability

So what is object-orientation? The key element of the object-oriented methodology, as the name suggests, is the *object*. An object is some real-world entity which a programmer wishes to model, or represent in the program. We encounter objects every day that we could model in a software program.

In **Figure 1**, an object (let's call it 'Bob') is shown. It has many characteristics, including 6 sides to its body, 2 arms, and 2 legs.

Let's also say that this particular object's function is to add up pairs of numbers.

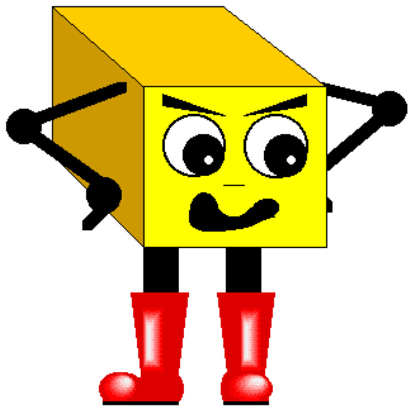


Figure 1: *Bob, a "square" object*

In **Figure 2**, a similar object to **Figure 1** is presented, except that it has 4 legs. Let's call the **Figure 2** object 'Mary.'

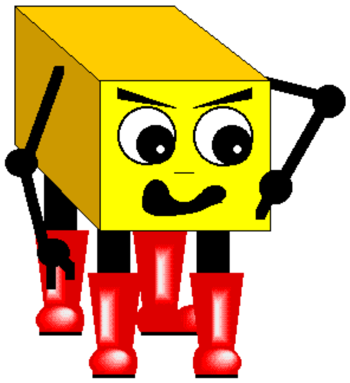
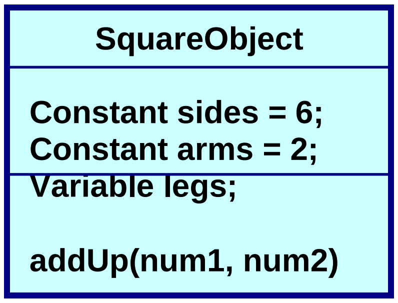


Figure 2: *Mary, another "square" object*

The characteristics or the nature of objects are defined by *classes*. Classes have 3 main functions – they detail the attributes, behaviours and state of an object. In the case of Bob and Mary, they might both belong to a class called SquareObject. The class SquareObject might then be described as shown in **Figure 3**.



```
SquareObject

Constant sides = 6;
Constant arms = 2;
Variable legs;

addUp(num1, num2)
```

Figure 3: *Class diagram for the class* SquareObject

The table presented in **Figure 3** is known as a *class diagram* and is a common diagrammatic technique used in object-oriented software engineering. The diagram is divided horizontally into three parts. At the top is the class name. In the middle, the characteristics associated with objects belonging to this class are presented. These characteristics are called *attributes* or *properties*. At the bottom are listed the functions that objects of this class are intended to perform, and these are called *methods* or *behaviours*. The combination of the attribute values and of the behaviours at a particular point in time gives us the state of an object.

Object-orientation is a mechanism for modeling problems in terms of interacting objects which are, themselves, defined in terms of a *class (or classes)*. Classes consist of attributes and methods that tell us the state of an object at a particular point in time. A class diagram helps designers create objects with all the necessary elements and provides a method of 'viewing' those elements and their properties at a glance.

There is an important distinction between classes and applications. It is this distinction that illustrates the first three of the above-listed features of the object-oriented methodology: *flexibility and adaptability, software reuse and extension, and information hiding*. Classes can be associated with many different applications and can be altered, refined or updated in isolation (*flexibility and adaptability, software reuse and extension*). Furthermore, a programmer working on an application need not be concerned with details of the operation of particular classes as long as he or she knows what data (input) the class needs to perform its function and what data (output) the class will provide (*information hiding*). The fourth feature (*consistent notation and* *integration*) is unique to the OO approach and results in the fifth feature (*reliability*), although this last is, of course, an objective of all software development methodologies.

6. The Application Programmers Interface (API)

When we compile a Java program, code from one or more *packages* may also be included. Packages are files containing Java code which come with the JDK and which facilitate common operations such as input and output. Because these operations are so common there seems little point in writing them over and over, and therefore they are provided as part of the language. (Note: the "packages" concept is common to many programming languages.) Collectively, the preprogrammed classes contained in the packages which are provided with the JDK are called the *Application Programming Interface* or API. The Java API consists of a large collection of classes (several hundred) organised into different packages. Each package thus contains a number of classes.

You can think of this organization as a library system. The Java API is the library, a package is a subsection that contains books relating to a certain topic, such as GUI, and the classes are the individual books. Common Java APIs are things such as a GUI label or a String datatype – things that are used in many different applications. We will be using a common GUI APIs in our programs this week – the JOptionPane.

7. An OO Solution to a Problem

We have already introduced the OO methodology (section V above); in this section we will embellish this introduction to give a much wider overview of the approach. In the object-oriented paradigm a solution to a problem is constructed by defining a set of interacting objects and classes. Objects are defined by what is termed a *class*, we say the objects "belong" to a class, or are *instances* of a class. We can envisage these objects existing in what we might call an *object space*; which in turn exists, together with the class definitions, in something which we might call the *solution space*. The objects and classes inside this space can then be considered to produce various elements (sub-solutions) of the desired end solution. For example, if we wish to determine the result of the following sum:

X = ((1+2) * (3+4)) - 5

we might decide that we require three types of objects:

1. A multiplication object
2. An addition object
3. A subtraction object

To obtain the solution to the above problem, these objects will have to communicate with each other (send messages). Therefore the subtraction object would have to send the sum (1+2)*(3+4) to the multiplication object before carrying out the required subtraction, which in turn would have to send the sums 1+2 and 3+4 to the addition object. Objects will also have to send messages back containing appropriate results.

For all this to happen we need an additional "control" *method* typically contained in what is called an *application* class to start the process off. In Java, this method always has the name "**main()**." The main method is recognized by the Java interpreter ('java' command) as the place where it should start running the code. Therefore, your applications will all start at the main method, and if you delete or rename this method, you will receive an error message.

Figure 4 shows a solution space where three objects of the form described above are displayed:

1. A cubic object that carries out addition operations

2. A cylindrical object that carries out multiplication operations

3. A spherical object that carries out subtraction operations

Each of these objects is defined in a class definition. Note that within this definition, the operations which *instances* of this class can perform are expressed. In the figure, there is only one instance of each class, but there is no reason why there cannot be many instances of the same class. The figure also includes an ApplicationClass which contains a **main()** method to start the solution process off. The solution process then encompasses the objects sending messages or 'talking' to each other so that collectively they can produce the desired result. Note that in theory, any object can 'talk' to any other object as long as the nature of the desired communication is expressed in the class definition. Although in theory the spherical object can talk to the cubic object, there is nothing in the class definition of the spherical object that tells instances of this class how to do so!

8. Objects, Classes and Inheritance

As you see from the above paragraphs, OO programming is founded on the concept of *objects*. Classes describe categories of objects, and this is why we say that a particular object is an *instance* of a class. Classes define the *attributes* (properties or data) that can be associated with an object, and the *methods* (operations or behaviours) that can be applied to those fields. Collectively the fields and methods belonging to a class are referred to as the *members* (features) of that class.

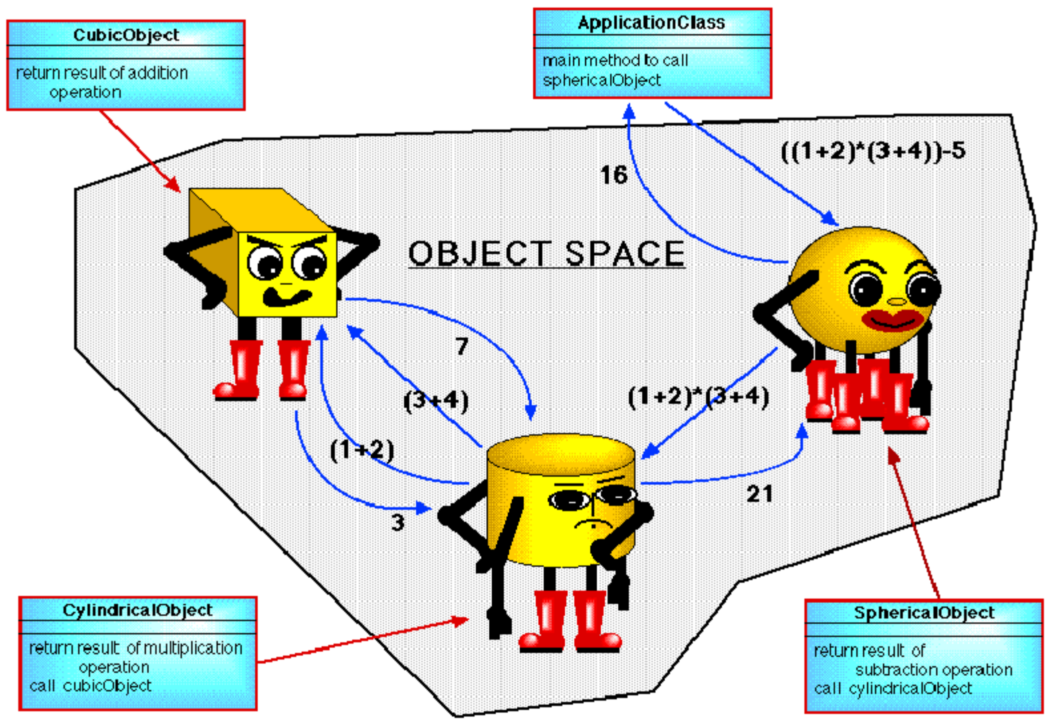


Figure 4: *An object oriented solution to the* X = ((1+2) * (3+4)) - 5 *problem*

Classes are often arranged in a hierarchy whereby *sub-classes* (child classes) *inherit* features from *super-classes* (parent classes). The advantage of inheritance is that we do not constantly have to redefine members which have already been defined. Inheritance is an important feature of OOP and we will cover this topic in-depth in a later seminar.

An example class hierarchy is presented in **Figure 5**. Note that by convention:

1. The arrows in a class diagram always point up the hierarchy toward the super-classes.

2. Class *identifiers* (names) in Java always commence with an upper case letter, and if two or more words are used together in the name, the first letter of each subsequent word is usually capitalized as well. This class identifier MUST be exactly the same as the name of the file. For example, a class named GrandParent must be saved in a file named GrandParent.java.

Where a class inherits from another class anywhere in the hierarchy, an instance of any sub-class in the hierarchy is also a legal instance of all its super-classes and therefore has (at least in principle) all the fields and methods associated with the super-classes available to it.

With that in mind, in **Figure 5**, an instance of the class Child is also an instance of the class Parent and the class GrandParent (but not the classes ParentSibling and ChildSibling). We say that the class child *extends* the class Parent which in turn *extends* the class GrandParent. Alternatively, we sometimes say that a certain sub-class is derived from a super-class.

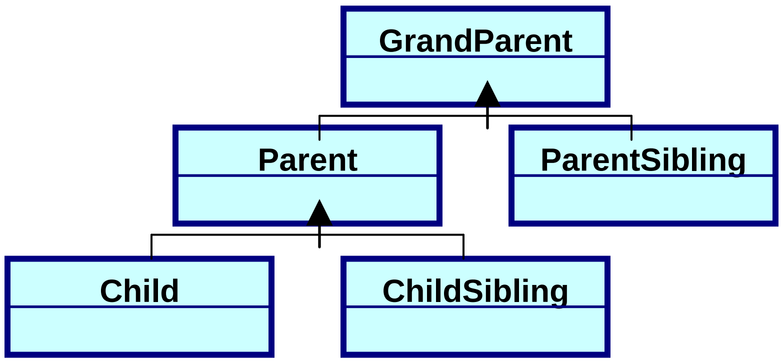


Figure 5: *Class diagram demonstrating a "family tree" class hierarchy*

Three categories of class field membership are identified:

> ***PUBLIC:*** Members which can be accessed from any where in a program (regardless of inheritance).
> ***PRIVATE:*** Members which can only be accessed from within an object, i.e. they cannot be inherited. To access such members from outside the object we must use a public member method.
> ***PROTECTED:*** Private members which are *inherited* by instances of classes *derived* from the current class.

The principle of information hiding in the OO methodology dictates that whenever possible, members of a class should be declared private to protect their contents. However, if members of another class require access to the values of a member attribute or method, these members must be declared public to permit such communication between objects.

9. Constructors

To create an instance of a class (i.e. to create an object) we use a special method called a *constructor.* We say that a constructor creates an *instance* of a class. When this happens, access to the methods defined for the class and a completely new set of attributes (of the form defined for the class) is bundled up with the instance identifier. Consequently we refer to *instance attributes* and *methods*.

Sometimes, given a particular application, we wish to include members in a class for which we do not wish there to be a unique copy for each instance. Such members are called *class members* or *static members.* In this case we refer to *class attributes* and *class methods* as opposed to instance attributes and methods. The term static is used to indicate that such members cannot be copied to form part of an object.

Figure 6 gives a class diagram of a single class called GrandParent (taken from **Figure 5**) which has two attributes and one method which is a constructor. Note that:

1. The first attribute, averageAge, is a class attribute because it is preceded by the modifier static.

2. By default the second attribute, age, is an instance attribute.

3. Both attributes are private members/features, i.e. they can only be accessed from within the class.

4. By convention, attribute names are always started with a lowercase letter. If attribute names are comprised of more than one word, each subsequent word is capitalised.

5. Constructors always have the same name (and the same capitalisation) as the class with which they are associated.

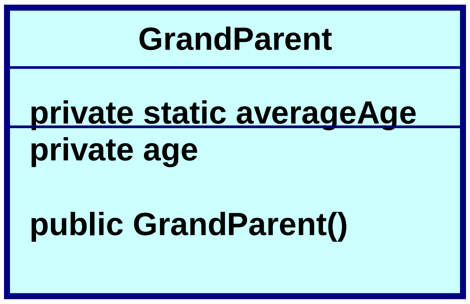


Figure 6: *GrandParent class containing a class attribute, an instance attribute, and a constructor*

If, in some application, we now want to create an instance of this class and call the instance 'Louise,' we would invoke the GrandParent constructor as follows:

GrandParent Louise = new GrandParent();

The keyword new indicates that a new instance of a class is to be created. When this happens, appropriate storage for a completely new set of members, i.e. the attribute age in the example class (but not the *class attribute* averageAge), is set aside for the newly created object. Note that we can have more than one constructor associated with a particular class. Also, a constructor may have one or more *formal parameters* (or *arguments*) associated with it - more on this later.

10. Attributes

An attribute describes a variable or data item associated with a class. For example, the GrandParent class given in **Figure 6** has a class attribute called averageAge and an instance attribute called age. From within a class, the attributes belonging to that class can be accessed directly by referring to them by their name alone (regardless of whether they are private, public or protected members). From outside the class an attribute can only be accessed if it is a public class member using either:

1. The name of an object in the case of an instance attribute, or

2. The name of the class in the case of a class attribute.

We use the dot operator to do this (sometimes referred to as the *membership operator*). In the first case, this is prefixed by an instance identifier (name), and in the second case by a class identifier. Therefore, in the first case where we are referring to the 'Louise' object we created above, we get:

Louise.age

or in the second case where we are referring to the class itself, we get:

Grandparent.averageAge

11. Methods

A method is a named sequence of instructions called *statements*. Methods are used to define the behaviours that we, as programmers, determine we will allow an object to perform. We can identify three types of methods:

1. Constructors (as described above),

2. Instance methods

3. Class methods

where instance methods are associated with a particular instance of a class and class methods with an entire class. **Figure 7** shows a redefinition of the GrandParent class used earlier (**Figure 6**) so that the class includes an instance method outputAge, and a class method outputAverageAge. In this example, the class GrandParent is now allowed by the programmer to output the average age of GrandParent classes as well as output the age of a particular GrandParent instance, or object.

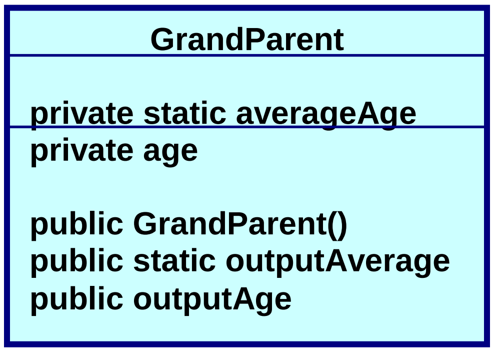


Figure 7: *GrandParent class containing a class method and an instance method*

A. Overloading

It is possible to *overload* method names (i.e. to have two or more methods with the same name in a single class). In this case, however, they must be differentiated by the number and nature of their formal parameters or arguments. One common use of overloading is to have several different constructor methods for the same class, each of which allows an instance of the object to be created in a slightly different manner with pre-set attribute values.

B. Method Calls

As with attributes, from within their class, methods can be referred to by their name alone. When used from outside the class, the method must be a public class member and referred to with either:

1. The name of a class in the case of a class method, or
2. The name of an object in the case of an instance method

Again, we use the dot operator to do this. In the first case the prefix is a class identifier (name), and in the second case it is an instance identifier.

```
GrandParent.outputAverage();
```

or

Louise.outputAge();

The formal parameters (the arguments for the operation) are optional. When we reference a method in this manner we are causing it to execute with respect to the class or instance that it is linked to. The above statements are therefore referred to as *method calls*. The object within which a method is invoked is sometimes referred to as the *receiving object* or the *receiver* (in that it may receive information returned by the method).

12. Anatomy of a Java Class Definition

A Java program consists of a set of one or more classes, many of which will be related to one another in some form of hierarchy. In the program, not every class needs to be related to every other and the types of relationships between classes may vary. In **Figure 8**, a framework for a Java class definition is presented.

```
import PACKAGE_NAME

public class CLASS_NAME{

        <ATTRIBUTE_DEFINITIONS>
        <METHOD_DEFINITIONS>
}
```

The lower case text in **Figure 8** indicates reserved words that have a special meaning. When writing your own programs, different words will be emphasized in different manners. One of the most important things to remember as you program is that Java is case sensitive, and certain words, such as the keywords above, must be written in the manner in which the program expects to see them. The keywords above will always be written in lower case.

Notes with respect to the framework presented in **Figure 8**:

1. We commence by indicating any packages that we might wish to use. For example, if we wished to use APIs from the java.lang package we would write:

    i. **import java.lang.*;**

    The .* indicates that we wish to use or to have access to all the classes defined in this package. Instead we could have defined particular classes that we wished to use, such as:

    ii. **import java.lang.System;**

    Note that the java.lang package is included in every Java program automatically without actually requiring an import statement.

2. The class name must be the same as that associated with the file. If we wished to create a HelloWorld class, then we would have to store the class definition in a file called **HelloWorld.java**. Tradition holds that the first program you write when learning a new programming language always prints "Hello World!" out to the screen and nothing else. This is a great program to try running directly from the DOS window.

3. The combination of the access modifier *public*, the reserved word *class,* and the name of the class is called the *class heading* or the *class signature*.

4. Although we have declared the attribute members before the methods there is no syntactical requirement to do this; however it is good software engineering practice to adopt some sort of conventions like this so as to enhance readability/understandability.

5. A class does not necessarily have to have any attributes but it would be rather pointless if it then also did not have any methods.

13. Anatomy of a Java Method Definition

```
<ACCESS_MODIFIER> RETURN_TYPE METHOD_NAME (ARGUMENTS){

        <LOCAL_ATTRIBUTES>
        <ACTION_CODE>
}
```

Figure 9 gives a similar framework for a Java method to that given in **Figure 8** for a Java class. This is the basic structure of a method showing the placement of keywords and data. In order to fully understand the anatomy of a method, let's begin by examining the main method – the method that starts each Java application. We'll discuss the purpose of the main method in section XV, but for now, we're just going to examine its format in **Figure 10**. Note that:

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

        <LOCAL_ATTRIBUTES>
        <ACTION_CODE>
}
```

Figure 10: *The Main Method Signature*

1. Access modifiers are keywords such as **public**, **private** and **protected**. These modifiers determine the access that your class and the others in your application have to the method. For example, if a method is a public method, any class from any application may 'call,' or have access to, that method. If a method is private, it may only be used within the class where the method is defined. The main method will always be declared as public.

2. The keyword **static** means that a method belongs to the class, not to its instances. This concept was discussed in section IX. The main method is where the application starts, so only a class-level method is needed. You might think of this as being similar to the way a word processing program works. When you start the program, you only need to start it once. Once it is running, you can create as many documents (or objects) as you'd like, but you don't need to start another environment to create and use these objects.

3. Every method in Java must have a *return type*. This describes the nature of the data item that is to be returned by the method. For example, we might define a method which adds two integers (int datatypes). Once we've performed the calculation, we presumably need to get the result back from the method, so we will tell the method to return an

integer value. The signature of this method would then have an int as the return type in the method signature. In cases where we do not wish to return anything, we use the keyword **void**. In the main method, all we're doing is instructing the program to perform a set of steps to run the program – we're not generating a result or return value that we need to send anywhere else. Therefore, the main method is described as a void method.

4. Names in computer languages are called *identifiers*. An identifier in Java consists of letters, underscores and digits commencing with a letter. Java is a *case sensitive* language. For example, the method identifier myMethod is not the same as myMETHOD. Further, unlike some languages, Java does not allow spaces. Instead, programmers use capital letters for inner words. Alternatively, the underscore character may be used, but this is not standard practice and should be avoided in this class. The name 'main' is always the method the JVC looks for when running a program. It should never be changed or capitalized or your compiler will throw an error.

5. You should always use meaningful identifiers as this enhances understandability. You should not be afraid of using long identifiers, although anything with more than 32 characters is probably overdoing it.

6. Often we need to pass data to a method to allow it to perform its function. For example, in the case of a method which adds two integers, we would pass in the integers as arguments to the method. Think of arguments as the data the method needs in order to perform its function. Unless the addition method receives two numbers, it doesn't have what it needs to perform the addition calculation. In the main method, the arguments are in the parentheses – 'String args[ ]' – and should not be changed. The square brackets indicate that the string element named 'args' is an array – we'll discuss arrays at length in a subsequent seminar.

7. This first line of the definition (comprising access modifiers, the method's return type, the method's name and the argument list) is called the *signature* of the method (some authors use the term *method heading*). The part contained between the curly brackets is termed the *body*.

8. The body contains definitions of any further *local* data items (other than the arguments and class-level attributes) that might be required and the individual code statements. A method is not required to include local data items, but there would be little point to the method if it also did not include any code statements. For example, if we created a method that was simply used to double any number that was sent to it, we would need to declare a local integer attribute to hold the value of 2 so that we could perform the calculation within the method.

14. Output

In this section, we will describe the mechanisms provided by Java to achieve data output. In order for a program to do anything it must receive some input data via an input stream and produce some output data to be sent to the output stream. The input may simply be a signal to run the program (with no actual input data), but it would be pointless to write a program which did not produce any output. Therefore, the simplest Java program we can write produces a single output statement. We have mentioned that traditionally, the output from a first program written when learning a new programming language displays the phrase "Hello World!" on the screen. We, too, will abide by this tradition!

When initially running the "Hello World!" program, do so from your DOS window. You should type the code into a basic text program such as Notepad, save the file in the form HelloWorld.java, and run the javac and java commands from the DOS window. You will see the output right there in the DOS window under your java command line. You should then create the same program in the NetBeans environment. Open a new file, write your code, save the file with the correct extension, and then from the toolbar, click the run arrow. Your program will run and you will see the output in the NetBeans output window.

To facilitate output, Java provides a number of classes. One of these is called PrintStream. This class contains (amongst other things) the methods **print()** and **println()**. The first method, **print()**, takes one *formal parameter* which it outputs to an *output stream* (e.g. the screen, secondary storage, etc.). The second method, **println()**, is used in an identical manner except that it automatically includes a new line character or return at the end of the output. Note that in both cases the formal parameter must be a sequence of one or more characters. Such a sequence is called a *string* and is indicated by enclosing the characters with double quotes ("). Strings are discussed in further detail below and in a later seminar.

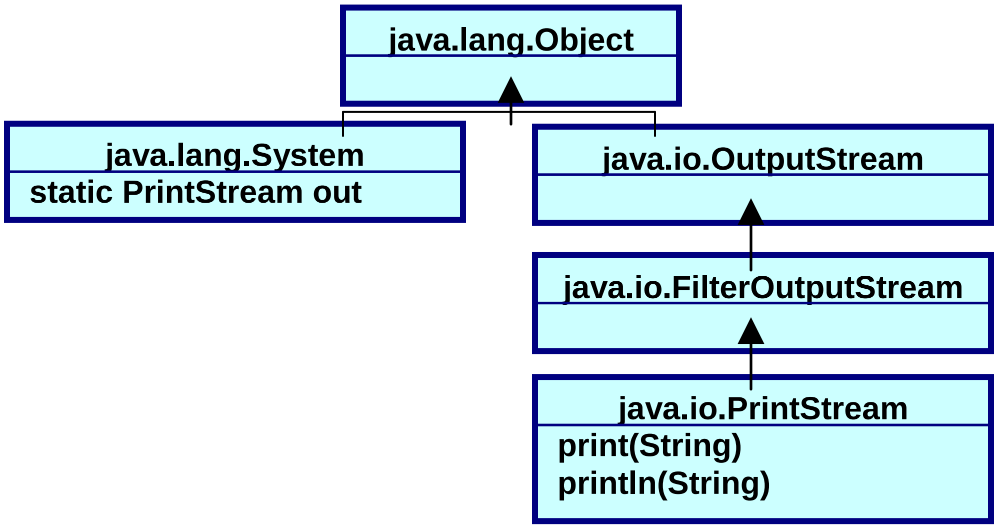


Figure 11: *Class diagram showing API classes associated with simple screen output*

A class diagram indicating the relationship the PrintStream class has with some other significant classes, at least in the context of output, is given in **Figure 11**.

Both **print()** and **println()** are instance methods and therefore require an appropriate object of the *type* PrintStream. In addition, to use either method, we must tell Java where to direct the output stream. Java provides us with a special screen output object to facilitate output to your computer screen (called **out**) which is contained within the class System. Because **out** is an instance of the PrintStream class, it carries with it all the paraphernalia associated with that class (i.e. access to the methods **print()** and **println()** and so on). The class System is contained in the package java.lang which (remember) is always compiled into every Java program.

If we wish to output 'Hello World!' to the screen we might include the following statement in a piece of Java code:

```
System.out.print("Hello World!");
```

or

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

Note the semi-colon character at the end of the lines. Every Java program statement must end with a '**;**' character in order to be read in the correct sequence blocks by the compiler. Note also that **println()**, when called without any arguments, will simply cause a carriage return (new line character or enter) to be passed to the output stream without any other output.

A. String Literals

Any sequence of characters enclosed in quotes, such as "Hello World," is called a *string literal* or simply a *string*. If we wish to include a quote character (") within a string we must prefix it with a backslash character (\). This is called the *escape character*. For example:

```
        System.out.print("Hello \"Friend\".");
```

would produce the output

```
      Hello "Friend”.
```

Without the escape characters, the Java compiler would recognise this as two strings, **Hello** and **.**, separated by the "operator" Friend. This is bound to confuse the compiler! Combinations of back slashes and special characters such as the quote character are sometimes referred to as *escape sequences*. The concept of escape sequences is common to many languages, for example C and C++. Further examples of escape sequences include '\n' which indicates a carriage return, and '\t' which indicates a tabulation ("tab") marker. So instead of writing

```
      System.out.println("Hello World Again");
```

we could have written:

```
        System.out.print("Hello World Again\n");
```

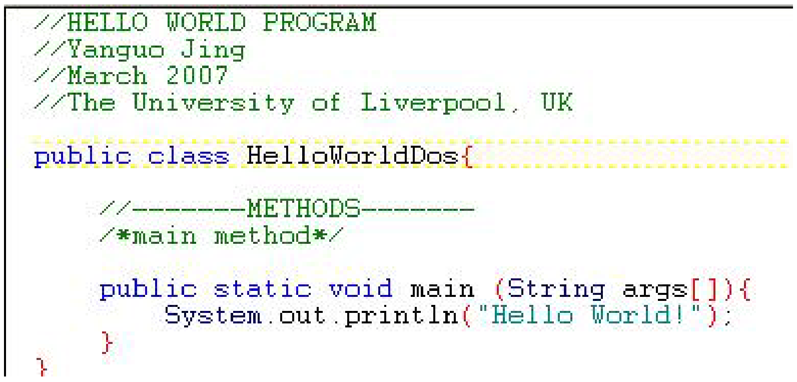
Sometimes when you are programming, a string will be too long to fit on a single line in the editor window. If we simply press the return key and continue typing in the next line we will have introduced a newline *control character* into the string which the editor can understand but the Java compiler would have problems with. One option would be to use a sequence of print methods. Alternatively we can *concatenate* a sequence of strings together using the '+' concatenation operator. For example:

```
System.out.print(“This is a very long string literal ” +
   “that unfortunately is so wide it ” +
     “disappears off the end of the screen.  ” +
   “However we can avoid this undesirable ” +
   “feature by splitting the string up into a ” +
   "sequence of window wide sub-strings and ” +
   “link them together using the Java \"\+\" ” +
   “concatenation operator.\n”);
```

Note the use of the escape character in \"\+\".

15. First Application Program (“Hello World!”)

A. DOS Window Version



```
//HELLO WORLD PROGRAM
//Yanguo Jing
//March 2007
//The University of Liverpool, UK

public class HelloWorldDos{

    //--------METHODS--------
    /*main method*/

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

Figure 12: *Hello World! Java program*
We are now in a position to put all of the above together and write our first Java application program (**Figure 12**). Type in the code above exactly as it is shown, with all the capitalization and punctuation just as it is above. Be sure to note where the brackets open and close. These brackets denote segments of your code. The first bracket after the class signature line is closed by the last bracket at the bottom of the file – this means that all the contents therein are part of the class code. The second pair of brackets starts at the end of the main method signature line and closes after the output statement, meaning that all the code inside is part of the main method. Once you have typed the code, save the file as **HelloWorldDos.java**. Next, go to the DOS prompt and type in:

javac HelloWorldDos.java

This compiles the file and creates the bytecode file, HelloWorld.class, in the same directory where you saved the .java file. Now that the file has been compiled, we can run this code using the Java interpreter by typing:

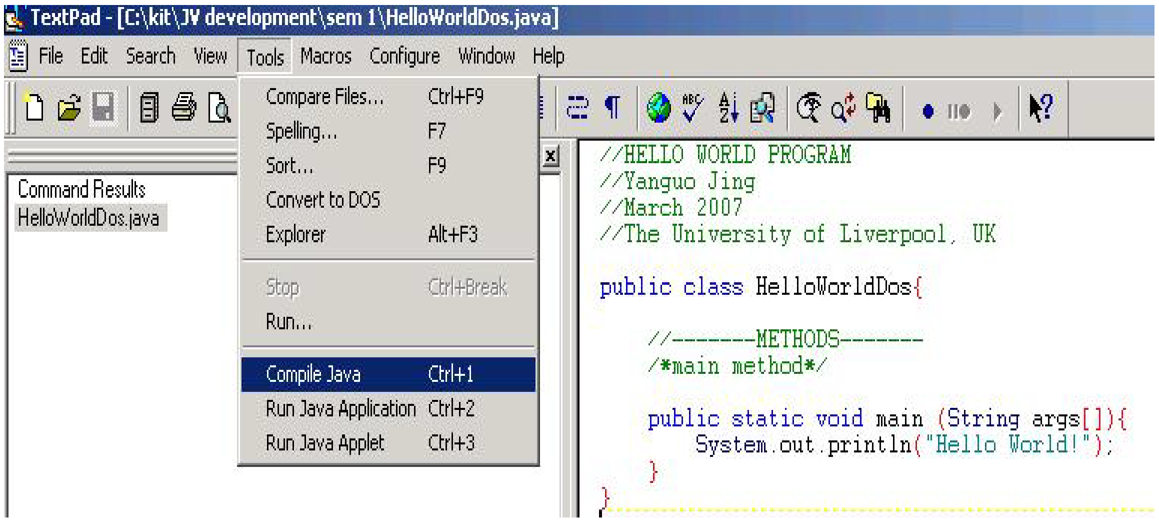
java HelloWorldDos

This will produce:

Hello World!

as output on your DOS prompt window.

B. TextPad version. Run TextPad, open the HelloWorldDos.java program. Choose the menu “Tools=>Compile Java” or “Ctrl+1” to compile the java program. Choose the menu “Tools=>Run Java Applications” or “Ctrl +2” to run the program.



```
//HELLO WORLD PROGRAM
//Yanguo Jing
//March 2007
//The University of Liverpool, UK

public class HelloWorldDos{

    //-------METHODS-------
    /*main method*/

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

Figure 13: *TextPad screen shot with HelloWorldDos*

Note that:

1. Lines of code commencing with '//' or '/*' are *comment* lines. These are ignored by the compiler. There are two standard types of comments available in Java:

    a. *Single line comment*: denoted by '//' (also found in C++, and imperative languages such as Ada). The compiler does not read anything beyond the // marks. For example:

    // This is a one line comment.

    b. *Multiple line comments*: indicated by a /* at the start and a */ at the end. This approach is also a feature of languages such as C (and C++). For example:

    /* This is a block of comments.
    Many programmers consider
    this to be much more elegant
    than a sequence of single
    line comments. */

    It is good programming practice to include comments in your code, so in the above case we commence with the name of the program (class), the author, the date when it was written, and where it was written. This information is all extremely useful when it comes to maintaining the code.

2. The program makes no use of features of external classes. Access to the **print()** and **println()** methods are encapsulated into the **out** instance and is included in the java.lang package which is compiled into every Java program.

3. The class name must be the same as the source code file in which it is stored (but with the suffix java), so the above text file should be saved in a file called **HelloWorld.java**. Care should be taken to choose meaningful names for classes so that a clear indication of the functionality of classes can be obtained simply by inspection of class names.

4. There is one method called **main()**. Every Java application program must have a method called **main()**, because this is the method from which the Java interpreter starts processing the code.

5. The **main()** method always has the modifiers **static** and **public** to indicate that it is a class method and should not be associated with individual objects, and to show that it can be accessed from anywhere within the program.

6. Similarly, the return type of the **main()** method must always be **void**. The top level method cannot return anything because it doesn't have anywhere to return data to, nor does it have data to return! The **main()** method is simply the starting place for the program to start running, so it doesn't have to produce any return values to the program.

7. In the case of a **main()** method, we must provide information concerning the formal parameters for the method (unlike any other methods this information cannot simply be omitted). However, there is a slight complication in that the arguments cannot be passed to main from another method but must be provided when the program is invoked - we refer to such arguments as *command line arguments*. This is a non-trivial concept and for the time being we will have to accept that this is indicated using String args[ ].

C. Note on Layout

As far as the Java compiler is concerned, layout, in terms of new lines and spaces (with the exception of strings) does not matter provided that the source code is syntactically correct. However, from a software engineering perspective a well-laid out program is much easier to read and understand than one that is not well-laid out. The following version of the code presented in **Figure 12** will compile and run, although it is very difficult to read:

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

It is therefore a good idea to write programs neatly using appropriate indenting, comments and white space so as to enhance understandability.

"If something looks good it probably is good!"

You can find an excellent guide to coding standards at the following URL:

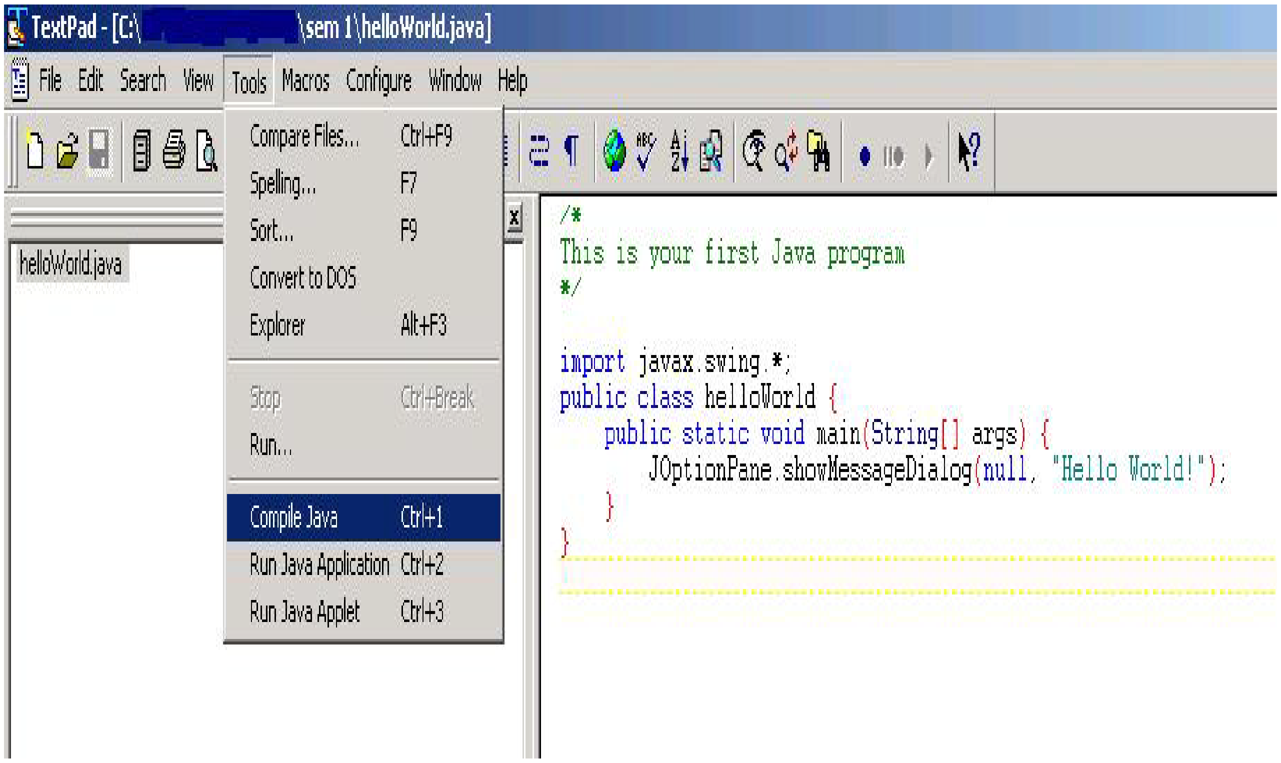
http://www.ambysoft.com/javaCodingStandards.pdf

16. Introductory GUI - JOptionPanes

While presenting output to the DOS window or output window can allow programmers to hone their skills and practice writing code in a ‘pure’ environment, most of today’s programs are based on GUI due to its easy presentation style to application users. Graphic programs capture the user’s interest more so than ‘flat’ command line programs, so while we are learning to code in Java, we will also be learning about how best to present information to our users through the use of graphics.

To start, we are going to slightly change our “Hello World!” program so that it is presented in graphic format. We will accomplish this presentation through the use of one of the simplest graphic elements, the JOptionPane. The JOptionPane is part of a package of Java APIs called the Swing package. These are elements that have been created to look like graphic elements we most often encounter, such as text boxes, command buttons, etc. The JOptionPane is a pop-up box that contains several methods we will use as we go through the course. The first one we want to display will simply flash the message "Hello World!" in much the same way as our other two programs have done.

Open you textPad (a text editor), create a new file, and enter the code below (**Figure 17**). Once you have changed the code, click the menu "Tools"=>"Compile Java" or hit the "CTRL+1" keys to compile the java program, and select the menu "Tools"=>"Run Java application" or hit the "CTRL+2" keys to



```
/*
This is your first Java program
*/

import javax.swing.*;
public class helloWorld {
    public static void main(String[] args) {
        JOptionPane.showMessageDialog(null, "Hello World!");
    }
}
```

Figure 17: *Hello World! Java program with JOptionPane*

run the application and you should see a message box pop up on your screen with the "Hello World!" message inside (**Figure 18**).

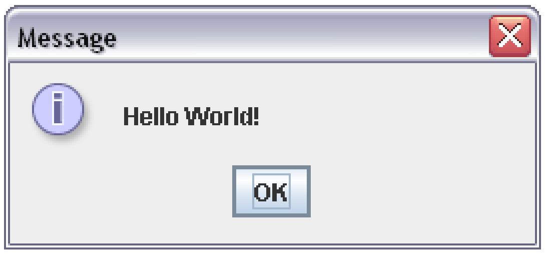


Figure 18: *Hello World! JOptionPane*

The important things to note here are that:

1. There is an import statement used. The **import** keyword gives our class access to the **javax.swing** package, or library. The Swing package is where the API for the JOptionPane is found. By adding an asterisk (*****) after the **javax.swing** package, we're saying that we would like access to all of the APIs in the Swing package – not just the JOptionPane. Alternately, we could specify that we only want access to that API by

Chapter "Before you begin"
Chapter 1 – 1.9, 1.10,
Chapter 2 – 2.1-2.3
Chapter 3 – 3.1-3.5, 3.10

writing

import javax.swing.JOptionPane;

but most often, using the asterisk is the best way to go since you will likely be using several elements from the Swing package in a graphical program.

2. The method showMessageDialog from the JOptionPane class requires two arguments in order to run. The first argument simply tells the application where on the screen to display the JOptionPane. When this argument is null, the box will show in the middle of the screen. The second argument is the string that we wish to display. Note that you must still use the double quotes around the text in order for the string to be read properly.

JOptionPanes can do a variety of tasks – they can display messages, accept text strings, display text in the title bar, and can use various built-in symbols to convey various graphic messages to the user. We will cover JOptionPanes more in depth as we move through this week's seminar.

Reading List:

Read lecture notes and the following from the textbook:

Chapter 1 –1.9, 1.10
Chapter 2 – 2.1-2.3
Chapter 3 – 3.1-3.5, 3.10
Chapter 11 – 11.1, 11.2