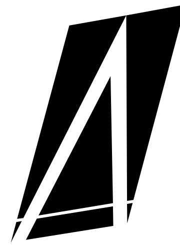


UNIT NO: D76 V35

UNIT TITLE:
Software Development:
Object Oriented Programming

Workbook 1 of 8

Outcome 1 (part) – Produce an OO specification to describe a set of classes of a specified problem

Outcome 3 (part) – Implement a solution from an OO specification demonstrating proficiency in using OO methods

Awarded for excellence

Computing (TN3)
Engineering, Computing and Business Studies

Welcome to OOP with Java!

The aim of this course is to provide a comprehensive introduction to the concept of Object Oriented Programming. There are many programming languages that are OOP-based, for example SmallTalk and C++, but for this particular course we will be using Java.

By the end of the series of workbooks you will have:-
Understood the key concepts of *object oriented programming (OOP)*
Learned about *object modelling*(OM) and *object analysis*(OA)
Learned some of the syntax and semantics of the Java language
Written a fully working OO program based on a set of user specifications
Become familiar with a Java environment and used some of the Java library classes.

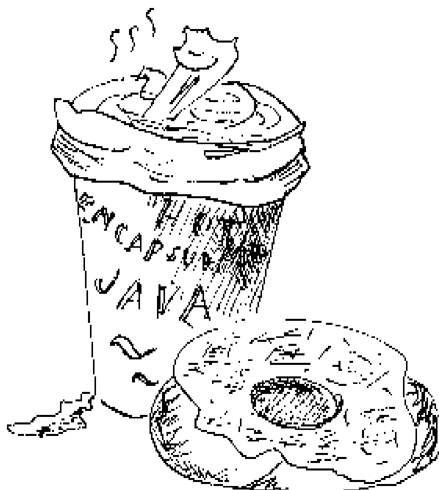
Object Design Methodology.

For all the analysis and design diagrams in these books, we will be using Unified Modelling Language (UML). This is a special notation, specifically designed for object-oriented systems or applications, from top-level down to detailed designs. This standard was first developed in 1997 and is now extensively used in the design and programming community. There are also many software programs (CASE tools) which support it's use, such as Rational Rose (see the website at www.rational.com).

Why Java?…

Java (originally called *Oak,* until somebody discovered that another company used "Oak" as *their* trademark), was developed at Sun Microsystems and launched in the autumn of 1995. The designers of Java had three main goals:-

- to provide program code which would run on many different platforms
- to produce robust programs with few bugs
- to create a language which would be easy to learn, use and develop



Because many of the original Java developers came from a background of C++, there are quite a lot of superficial similarities between the two languages.

There may or may not be truth in the rumour that the aforementioned developers thought of the name "Java" on a visit to the local Starbucks coffee house.

Java was originally intended to be a language for programming embedded applications, but because of its cross-platform capabilities, it soon became popular as a programming language for the World Wide Web. Java is now best known for programming *applets,* or mini-applications, which run within the context of a larger application, such as a network browser. This course concentrates on Java as a language for stand-alone applications.

If you want to find out more about the design of the Java programming language, you can read the Java "White Paper", which can be found on the http://java.sun.com/ site. The paper was written by the original designers of the language and explains their aims and objectives.

How Java Works.

Normally a programming language compiler is designed to compile programs into an executable format, which will only run on a specific type of machine. A Pascal compiler designed for a PC would not normally run on an iMac, for example.

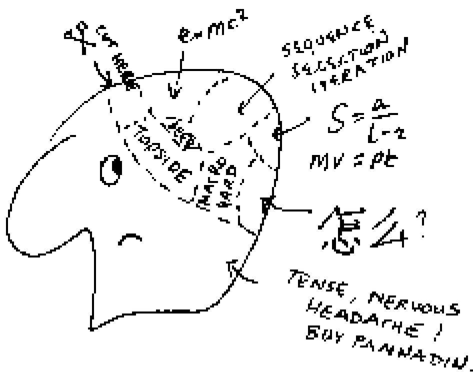
Programs are usually copied across the Internet and loaded for execution on a local client machine that could be of any other hardware platform. Java gets round this discrepancy in an ingenious manner - it compiles its programs into a *bytecode* format, which acts like a sort of universal assembly language. The Java program is then loaded locally (from the hard disk) or externally (across the Internet) in its bytecode format. Note that this bytecode is *not* executed directly - it now has to be *interpreted* by the local client machine.

The *interpreter* (the program which executes the bytecode) translates each instruction into an equivalent sequence of local machine instructions. The interpreter and underlying hardware are collectively known as the *Java Virtual Machine* because together they create the illusion of a different machine architecture - one that can execute bytecode instructions.

The Java compile-execute cycle therefore differs slightly from a "normal" compile-execute cycle, thanks to this extra bytecode middle-man!

- The Java source code must be compiled into bytecode format using the *javac* compiler.
- The bytecode must be loaded, either from a local machine, or a remote server on the Internet.
- The bytecode is now verified and interpreted in order to execute the Java applet or program. This can either be done explicitly by running the Java Virtual Machine or implicitly by Internet Explorer or Netscape Navigator (which runs the Java Virtual Machine "transparently").

Java plays Mind Tricks!



Approaching an Object Oriented design is entirely different from the Procedural languages you may have already encountered. Unfortunately the only way to become proficient in this new way of thinking is by hard graft – and making sure that you have mastered the basics before attempting anything too complicated!

With this in mind, we will use this first workbook to learn some simple Object Oriented design, and then some Java syntax in order to write some very basic stand-alone programs. Our programs will, of course, become more complex (and therefore more realistic) as the series progresses.

Why Object Orientation?

One of the major advantages of object technology is that the system or program can model real-life abstractions in a realistic way - in other words, the design mimics the real world. An OO model can also serve as a model of the business - the systems analyst and software developer are both working to the same model. This is far preferable to a business model that is ignored by software developers, and a software design which ends up bearing no relation to the architecture of the business!

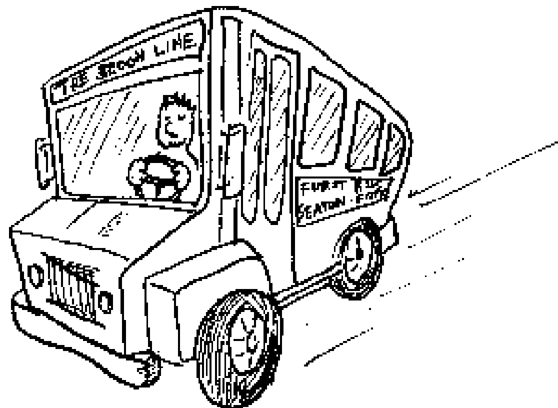
OO systems also lend themselves ideally to prototyping - this could be anything from an interface mock-up to a rough version of the entire system. The biggest advantage of OO systems when prototyping is that large chunks of the language or system are reusable - existing systems can be taken and have any special processing added, to give the required result. Encapsulation also helps when designing prototypes - we will find out more about these concepts later.

Programming in an Object Oriented Environment.

The world consists of many objects, most of which manipulate or interact with other objects or data. In an OO environment, we will think of our programs as consisting of a number of elements, or objects.

A bus is an object that transports people (which are also objects) to different locations. The bus can change its speed, direction, route (and even colour, at least in Aberdeen!) in order to accomplish this. You could think of the ignition key, wheel, accelerator etc. as functions, because these all directly manipulate the data values of the bus.

As a passenger in this bus, however, you won't particularly care *how* these things work. All you need to know is that the bus will go and stop, more or less where you want it to; you know how to choose the correct bus, how to get on it, and how to stop it by pressing the bell. You would also expect to find the same functions and attributes on any other bus – an individual bus is therefore a *bus object,* with unique bus properties.

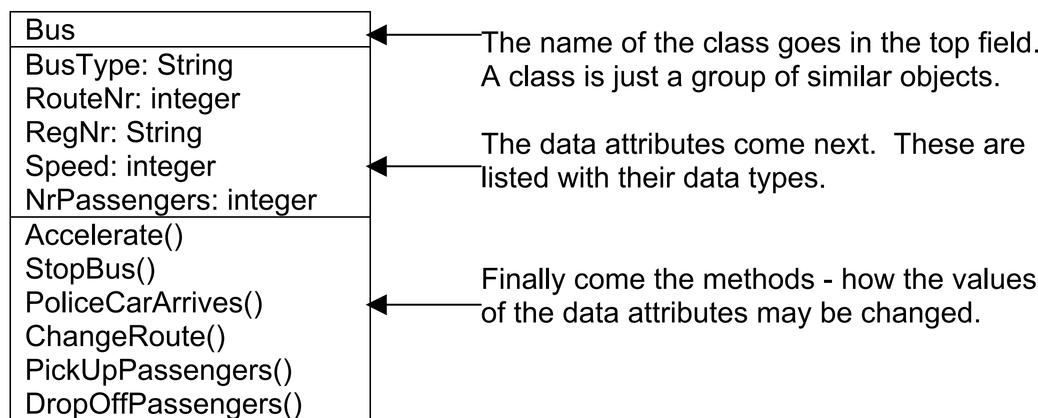


Most popular software applications are written for a GUI environment, and this is a good example of how this "Object Orientation" works in practice. You do not think about how a Window works, or what it's private data fields are; you just have to know how to call the various functions in order to make that window operate. These could be – resizing, minimising, hiding, closing etc. and so long as you know what these functions are, and how to call them, you can efficiently use that window without having to worry about the internal workings.

Both the bus and window examples above, demonstrate one of the fundamental concepts of Object Oriented Programming – the concept of **encapsulation**, or "data hiding", which we have already mentioned, and will examine in greater detail later. In the meantime, we will begin by looking at some very simple Java syntax, in order to get you started.

Defining an Object with Java.

When dealing with an object, we have to think about the attributes, or data values, that are attached to that object. In describing a bus, we could include values like bus-type for single or double decker, route number, registration number, whether it is stopped or moving, and so on. These attributes will be the same for every bus, although the individual values for each particular bus may change. We can graphically depict a bus like this:-



The only way that the data values can alter is by calling one of the functions, or *methods,* in order to make the required changes. For example, the speed of the bus may be currently 25 - calling *StopBus()* would make this change to zero. Similarly, if there are 10 passengers on the bus, calling *PickUpPassengers()* or *DropOffPassengers()* could make this number change.

Getting Started - Some Java Environments.

We will be working in two basic kinds of environment when writing our Java for this course - the Sun OS 5.7 Unix-based system on the College server, and a PC- or iMac- based environment, if you wish to practice at home.

The College Option

Using your usual name and password, log on to UNIX by either Telnet or X-Windows. The first thing to do is to create a new directory to hold all your Java stuff. **It is also advisable to create a new sub-directory for every separate Java program you are about to write.** This is because our Java programs will eventually consist of a number of modules, or *classes*, and it will save a lot of trouble later if these are all kept together!

Write your Java program using the *vi* or text editor, saving it with a *.java* extension.

To **compile** the program or class, use the *javac* compiler command:-

```
javac MyProgram.java
```

Note that the compiler may take a few moments to complete. Once the program compiles correctly, look in the directory and you should see your source-code has been translated into a bytecode class - e.g. `MyProgram.class.` We are now ready to **run** the program on the JVM, using the *java* command:-

```
java MyProgram
```

and with a little luck, your program should be up and running!

The Home Option

The first thing to do is to download the Java Development Kit from http://java.sun.com/webapps/download/DisplayLinks - choose Download j2sdk-1_4_2_01-windows-i586-iftw.exe . This is a small 300kB program which, when run, installs the latest version of the development tools. (Your computer has to be connected to the Internet for this to work – alternatively, there is an Offline Installation version, but be warned, it's over 45mB!). It will install:-

- *javac* the Java compiler
- *java* the Java interpreter which turns your local host machine into the Java Virtual Machine and executes the bytecode
- an *appletviewer* - sort of a mini web browser which can be used to test Java applets
- *jdb* the Java debugger
- *javadoc,* a tool which will automatically scan Java source code and extract comments. It then puts these comments in an HTML page that can be viewed in a regular browser.

Once you have run the *Setup* files and re-booted, you can use any text editor to write your Java programs. (Notepad is fine – avoid Word, because it formats things and that just confuses the compiler). **To compile, first make sure that your *.java* file is fairly near the root directory -** the Desktop is fine - then go down to the Start bar and open up an MS-DOS (Command) prompt window. Change directory until you are in the same place as your *.java* file, then use the *javac* and *java* commands as described above.

(If your compiler does not behave first time around, you may have to manually set paths on your computer. See http://java.sun.com/j2se/1.4.2/install-windows.html for full instructions.)

RTFM!

Here are some useful tips when writing your own code. You should refer back to this page regularly, and it applies to all programming disciplines – not just OOPs and Java.

1. KISS - keep it simple, sunshine! Until you have your basic code working, do not try adding any fancy bits to your methods, or all-singing, all-dancing methods to your classes. When writing a new method, just a simple

```
System.out.println ("This is a test for my mega fantastic method");
```

in the method body will be fine. If adding a whole new class, an empty constructor and a simple "test" method as above will be enough to get it running. Leave everything else out at this stage. By this approach, you are less likely to create coding errors that may be difficult to locate. You may well be butchering a perfectly correct method, when another bit of code somewhere else is actually at fault!

2. Create stub methods and classes wherever required. If your code refers to another class, or method in that class, make sure that at least a stub exists for it. Your class can then call that method so that you can test the overall program and ensure that the basic code works. Only when you have the code running, should you start adding bits to your methods!

3. Make changes SLOWLY. Make ten changes, save a class, compile it and encounter an error - you will have to look in at least ten places to try and find it. If you make only one change at a time, the part that caused the error will be a lot easier to locate. This may seem a very slow way of doing things, but it will pay off in the long run.

4. When given sample code, copy it EXACTLY. DO NOT make any changes to it until you have run it first and verified that it actually works. If you don't do this, you won't know whether it is your adaptations that have caused the problem, or if the sample code was faulty to begin with! (Even the “experts” sometimes make mistakes).

5. Take regular breaks. If it all gets too much and you are spending ages trying to find one daft error, get up from your machine and go for a walk round the block. Then come back and look at your code again. Sometimes detaching yourself from the problem, even for a few moments, will help you find the solution.

A First Program.

Create a new sub-directory to hold your first program, then open up a new file called *Welcome.java.* **Note that Java is case-sensitive and the filename must be spelled exactly as shown.** Either type in, or cut and paste, the following source code into the file.

```
// Our First Program - Welcome.java
// Compile it by typing - javac Welcome.java
// Execute it by typing - java Welcome
// The interpreter now looks for the Welcome.class file
// DON'T type - java Welcome.class (or else the interpreter
//      will try looking for Welcome.class.class !!!)

public class Welcome
{
        public static void main(String[] args)
        {
              System.out.println("Hello Good Evening and Welcome!") ;
        }
}
```

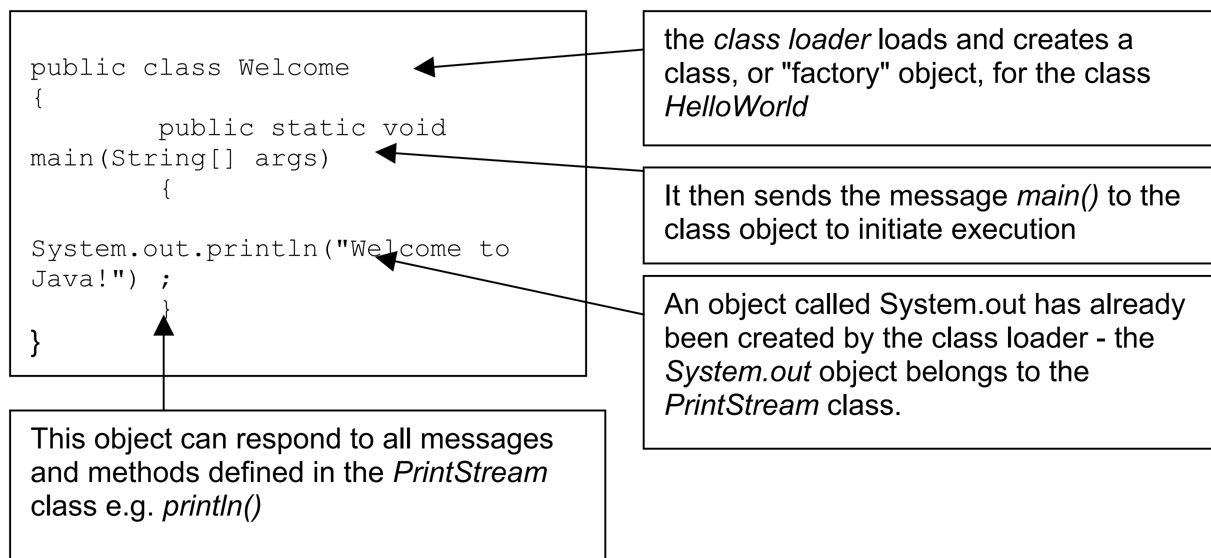
Save and compile the file as demonstrated above. Note the presence of a new file called *Welcome.class* in your current directory. This file contains the bytecode generated by the Java compiler.

Now run the compiled program on the JVM by typing the *java* command.

```
java Welcome
```

This executes the bytecode contained in *Welcome.class.*

What is actually happening here?…



Do not worry if you do not understand any of the above terminology - we will be examining classes, objects, attributes and methods in much greater detail in later workbooks.

Exercise 1.1

Edit the Java source code and modify the program to display your name (instead of the 'Welcome′ message). Recompile and check that the program executes successfully.

Primitive Data.

The Java language has a wide source of *Primitive Data Types;* as well as the usual integers, characters and real numbers found in other languages, there is a selection of others, depending on the size of the value required. Java is very conscious about the amount of memory something is likely to occupy!

The full set of Java Primitive Data Types are as follows:-

Primitive data type	Data type contains:-	Min and Max Values	Default value	Size (bits)
Byte	Signed integer	-128 to 127	0	8
Short	Signed integer	-32768 to 32767	0	16
Int	Signed integer	-2147483648 to 2147483647	0	32
Long	Signed integer	-9223372036854775808 to 9223372036854775808	-	64
Float	IEEE754.floating point	+/- 3.40282347E to +/- 1.402398	0.0	32
Double	IEEE754.floating point	+/- 1.79769313486231570E+308 to +/- 4.94065645841246544E-324	0.0	64
Boolean	True or false	Not applicable	False	1
Char	Unicode character	\u0000 to \uFFFF	\u0000	16

Note that this table does not include a data type equivalent to a “string”. This is because in Java, a String is actually an *object* in a *String* class and has to be accessed by an appropriate method. (We will be looking at this later, so don’t panic!).

Here are some wee exercises to get you used to dealing with primitive data types.

Exercise 1.2 - Adding.java

This file *Adding.java* is a little example of adding a couple of signed integers. Either type in, or cut-and-paste, the following code as before; then compile and run the program.

```
public class Adding
{
        public static void main(String[] args)
        {
                int x ; // declare variable x as an int
                int y ; // declare variable y as an int
                x = 34 ; // lets assign it a value
                y = 12 ; // and here as well
                System.out.print("x = ") ;
                System.out.println(x) ;
                System.out.print("y = ") ;
                System.out.println(y) ;
                System.out.print("x + y = ") ;
                System.out.println(x + y) ; // does the addition
        }
}
```

Our two variables *x* and *y* only have a scope which is valid for the *main()* method; if we were to introduce another class method, the new method would **not** be able to access *x* or *y*. (We look at the scope of methods later, so if you didn't understand that just yet, don't worry). We have declared *x* and *y* as *int*s, which means that these variables can hold *integer* values.

Some More Java Syntax - Arithmetic Time!

As well as using the + symbol to add numbers in Java, we can also use the following.

- - subtraction
- * multiplication
- / division
- % remainder on division (modulus)

Exercise 1.3
Amend *Adding.java* by adding extra lines so that *x* and *y* are also subtracted from each other, multiplied by each other and divided by each other.

Floating Point Numbers.

We will now go on to experiment with floating point, or *real,* numbers. Java provides us with two data types – the *float* and the *double*. These two types occupy different amounts of memory, and when we declare the variables we must append the letter F or D to them to specify which one we wish to use. See the program below for a practical demonstration.

Exercise 1.4 – RealNumbers.java

```
public class RealNumbers
{
        public static void main(String[] args)
        {
                float x, y ; // declare local variables
                x = 3.4F ; // assign value (note the F)
                y = 1.2F ; // assign value
                System.out.print("x = ") ;
                System.out.println(x) ;
                System.out.print("y = ") ;
                System.out.println(y) ;
                System.out.print("x + y = ") ;
                System.out.println(x + y) ;
      }
}
```

Exercise 1.5

Rewrite the above and run it using *double* values in place of floats. Note that
where no letter is appended to a floating point constant, it is assumed to be a *double* by default.

Exercise 1.6

Modify the program so that it tries to divide a *float* value by zero. What happens?

Here Comes the Bus again!

In order to create a Bus object in Java, we must first make up a template, or *class,* to give us the framework for building our bus. The following is a simple Bus class, which can be used as a template to create many different bus objects.

Start by comparing the source code on the following page to the Bus diagram on page 5, and try to match the lines of code with the design.

```
public class Bus
//define the Bus class and make it public
{

   //sets up some Bus data variables
   String theBusType = new String ();
   int theRouteNumber;
   String theRegNumber = new String ();
   int theSpeed;
   int thePassengers;

   //a constructor method - more about this later
   public Bus()
   {
   }

   //now come the "get" methods, one for each attribute, which return
   //the current values when we need to look at them.
   //More about these later….

   public String getBusType()
   {
      return theBusType;
   }

   public String getRegNumber()
   {
      return theRegNumber;
   }

   public int getRouteNumber()
   {
      return theRouteNumber;
   }

   public int getSpeed()
   {
      return theSpeed;
   }

   public int getPassengers()
   {
      return thePassengers;
   }

   //next come the "set" methods, one for each attribute, which
   //set up the current values - we won't use these for the moment, //but they
   will be needed later on.

   public void setBusType(String type)
   {
      theBusType = type;
   }

   public void setRegNumber(String regnumber)
   {
      theRegNumber = regnumber;
   }
```

```
   public void setRouteNumber(int newRoute)
   {
      theRouteNumber = newRoute;
   }

   public void setSpeed(int newSpeed)
   {
      theSpeed = newSpeed;
   }

   public void setPassengers(int numPassengers)
   {
      thePassengers = numPassengers;
   }

   //now come the "special" methods, which are used
   //to alter the individual data values.  We will look at these
   //in greater detail later too.

   public void Accelerate(int change)
   {
   // code would go here to alter the speed of the bus
   }

   public void StopBus()
   {
   //code would go here to change the speed of the bus to zero
   }

   public void PoliceCarArrives(int change)
   {
   //code here to alter speed of bus to either 100 or 0
   //using some sort of IF statement to decide which
   }

   public void ChangeRoute(int newRoute)
   {
   //alters the route of the bus from one route number to another
   }

   public void PickUpPassengers(int morePassengers)
   {
   //code to increase the number of passengers on board
   }

   public void DropOffPassengers(int lessPassengers)
   {
   //code here to reduce number of passengers
   }

} //end of this class
```

You can think of this class as being like a rubber stamp, with a box for bits to be filled in – use the stamp to create many “blank” buses, then you can fill in the data values for each individual one. Do not worry if you do not fully understand how this works at this stage –we will be devoting much more time to the development of objects and classes over the course of this unit. In the meantime, cut&paste, or type, and compile the above as usual. **Do not expect it to do anything yet (this happens in Book 2)**. All we have created is a blank template – in the next book, we will start building some bus objects with it.

Exercise 1.7 - Mobile Phones.

Go back to the bus class above, re-read the notes and then use this as a starting point for the following exercise.

Consider a mobile phone. What are it's attributes, and which data types would you use to represent them?

Which of the attributes can be changed? What names would you give to the methods that would alter them?

Draw the Class diagram for the mobile phone, using the Bus diagram as a guide. Note that you would normally NOT include the "get", "set" and "constructor" methods in the UML diagram - just the "special" ones at the bottom.

Exercise 1.8

Look at the sample solution for 1.7 on the disk. How does it compare to yours? Now implement the mobile phone class in Java. (Use either your own version or the sample one.) We will build up a small OO system using the mobile phone class later.

Exercise 1.9 - Gemstones.

Consider the following scenario.

A jeweller has a stock of coloured gemstones for setting; as well as the usual mineral stones such as diamonds, sapphires, rubies etc., he also works with coral, pearl and other materials of organic origin.

Gemstones are normally weighed by the carat - they are priced by so-much per carat, but their value may also fluctuate depending on current market prices. The hardness of a gemstone is measured by a value on the MOH scale - typically 10 for diamonds, 6.5 for amethysts and 3 for coral.

What is not widely known is that the characteristics of some gemstones can be changed by heating them - for example, many commercially available Citrines (brown or yellow) are actually heat-treated Amethysts (originally purple or grey).

Design a gemstone class, incorporating the above.

Exercise 1.10

Look at the sample solution for 1.9 on the disk. How does it compare to yours? Now implement the gemstone class in Java. (Use either your own version or the sample one.) In a later workbook, we will look at how to alter the attribute values of both mobile phone and gemstone objects.

Exercise 1.11 - Documentation Standards.

Study the documentation on the diskette for the Bus class. Compare it to both the UML diagram and the source code for the class. Using the blank template, complete the class documentation for both the mobile phone and gemstone classes.

Congratulations! You have now completed Workbook 1!

Resources.

Object Oriented Programming is not new. It has its origins in the late 70's and was born out of a general dissatisfaction with the level of success (or to be more realistic, failure) which was being achieved in large-scale software development projects. Unfortunately the dramatic improvements in price-performance of hardware systems were not being matched with similar gains in software development.

The first programming language to provide direct syntactic support for Object Oriented Programming concepts was *SmallTalk.* However the language that now dominates the present day marketplace is *C++.* At the level of this course, there are very few differences in syntax between Java and C++, so many books that give examples in C++ can also be used for this unit.

The Object Oriented Approach - Satzinger and Orvik
Course Technology 1996
ISBN - 0 7895 0110 4
An invaluable wee book, with a good overview of the growing field of Object Technology - does not concentrate on any one language, which is nice, but examples are not documented using UML.

Beginning OO Analysis and Design with C++ - Jesse Liberty
Wrox Press, 1998
ISBN - 1 861001 33 9
A superb book incorporating both OO Design using UML and implementation in an OO language. Unfortunately this is C++ but don't let that put you off, this is still a great book. Only other drawback is the £32.50 price tag.

Designing Flexible Object Oriented Systems with UML - Charles Richter
Macmillan Technical Publishing, 1999
ISBN - 1 57870 098 1
This is heavy duty OOP theory, covering all you will ever need to know. The $40 cover price is pretty heavy too, but if you can borrow a library copy or obtain a second hand one then it is well worth doing so.

Java in a Nutshell - David Flanagan
2nd Edition (Covers Java 1.1) - O'Reilly 1997
ISBN - 1 56592 487 8
Briefly covers OO concepts, but helps you get Java applications up and running fast. A useful resource book once you have gained a little Java knowledge, and a real investment for the serious Java developer.

Java Examples in a Nutshell - David Flanagan
2nd Edition (Covers Java 1.1) - O'Reilly 1997
ISBN 0-596-00039-1
A tutorial companion to the above, but a wee bit more user-friendly for beginners. Covers Java standalone applications quite well, rather than concentrating solely on applets - another drawback of many Java books available.

`Tester Pots` - you can get sample chapters of both of the above at www.oreilly.com

Java Gently: Programming Principles Explained - Judy Bishop
2nd Edition - Addison Wesley 1998 now OOP - third edition currently being printed.
ISBN - 0201342979
A nice introduction which covers both design principles and programming examples, right up to some fairly advanced features. Highly recommended.

`Tester Pots` - see a synopsis of the book and download source code from http://www.cs.up.ac.za/javagently

Teach Yourself Java in 21 Days – Lemay and Perkins
2nd Edition – Sams Net 1997
ISBN - 1 57521 392 3
Again, a highly readable text, but which tends to emphasise programming examples and applet programming, as opposed to OOP design. Also introduces alternative Java environments, so may be of greater use to "intermediate" Java developers.

Java For Dummies – Aaron E. Walsh
3rd Edition – IDG Press 1998
ISBN - 0 7645 0417 7
The “For Dummies” series is normally highly recommended, but the “Java” one is a wee bit disappointing. It focuses almost entirely on applet programming and introduces some Javascript, which can be confusing for novices (as Java and Javascript tend to be used for completely different purposes!)

Java on the World Wide Web.

There are lots of sites on the WWW dedicated to Java. Here are some handy links:-

http://developer.java.sun.com/developer/onlineTraining/new2java/gettingstartedjava.html - lots of tutorials and helpful information for Java beginners.

http://home.wxs.nl/~gwbrink/JavaClockMaker.htm - customise your own applet to create a Java digital clock for a web page. Not particularly educational, but a nice five minutes harmless entertainment.

http://www.java.co.uk - Refreshingly British-based site with a chatroom.

http://www.javashareware.com - lots of odd Java tools, classes etc. available to download. Also has a *very* nice Post-A-Question section, invaluable for learners!

http://java.sun.com/j2se/1.4.2/docs/api/ - where to find the documentation for all the Java classes. Later on, you will refer to this link A LOT!

Glossary of terms used in these Workbooks

applet - a *Java* mini-program, which is intended to be loaded and executed in a web browser e.g. Netscape or Explorer. An applet cannot be executed outside of any web browser - it is **not** a standalone program! It can be loaded from any remote server connected to the Internet, and it may be subject to certain security restrictions imposed by the browser.

application – a stand-alone *Java* program. Contains one class with a *main()* method.

browser – a program which understands HTTP (hypertext transfer protocol) and which can request HTML pages from any remote server on the WWW. Once an HTML page has been received the browser will interpret the page and display its contents on a window. The most common browsers are Internet Explorer and Netscape Navigator. Browsers must be java-enabled for applets to display.

C++ - one of the original OOP languages, syntactically very similar to Java.

C# (C Sharp) - one of the newest OOP languages, developed by Microsoft and syntactically very similar to both Java and C++.

CASE tools - software application for drawing models of software or systems (Computer Assisted Software(or Systems) Engineering). Some popular programs are Rational Rose and SSADM Select.

Class - a group of similar objects which share the same internal data structure and methods.

Class Library - see *Library.*

Eiffel - the first OOP language to be developed, still occasionally found as a "starter" language for teaching OT and OOP.

Encapsulation - one of the three tenets of OOP - class data is hidden from the user, who can only access it through an external interface.

home page - an HTML page, usually loaded by default by a web browser, which constitutes the "root" (starting page) of any session on the WWW.

FTP (File Transfer Protocol) - a communication protocol which operates on top of the Internet protocol (TCP/IP) and which permits files to be transferred to or from any remote host connected to the Internet

HTML - a textual web page description language called Hyper Text Mark-up Language. An HTML file is interpreted by a browser which then displays it in the window.

HTTP - Hyper Text Transfer Protocol, a communication protocol (which operates above the Internet protocol TCP/IP) which permits programs to retrieve HTML pages from across the WWW. (This most obvious example of a program that uses the HTTP protocol is a web browser.)

Hyperlink – a link to another web site, file or document, embedded in an HTML page. Clicking on the link reference (usually underlined and in a different font colour) makes the browser "jump" to the destination site.

IDE - Integrated Development Environment, a software tool which integrates the standard programming language development tools (e.g. text editor, compiler, debugger etc.) and presents a single uniform interface for creating, editing, compiling and debugging programs. IDE's currently available for Java include **Forte for Java (**Sun Microsystems), **Visual J++** (Microsoft) and **JBuilder** (Borland).

Inheritance - one of the three tenets of OOP classes. Small, specialised classes are built from larger, more general ones.

Intranet - a private inter-network of individual processors, local area networks, or wide area networks. These all use the same communications protocols and have the same available services as the Internet, but there are also security restrictions in place to limit the exchange of information. Intranets are normally used by companies as means of providing corporate services and information for internal consumption only.

Internet - a heterogeneous inter-network of individual processors, local area networks and wide area networks, which can run communication software (based on the TCP/IP protocol) to allow the transfer of data between machines connected to the Internet.

JAR (Java Archive File) - a collection of Java classes bound up in a package similar to a WinZip file. The JAR can be decompressed and the classes used individually, but more commonly the JAR is kept as one unit and used as part of an applet. JARs are often generated by Java IDE packages which may contain their own pre-defined classes, used in place of the standard Java ones.

Javascript – a scripting language which can be embedded in HTML page, to provide extra functions such as displaying the date, etc. Javascript is **not** used to write independent applications and has nothing whatsoever to do with Java, apart from looking a wee bit like it!

JDK – the Java Development Kit, a collection of Java software development tools and predefined software libraries.

JVM - The Java Virtual Machine, a piece of software which sits on the users' platform and interprets the bytecode in a Java class file.

Library - a number of pre-defined classes which come with the JDK installation package. *String* and *Math* are common examples of these.

Object - an entity in space and time, which can be tangible or abstract. It encapsulates one or more items of data and responds to a very tightly defined collection of messages, which execute methods attached to all objects in the class. An object can be described as a particular instance of a class of similar objects.

OA (Object Analysis) - an analysis method which attempts to interpret user requirements in terms of logical classes of objects.

OD (Object Design) - a software design technique which uses classes as the fundamental building blocks for constructing entire software applications.

OOP (Object Oriented Programming) - a program development technique which tries to implement an OO design as a program.

OT (Object Technology) - the overall concept of defining systems or software in terms of objects.

Package - a collection of pre-defined classes, bundled together as a package, which can be incorporated in a Java program. For example *import java.io.** builds the standard input/output package of classes into that program.

Protocol - a set of “rules” defining the execution of a data communication link

POP – Post Office Protocol, a type of SMPT (see below).

Reusability - one of the three tenets of OOPs. Classes should be designed to be re-used as much as possible.

SmallTalk - another early OOP language, now supported by Dolphin Microsystems and commonly used as a "learning" OOP language.

SMPT - Simple Mail Transfer Protocol, a communication protocol which operates on top of the Internet protocol (TCP/IP) and which permits electronic mail messages to be sent to and received from any remote host connected to the Internet.

UML - Unified Modelling Language, a standardised notation for modelling both software and systems. This standard is extensively used in design, systems analysis and software development communities.

URL - Uniform Resource Locator, the web address of a resource anywhere on the Internet.

WWW - World Wide Web (or known just plainly as the WEB), a subset of the Internet comprising those servers which understand HTTP (hypertext transfer protocol) and which can therefore deliver on demand HTML pages to any remote client running browser.

Questionnaire for Work Book 1.

Please complete and return this questionnaire after you have finished this workbook. Your feedback and comments are important as they help us with the ongoing development of this OOPs course.

Please check the appropriate box.	Agree a lot	Agree a little	Neither	Disagree a	Disagree a lot
This workbook was what I expected.					
This workbook was useful.					
This workbook was irrelevant.					
I want to do the next workbook.					
The exercises were too easy.					
The exercises were too hard.					
The workbook was easy to understand.					
The sample programs were easy to understand.					
I would like more programs in my workbook.					
I would like more exercises in my workbook.					
I would like more theory in my workbook.					
There was too much material to complete in one week.					
There was not enough material to last a week.					
The workbook was relevant to the week's lecture.					
I've visited some of the websites in the workbook.					
I found the diskette useful.					
Any other comments? Please say here =>					

Please return this questionnaire to Room 114. Comments, suggestions and corrections may also be e-mailed to c.nyssen@abcol.ac.uk. Thank you.