Programming In Java

BOOK 1

Introduction

CONTENTS

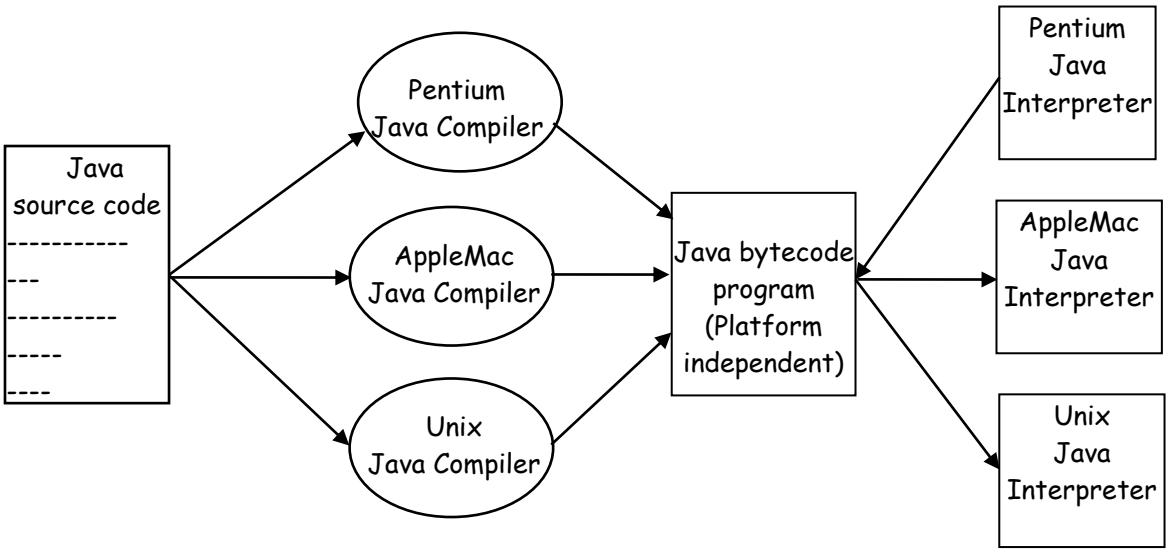
1. SOME PRELIMINARIES

1.1 The Difference Between the Internet and the Web

The Internet is the network infrastructure. It is a collection of computers able to communicate with one another, using a suite of protocols. Computers and networks are registered with the *Internet Information Center* (InterNIC). The Web refers to the web of inter linked documents available on the internet.

1.2 Java Is Platform Independent

Many people connect Java directly with the Internet but many organisations are making the change to Java as a main stream programming language and are discovering that it offers more than its original purpose of implementing applets for Web pages. It can be used as a robust and versatile computing platform for building distributed software applications.



The above diagram shows, for example, a java source program written on a Pentium machine can be run on a Unix machine. It is the Java bytecode program that is transferred to the host machine and then interpreted by the host machine.

1.3 Java Is Object Oriented

You can write Java programs without fully understanding about object technology but you will never write powerful Java programs that exploit all the features of object technology unless you have a sound grasp of objects.

1.4 So What Do I Need

To develop Java programs you will need

- the Java software development kit
- a suitable development environment (an IDE or an editor)

You can download the Java software development kit from Sun Systems. At the time of writing these notes the kit was available at

www.javasoft.com

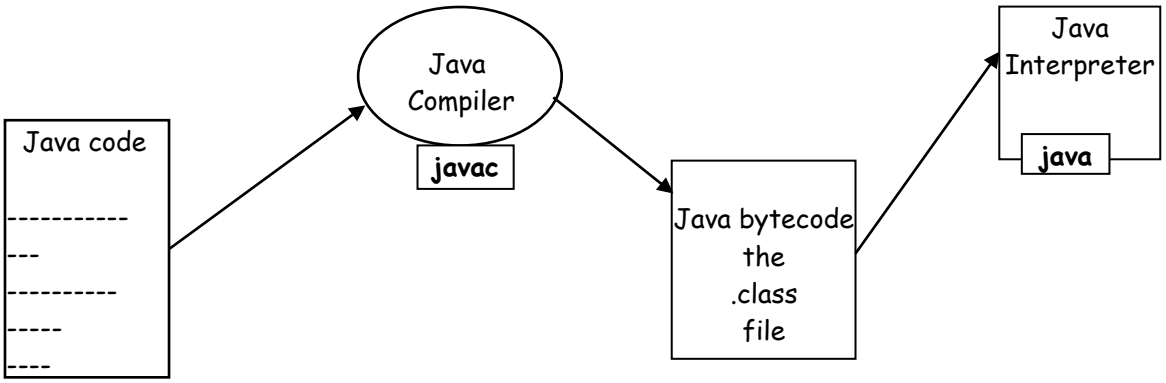
From the main page of the site select Products & APIs. You should then be able to find the following two links.
The software development kit is downloaded with
Java™ 2 SDK Standard Edition v1.3.1 (SDK)
and the supporting documentation with
Java™ 2 Platform, Standard Edition, v 1.3.1 Documentation (Docs)

(both are pretty big so be prepared for a wait…)

To create your Java source code you will need an IDE or a simple editor. There are lots of shareware products available. Search the web and find one that suits you. One that I found very good was a product called JCreator available from
www.JCreator.com

There are of course commercial products like JBulder, J++,

1.5 The Java Development Kit



1.5.1 The Components

This is the collection of tools that you will use to compile, run and debug Java programs. The components of the kit are:
appletviewer
The appletviewer is a utility, which is used to display code, which has been embedded into HTML files. It is used to debug a Java applets before committing them to the more normal environment of a browser.

java
java is the interpreter that runs the object code produced by the Java compiler javac. You will use this program very frequently.

javac
javac is the compiler for the Java programming language. This processes a character file containing Java source code and produces a class file, which can be executed by java. Again, you will use this program very frequently.
javadoc
javadoc is the Java documentation generator. This tool takes Java source code into which has been embedded special comments and will produce HTML files that contain useful documentation about the code that has been processed.

javah
javah is the native method C file generator. This is used to interface code written in C with the Java code that you have written.

javap
javap is a class disassembler. It processes the object code produced by java and displays it in a human readable form.

jdb
jdb is a very primitive debugger which allows you to carry out common debugging functions such as examining variable values. Without access to a better debugger it is probably better to rely on inserting your own diagnostic print statements in the code whenever you have run-time problems.

1.6 The Normal Process of Compiling and Running a Java Program

1.6.1 Setting Your System PATH and CLASSPATH Variables

In the following explanation it is assumed that the Java SDK has been installed in the directory c:\jdk1.3.

In Dos, Windows, Unix and other operating systems the main role of the PATH variable is to enable the operating system to find programs to be executed. The Java SDK compiler (javac.exe) and the Java SDK interpreter (java.exe) are programs so when you execute these program (to compile Java source code and then to run your Java programs) the operating system needs to be able to find javac.exe and java.exe. To enable your operating system to do this you need to ensure that the PATH variable contains the path where the Java compiler (javac.exe) and where the Java interpreter (java.exe) are located.

For example if the locations are,
c:\jdk1.3\bin\javac.exe
c:\jdk1.3\bin\java.exe

your PATH variable must contain the entry,
PATH=...;c;\jdk1.3\bin;...
where the dots represent other path entries.

Your CLASSPATH variable tells the Java intepreter (java.exe) where to find your Java classes. You must set CLASSPATH to identify the locations of your classes. For example if you are developing some Java work and your class files are stored in the folder,
c:\dump\javawork\ja2

then the system CLASSPATH variable must include
CLASSPATH =...;c:\dump\javawork\ja2;...
where the dots represent other path entries.
Some integrated development environments (eg JBuilder) will handle many aspects of the path and classpath roles. If you are using such an IDE the strict requirement of setting the path and classpath variables may not apply. There are other ways within the IDEs that these requirements can be met. However for a full mastery of the Java SDK you should understand the importance of these two variables.

1.6.2 Developing, Compiling and Running

- Create the source code of the class using your chosen editor. Suppose the name of the class is SimpleSeat...
```
public class SimpleSeat {
}
```

- Save the file containing the class as **SimpleSeat.java.**

- **Note that the file name has to be the same as the class name.**

- If there is more than one class in the file then nominate only one of the classes as public by prefacing it with the keyword public and use that class name as the file name.

- Compile the source code using the Java compiler,
```
javac SimpleSeat.java.
```

- The source code of your class should now compile and will either produce syntax errors or will compile cleanly. If syntax errors were produced then correct them and recompile the code.

- Once the code has been clean compiled then run the program by using the Java interpreter.
  java SimpleSeat

1.7 The Java Runtime System (JRE)

When you install the Java SDK the runtime system is also installed. On my Win98 machine the run time system (JRE) was installed as c:\Program Files\JavaSoft. For full information you should consult the readme file in this folder.

The Java(TM) 2 Runtime Environment contains the Java virtual machine, runtime class libraries, and Java application launcher that are necessary to run programs written in the Java programming language. It is not a development environment and does not contain development tools such as compilers or debuggers. For development tools, see the Java 2 SDK, Standard Edition, v1.3.

The Java 2 Runtime Environment includes the Java Plug-in product which enables support for the Java 2 platform on recent releases of Netscape Navigator and Microsoft Internet Explorer. For more information, see the Plug-in web page at http://java.sun.com/products/plugin/ .

1.7.1 Deploying Applications with the Java 2 Runtime Environment

A Java-language application, unlike an applet, cannot rely on a web browser for installation and runtime services. When you deploy an application written in the Java programming language, your software bundle will probably consist of the following parts:

Your own class, resource, and data files.
A runtime environment.
An installation procedure or program.

To run your application, a user needs a Java virtual machine, the Java platform core classes, and various support programs and files. This collection of software is known as a runtime environment.

The Java 2 SDK software can serve as a runtime environment. However, you probably can't assume your users have the Java 2 SDK installed, and your Java 2 SDK license doesn't allow you to redistribute SDK files.

To solve this problem, Sun provides the Java 2 Runtime Environment as a free, redistributable runtime environment. The final step in the deployment process occurs when the software is installed on individual user system. Installation consists of copying software onto the user's system, then configuring the user's system to support that software.

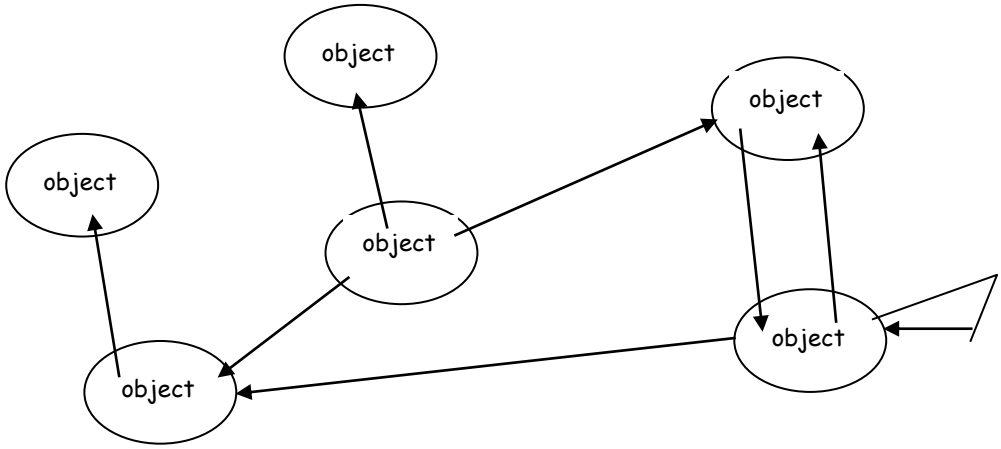
This step includes installing and configuring the runtime environment. If you use the Java 2 Runtime Environment, you must make sure that your installation procedure never overwrites an existing installation, unless the existing runtime environment is an older version.

The Win32 version of the Java 2 Runtime Environment is distributed as a self-installing executable. A simple way to redistribute the Java 2 Runtime Environment is to include this executable in your software bundle. You can then have your installation program run the executable to install the Java 2 Runtime Environment, or simply instruct the user to install the Java 2 Runtime Environment before installing the rest of your bundle. In this installation model, the end-user will have a "public" copy of the Java 2 Runtime Environment just as if it had been downloaded from Sun's website and installed separately.

2. INTRODUCTION TO CLASSES AND OBJECTS

2.1 Object Oriented Programming

Object Oriented programming sees the world as a community of interacting objects. As part of the overall application domain, objects assume responsibilities for managing parts of the system. This collection of objects, working together, provides a complete software solution to the application domain. Objects communicate by sending messages to each other. In one sense objects provide services for other objects. During its lifetime an object may change its state as the demands of the system change. No one object can change the state of another object other than by sending the target object a message that it has previously validated and approved. In order to change it's own state an object may even send itself an approved message. In any particular application some objects may persist for the whole life span of the application. Others may be temporal in that they are created to carry out some required task and when their work is done they are destroyed.



Object oriented programming aims to construct as modular a program as possible, enabling the easy update and reuse of code and understanding of the software system. The philosophy behind OOP is to model the real world as closely as possible. A problem is decomposed into easily understood concepts. All problems are composed of concepts, such as Customer or Loading Bay or Warehouse. If these are programmed by putting their behaviours and properties into modules they can be included in any program where they may be required. The modules described above are declared as classes. They are blueprints from which objects, individual instances of the class, are instantiated. You could be described as an object, an instance, of the class Human.

A program accomplishes its work by passing messages between objects, through the cooperation of connections of objects, rather than by a flow of information around a system.

It is important to keep the inner workings of a class secret from the rest of the co-operating objects. This guarantees data security and is called encapsulation. Communication between objects is achieved through a number of methods which comprise the interface of an object.

The behaviours and properties of a class may be altered without affecting the other parts of a program if the interface is left unchanged.

Another object oriented programming technique is called inheritance. Having decided on a group of concepts to model, it is not uncommon to find that several have behaviours and properties in common. These may be separated to form a more general class from which more particular ones may be derived. Hierarchies of classes may be developed in this way, each succeeding level incorporating the members of those above them.

It is clear that Man and Woman are both derived from Human, but it may be required that Human is in turn derived from Mammal if other classes in the program share the peculiarities of mammals. This will prevent a great deal of duplication of code and also simplify a very complex concept like Human by breaking it down into more manageable parts.

2.2 Object Oriented Analysis and Design

The main purpose of applying oo anaylsis and design methods to an application domain is to
- identify the classes in the system
- identify the role of each class in the system
- identify the relationships between these classes.

Object-oriented analysis and design can be characterised as

- Finding the classes

Identifying the classes (objects) that are pertinent to a particular problem domain.

- Specifying the responsibilities of the class

Determining the role and responsibility of each class within a system

- Identifying the collaborators for each class

Determining those classes without whose assistance it would not be possible for a class to carry out its responsibilities

- Refining responsibilities with use-cases

Learning more about how the objects in a design interact and communicate with one another through the use of sample execution scenarios or role playing. Use-cases capture the dynamics of an object-oriented system - the messages of interest that are passed between the objects, and the associated flow of control.

- Identifying relationships between classes

Refining a design through the identification of important relationships between classes (eg is-a, is-like and is-part-of)

- Refining classes into hierarchies

Discovering logical is-a relationships between classes using the notions of specialization and generalization, Translating into physical class hierarchies to achieve code reuse through inheritance and subclassing. Discovering abstract classes is an important activity in maximizing code reuse.

- Factoring Responsibilites

Migrating behaviour from one class of objects to another within an organised set of related classes so as to achieve implementations of behaviour which may be shared by subclasses

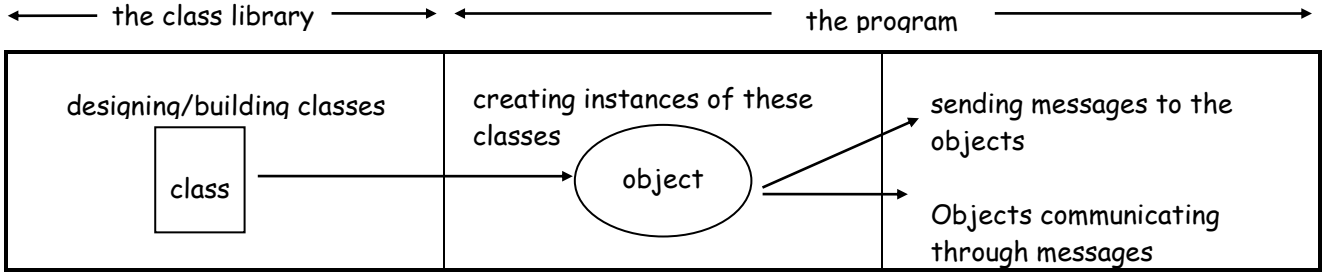
- Looking for reusable design frameworks

Identifying subsystems of collaborating classes, instances of which interact with each other in a manner which can be reused for some particular task or for some particular application domain.

2.3 The Class

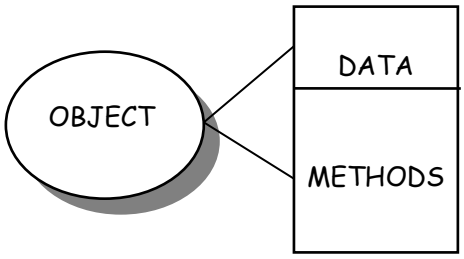
Object Oriented programming is concerned with,

- Designing and building classes,
- identifying relationships between the classes
- creating instances (objects) of these classes,
- writing programs by putting these objects to work and getting the objects to communicate with each other by sending messages to each other

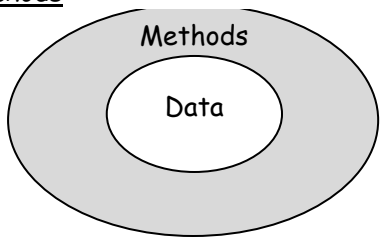


Essentially a class is a general description of all objects instanced from the class. We create and manipulate instances of classes, called objects... hence Object Oriented Programming. Particular instances of a class are called objects.

In object terminology the data is known as the object's attributes which are used to describe the **state** that an object (an instance of the class) might be in. The operations are called *methods* and are used to define the **behaviour** of the object. The attributes (data) describe the state of the object and the methods describe the behaviour of the object. The object's methods are used to query the state of the object and to change the state of the object. State is held by **instance variables**. The collective values of all the instance variables is the state of the object. A change to any one of these variables changes the state of the object.



Encapsulation - access to the data only through the methods



When we say that an object (ie an instance of the class) is fully encapsulated we mean that the only way into the object's data is by using the object's methods. The only way in which the object's state can be changed is by using the object's methods, and even then, the only changes that can take place on the data are those that the designer/implementor of the object's class has allowed for.

The set of methods provide an interface to the object, in that we can use some methods to query the object's state and others to change the object's state. This set of methods is often called the object's **protocol**. This set of methods is defined by the class of the object, not by the object.

Protection of the attributes (the instance variables) by the methods produces the idea of **encapsulation.** Since the only access to the object's attributes is through the object's methods the object's state can be protected against random 'outside' influences. The object's state can only be changed in a controlled way... by the methods. In programming terms this means that a team of programmers working on a project cannot 'randomly' change an object's data using their own methods. There is a consistent, controlled interface to the object's data. Access to data of an object class is only available through the methods of the class. Encapsulation results in reliable, maintainable code that can have one part of it changed without the rest falling over as a result of dependencies.
The user of the object does not need to be aware of the internal data structures. In many ways a user sees an object only through it's methods… it's interface.

In Java the attributes are defined by the **instance variables** and the methods that act on the instance variables are the **instance methods** (usually shortened to just methods).

The class is a general description of the set of all its objects.

An object is a self-contained piece of software that responds to a particular set of messages and is able to hold information.

The information that an object holds (the values of its attributes/instance variables) at any one time is known as its state.

Objects are organised into classes. Objects belonging to the same class (instances of the class) have the same attributes and respond to the same set of messages, responding to each message in an identical manner. Any initialisation of each instance is identical.

A key feature of object technology is abstraction - the reduction of complex systems into component parts represented by objects. This abstraction enables small teams to understand and develop large, complex systems.

Object technology lets developers define new data types that match real world items (objects) such as 'bank account' or 'invoice'. Using object technology, the concept gap in taking real world objects and turning them into software is minimised.
When class B is a subclass of class A, the objects of class B respond to all the messages for objects of class A, and may also respond to some additional messages.

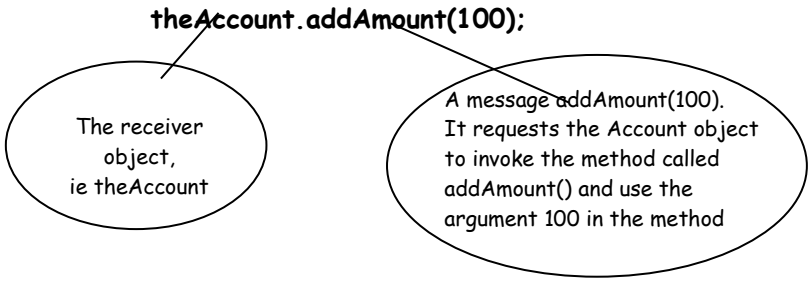
The objects of the subclass have the same properties as those of its superclass, and may have additional ones.

An object oriented software system is structured as a community of interacting agents, called objects. Each object has a role to play. Each object provides a service, or performs an action, that is used by other members of the community (OO Programming, Budd,)

Objects are like agents... we send messages (instructions/requests) to objects and they carry out some service for us. Objects have responsibilities to perform as part of the overall software project.

2.4 Sending Messages to Objects

The Java expression **theAccount.addAmount(100);**
is an example of the message addAmount being sent to the receiver object theAccount. The message carries with it a parameter (100) that will be used by the corresponding method.



When an object receives a message it searches through its protocol list (ie the list of methods defined in its class) in an attempt to find a matching method. If it finds one the method is executed. If one cannot be found, then if the object has any super classes (see next section) a search is carried out in the super classes protocol lists. If no corresponding message can be found the Java system will report an error. To invoke a method you must send a message - to some object.
Notice also that messages can be sent to classes as well as objects. You will see more details of this later in these notes.

- A message may,
  - change the state of an object;
  - make an object do something without altering its state;
  - get back some useful information from an object;
  - cause an object to send a message to another object;
  - be used by an object to send a message to itself.
- The name of a message is called the message selector.
- Some messages have arguments in order to include information with the message.
- The set of messages to which an object responds is called its protocol.
- An object is able to find out about its own state. {use this.queryMethod() }
- The initial state of an object has to be prescribed - usually the class constructor does this.
- A class groups together objects with the same characteristics (the same properties and potential behaviour). When programming, a class is used to define how objects (instances) of the class will be created.

- One class description serves to describe all the objects of that class – the information each can hold and the set of messages to which each can respond.
- The class description is a template for its members (ie created objects), so that each member has the same properties to hold information and responds to the same messages with the same resultant behaviour.
- The 'state' of an object is the sum total of all it's properties values.
- Every object has a set of attributes associated with it. The state of an object at a particular time is determined by the values of these attributes.
- The protocol described by a class is the set of messages understood by the member objects.
- The same message sent to objects of different classes may provoke a different behaviour in each class. (this is known as polymorphism)
- A class may have a subclass.

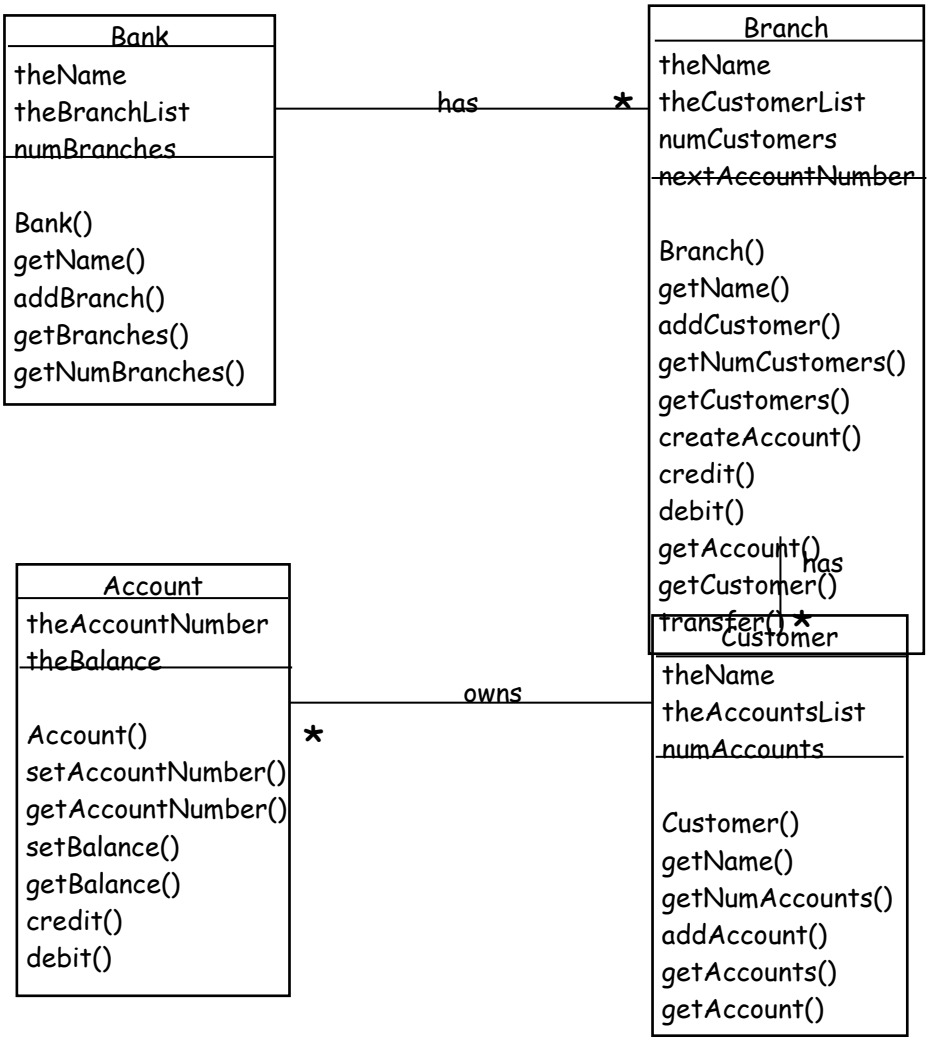
2.5 The Classes and Objects in a Simple Banking System

The first version of our simple system will be constructed using the following classes

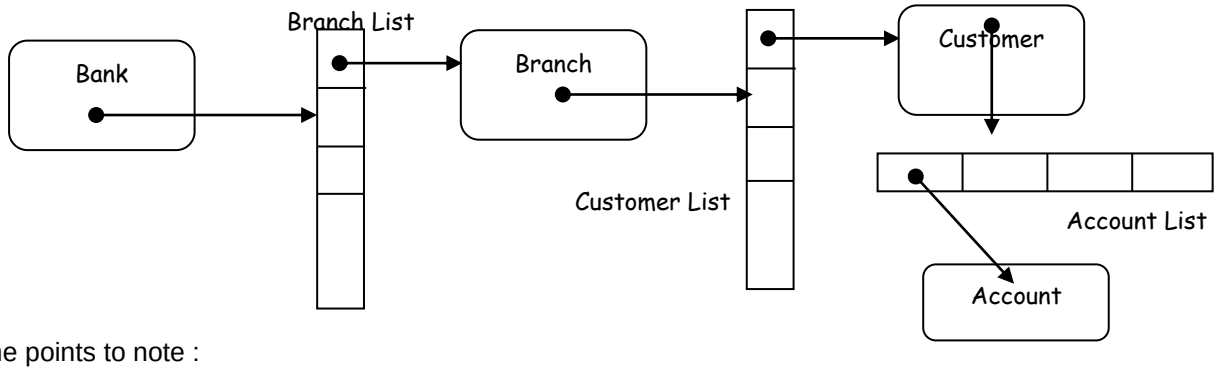
an Account class	a Customer class
a Branch class	a Bank class

Later, in the section on inheritance, we shall take another look at the Account class.

2.5.1 The Class Diagram



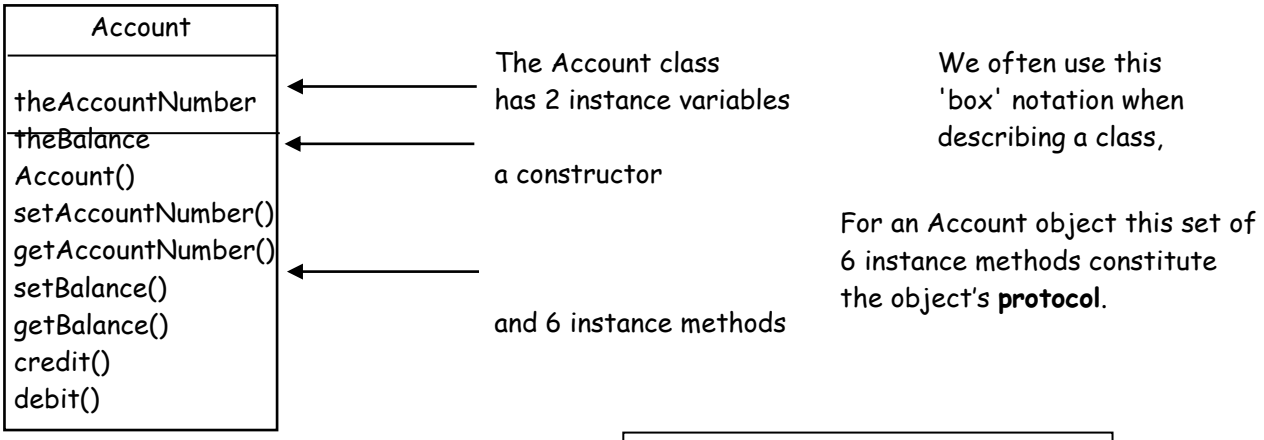
2.5.2 The Object Diagram



Some points to note :

- from within the Bank object a reference is kept to a list of branch object
- each element of the list is a reference to a Branch object
- each Branch object keeps a reference to a list of customer objects
- each customer object keeps a reference to a list of Account objects.

2.6 The Account Class




```
public class Account {
    private String theAccountNumber;
    private int theBalance;

    public Account(String accN,int initBal) {
        this.setAccountNumber(accN);
        this.setBalance(initBal);
    }

    public void setAccountNumber(String aNumber) {
        theAccountNumber = aNumber;
    }

    public String getAccountNumber() {
        return theAccountNumber;
    }
    public void setBalance(int anAmount) {
        theBalance = anAmount;
    }

    public int getBalance() {
        return theBalance;
    }
```

the Java class looks like this... these are the 2 instance variables

The constructor is used when an instance of the class (ie an object) is created with the new operator. You can think of the constructor as initialising the object by setting any instance variable values with initial values.. ie setting the initial state of the object.

void here tells us that the method setAccountNumber() does not return a value.
String here tells us that the method

```
    public void credit(int anAmount) {
        int tempBal = this.getBalance();
        tempBal += anAmount;
        this.setBalance(tempBal);
    }

    public boolean debit(int anAmount) {
        int tempBal = this.getBalance();
        if (tempBal - anAmount >= 0) {
            tempBal -= anAmount;
            this.setBalance(tempBal);
            return true;
        }
        else return false;
    }
}
```

tempBal += anAmount;
is the same as
tempBal = tempBal + anAmount;

In using the keyword this, the object is effectively sending a message to itself.

The use of the keyword this arises when encapsulation is enforced with the methods.

2.7 Encapsulation (Setter and Getter Methods)

The code of the Account constructor is an example of encapsulation being used.
Consider the instance variable theBalance. In addition to its declaration, there are a number of references to the instance variable within the various methods of the Account class. The two methods setBalance() and getBalance() are known as accessor methods (often called setter and getter methods). Notice how every reference to the balance instance variable is made by using the methods setBalance() and getBalance()... except of course within these two methods.

There is a strong argument within the object world that such accessor methods should always be used to set and return instance variables values. The argument for using accessor methods is based around maintenance considerations. If the internal representation of an instance variable is changed then methods which reference the instance variables by means of the accessor methods would not need updating.

In software exhibiting strong encapsulation, only accessor methods are allowed to make direct reference to instance variables. Unfortunately Java does not enforce encapsulation.

So the following is a (poor...) alternative to using strong encapsulation.

```
    public Account(String accN,int initBal) {
        theAccountNumber = accN;
        theBalance = initBal;
    }

    public boolean debit(int anAmount) {
        if (theBalancel - anAmount >= 0) {
            theBalancel -= anAmount;
            return true;
        }
        else return false;
    }
```

2.7.1 Using The Keyword (this) To Refer to The Instance Variables

Java also allows you to use this to refer to the instance variables of the class usually in the following context where an argument name is the same as an instance variable name. Probably best practice to avoid this.

```
public setLength(int maxLength) {
    this.maxLength = maxLength;
}
```

2.7.2 Testing the Account Class

When we look at static (class) methods later, the significance of the keyword static in the header of the main method will be explained.

We now construct another class that we use to can test the methods of the Account class.

```
public class TestAccount {
    public static void main(String args[]) {
        Account myAccount = new Account("1234",100);
        String theAccNumber = myAccount.getAccountNumber();
        int theBalance;
        theBalance = myAccount.getBalance();
        System.out.println(theAccNumber + " : " + theBalance);
        myAccount.credit(50);
        theBalance = myAccount.getBalance();
        System.out.println(theAccNumber + " : " + theBalance);

        boolean canDebit;
        canDebit = myAccount.debit(100);
        if (canDebit) {
            theBalance = myAccount.getBalance();
            System.out.println(theAccNumber + " : " + theBalance);
        }
        else System.out.println("Amount too large...");

        canDebit = myAccount.debit(51);
        if (canDebit) {
            theBalance = myAccount.getBalance();
            System.out.println(theAccNumber + " : " + theBalance);
        }
        else System.out.println("Amount too large...");
    }
}
```

here we send the message getBalance() to the object referenced by myAccount. The object myAccount returns the current balance and the value is assigned to the temporary variable

here we just ask the myAccount object to credit it's current balance with 50. No response is received from the object.

here we ask the myAccount object to attempt to debit its current balance. If the object can carry out the debit it indicates this by returning a true value. If it can't carry out the debit the object returns a false value.

2.8 Creating Objects - The Constructor

The constructor is used to when an instance of the class (ie an object) is created with the new operator. You can think of the constructor as initialising the object by setting any instance variable values with initial values.. ie setting the initial state of the object. The primary role of the class constructor is to reserve memory for a new object as it is created. An interesting point is that the constructor only reserves memory space for the instance variables, since it does not have to reserve space for the methods. The methods exist only with the class. This means that when we create more than one object from a class the methods exit only once. Methods are tied to the class and not to the object. All objects of the same class have the same set of methods, the same protocol.

There are 2 kinds of constructor.

1. The default constructor.
2. A user defined constructor.

1. The Default Constructor

This constructor does nothing more than reserve memory space for the object. It performs no processing on the object's data. See 6.6 for further comments on this.

2. A User Defined Constructor

When objects are created we usually want to assign initial values to the object, ie to set up the initial state of the object. Since objects of the same class are often required to be created in the same state, this initial state is normally achieved by the parameters we pass to the constructor method.

2.8.1 Variable Initialisation

Only instance variables are automatically initialised to their default values. Local variables, those declared and used in methods, must be set before use.

2.9 Creating Objects

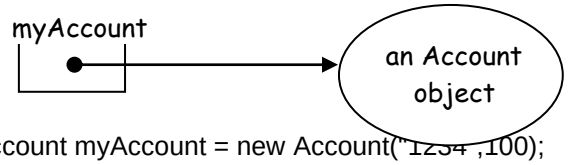
2.9.1 The new Operator and Reference Semantics

New is an operator, (actually a unary operator). The operand of new is the class constructor. The combination of new and the class constructor creates and allocates space for an object (an instance) of the

class. The constructor code is executed and new returns a pointer to the allocated space. The execution of the code

```
Account myAccount = new Account("1234",100);
```

produces the following,



an instance of the Account class is created; initialised so that the instance variable accNumber is set to the string value "1234", and the instance variable balance is set to 100

Account myAccount = new Account("1234",100);

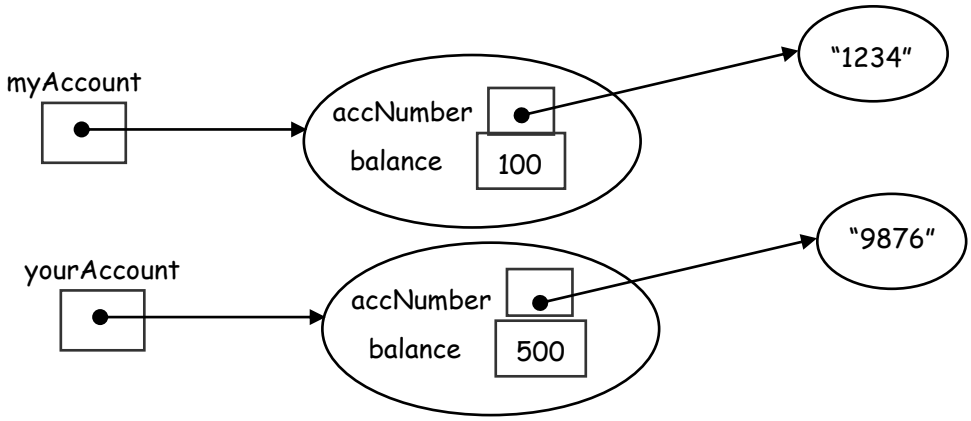
the identifier myAccount is a reference to the Account object. The identifier myAccount contains the address of the Account object. Since we never deal with objects directly, but reference them by variables we say that Java uses reference semantics.

2.10 Practical Work

You should now attempt exercises 1 and 2 from the Java Exercises.
Exercise 1 requires you to add extra code to the Account test class and exercise 2 requires you to develop the Customer class and then test the methods of the Customer class.

2.11 Reference Semantics and Object Comparison

Suppose we need to ask the question if two Account objects are the 'same'. Suppose we have
Account myAccount = new Account("1234",100);
Account yourAccount = new Account("9876",500);

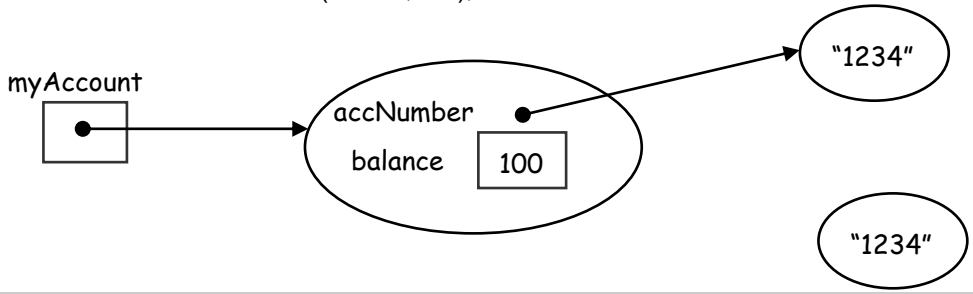


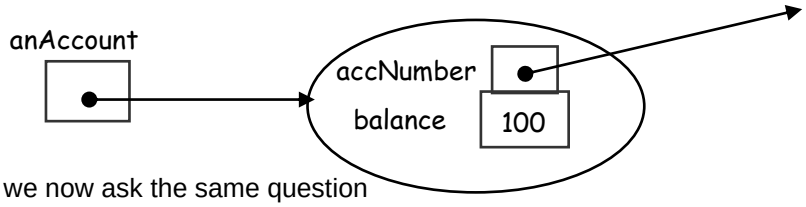
One way of asking the question might be
if (myAccount == yourAccount) …

here it looks as though the answer will be no, and in fact the answer is no… but why ?

Next consider the new situation…

Account myAccount = new Account("1234",100);
Account anAccount = new Account("1234",100);





Suppose we now ask the same question
if (myAccount == anAccount)…. what now will be the answer ?

Again it will be no. The Account object referenced by the variable myAccount is not the same Account object that is referenced by the anAccount variable. All that the == operator compares is the value of the variable myAccount with the value of the variable anAccount. The variable myAccount contains the address of the 'first' Account object and the variable anAccount contains the address of the 'second' Account object. Of course they are not the same.

But what if we want to interpret the 'same' as having the same state. That is we are really asking the question 'are the state of two objects the same'. In many situations this is what we would really mean if we are asking if two objects are the same. This would be particularly relevant in searching operations. From above it is clear that we cannot use the == operator to answer a question if two objects have the same state.

What we need is another way of carrying out the testing. It turns out there is one, albeit a rather confusing one. All classes in the Java API are provided with a 'default' equals() method. They all inherit one from the Object superclass. This means we can write the following,

if (myAccount.equals(anAccount)) …

But sadly this again will say no in the above case. The version of equals() inherited from the Object class does not compare the states of two objects, in this case two Account objects, but simply compares if they are the same object. It works like the == operator in that it simply tests if the variables referencing the two objects are equal; that is, do the two variables point to the same object.

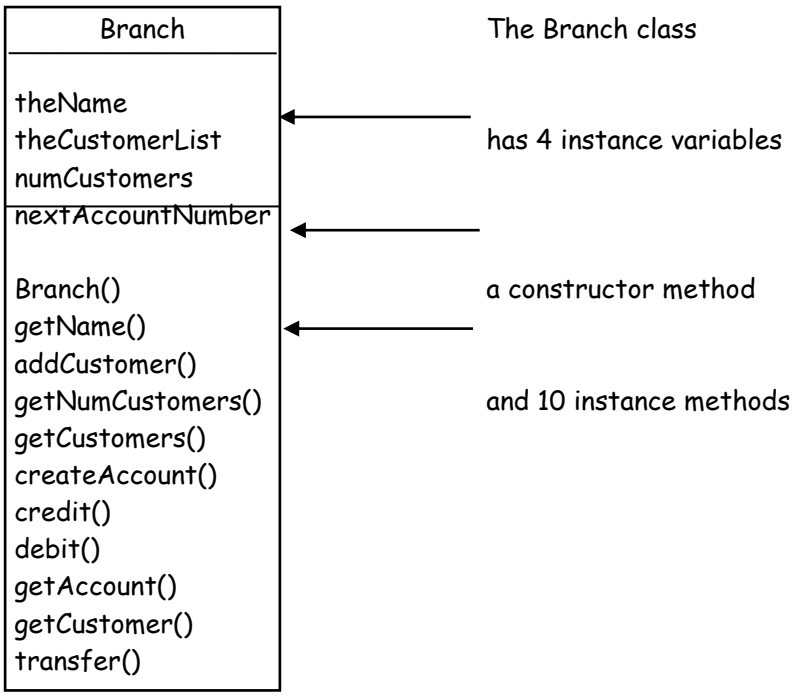
For the equals() method to work properly for any pair of Account objects we have to override the method from the Object class. All of this means that we have to add a new method to our Account class as follows,

```
public boolean equals(Object obj) {
    Account param = (Account)obj;
    return    (   this.getAccNumber().equals(param.getAccNumber()) &&
                  this.getBalance() == param.getBalance()));
}
```

Generally in software applications when we ask if two 'things' are equal we usually mean do they have the same value? Now in object technology we transfer the word value to the word state. If we now ask are two objects equal we generally mean do they have the same state. Two objects will be equal if their corresponding instance variables are equal. In 'everyday' terms we would expect the question… does Account x equal Account… to be yes, if both Accounts have the same state.

As this is such an important issue there are several references and explanations later in the books.

2.12 The Branch Class




```
public class Branch {
    private String theName;
    private Customer[] theCustomerList;
    private int numCustomers;
    private String nextAccountNumber;

    public Branch(String aName, String aStartAccNumber) {
        theName = aName;
        theCustomerList = new Customer[10]; // no more than 10 customers...
        numCustomers = 0;
        nextAccountNumber = aStartAccNumber;
    }

    public String getName() {
        return theName;
    }

    public void addCustomer(Customer aCustomer) {
        theCustomerList[numCustomers++] = aCustomer;
        this.createAccount(aCustomer);
    }

    public int getNumCustomers() {
        return numCustomers;
    }

    public Customer[] getCustomers() {
        return theCustomerList;
    }
    public void createAccount(Customer aCustomer) {
        Account aNewAccount = new Account(nextAccountNumber,0);
        aCustomer.addAccount(aNewAccount);
        int tempInt = new Integer(nextAccountNumber).intValue();
        ++tempInt;
        nextAccountNumber = new Integer(tempInt).toString();
```

```
    }

    public void credit(String anAccountNumber, int anAmount) {
        Account targetAccount = this.getAccount(anAccountNumber);
        targetAccount.credit(anAmount);
    }

    public boolean debit(String anAccountNumber, int anAmount) {
        Account targetAccount = this.getAccount(anAccountNumber);
        boolean debitOk = targetAccount.debit(anAmount);
        return debitOk;
    }

    public Account getAccount(String accNumber) {
        Account aTempAccount = null;
        Customer nextCustomer;
        for (int i = 0; i < numCustomers; ++i) {
            nextCustomer = theCustomerList[i];
            aTempAccount = nextCustomer.getAccount(accNumber);
            if (aTempAccount != null) break;
        }
        return aTempAccount;
    }

    public Customer getCustomer(String aName) {
        Customer theCustomer = null;
        for (int i = 0; i < numCustomers; ++i) {
            if (aName.equals(theCustomerList[i].getName())) {
                theCustomer = theCustomerList[i];
                 break;
            }
        }
        return theCustomer;
    }


    public boolean Transfer(Account fromAccount, Account toAccount, int anAmount) {
        // Attempt a transfer of anAmount from the first account, fromAccount, to the
        // second account, toAccount
        // return true if the transfer is valid, false otherwise
    }
}
```

2.12.1 Testing the Branch Class

Testing the Branch class is a little more complicated than testing the Account and Customer classes. We therefore adopt a slightly different approach. We still use a test class TestBranch but this time we create a TestBranch object and then use this object to do the testing for us. Use this class to test your Branch class.

```
public class TestBranch {
    private Branch blackBurn;

    public TestBranch() {
        // First create some customers...
        Customer aCustomer = new Customer("jeff");
        Customer anotherCustomer = new Customer("pat");

        // Next create the Branch and add some customers...
        blackBurn = new Branch("Blackburn","1000");
        blackBurn.addCustomer(aCustomer);
```

here we create two Customer objects and refer to them as aCustomer and anotherCustomer; the name associated with the aCustomer object is jeff

create a new Branch object and add the two customers to the customer list of the branch

```
        blackBurn.addCustomer(anotherCustomer);

        // Create accounts for the customers
        blackBurn.createAccount(aCustomer);
        blackBurn.createAccount(anotherCustomer);
        blackBurn.createAccount(aCustomer);

        this.displayBranchDetails();

        blackBurn.credit("1000",50);
        blackBurn.credit("1003",100);

        this.displayBranchDetails();

        boolean debitOk = blackBurn.debit("1000",20);
        if (!debitOk) System.out.println("Cannot debit account 1000...");
        debitOk = blackBurn.debit("1001",30);
        if (!debitOk) System.out.println("Cannot debit account 1001...");
        this.displayBranchDetails();
    }

    public void displayBranchDetails() {
        // Display the details at the Branch
        String theBranchName;
        Customer nextCustomer;
        Account[] accountsHeld;
        Account nextAccount;

        theBranchName = blackBurn.getName();
        System.out.println(theBranchName);
        Customer[] customerList = blackBurn.getCustomers();

        // Process each customer...
        int currentNumCustomers = blackBurn.getNumCustomers();
        for (int i = 0; i < currentNumCustomers; ++i) {
            nextCustomer = customerList[i];
            System.out.println(nextCustomer.getName());
            accountsHeld = nextCustomer.getAccounts();
            // For each customer process the held accounts...
            int currentNumAccounts = nextCustomer.getNumAccounts();
            for (int j = 0; j < currentNumAccounts; ++j) {
                nextAccount = accountsHeld[j];
                System.out.println(nextAccount.getAccountNumber() + " : " + nextAccount.getBalance());
            }
        }
    }

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

the use of the keyword **this** is to send a message to the current TestBranch object. In this case it is to execute the displayBranchDetails() method... see below for the method.

request the blackBurn Branch object to credit account number 1003 with 100

2.13 Practical Work

- You should now attempt exercise 3 from the Java Exercises.
  You are required to implement the transfer() method and then add extra code to the TestBranch class.

- Do exercise 4 which requires you to implement and test the Bank class

2.14 Sample Code for The Banking Classes

2.14.1 The Account Class

The Account class is provided as part of the notes

2.14.2 The Customer Class

```
public class Customer {
    private String theName;
    private Account[] theAccountsList;
    private int numAccounts;

    public Customer(String aName) {
        theName = aName;
        theAccountsList = new Account[6]; // no more than 6 accounts held...
        numAccounts = 0;
    }

    public String getName() {
        return theName;
    }

    public int getNumAccounts() {
        return numAccounts;
    }

    public void addAccount(Account anAccount) {
        theAccountsList[numAccounts++] = anAccount;
    }

    public Account[] getAccounts() {
        return theAccountsList;
    }

    public Account getAccount(String anAccountNumber) {
        Account theAccount = null;
        for (int i = 0; i < numAccounts; ++i) {
            if (theAccountsList[i].getAccountNumber().equals(anAccountNumber)) {
                theAccount = theAccountsList[i];
                break;
            }
        }
        return theAccount;
    }
}
```

2.14.3 The Branch Class

The Branch class is provided as part of the notes. Here is the additiinal transfer() method,

```
    public boolean transfer(Account fromAccount,Account toAccount,int anAmount) {
        if (fromAccount.debit(anAmount)) {
            toAccount.credit(anAmount);
            return true;
        }
        else return false;
    }
```

The following is an example of some additional code added to the TestBranch class to test the transfer() method...

```
        // Test the transfer() method, first a valid transfer...
        Account sourceAccount = blackBurn.getAccount("1000");
```

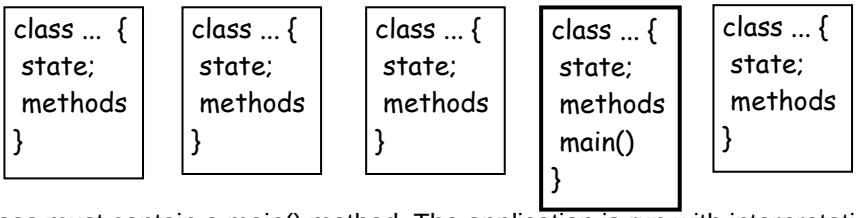
```
        Account destinationAccount = blackBurn.getAccount("1001");
        boolean transferOk = blackBurn.transfer(sourceAccount,destinationAccount,10);
        if (!transferOk) System.out.println("Should not see this message...");
        // See the results of the transfer..
        this.displayBranchDetails();
        // Now an invalid transfer...
        transferOk = blackBurn.transfer(sourceAccount,destinationAccount,100);
        if (!transferOk) System.out.println("Should see this message... transfer invalid");
```

2.14.4 The Bank Class

```
public class Bank {
    private String theName;
    private Branch[] theBranchList;
    private int numBranches;

    public Bank(String aName) {
        theName = aName;
        theBranchList = new Branch[5]; // no more than 5 branches...
        numBranches = 0;
    }

    public String getName() {
        return theName;
    }

    public void addBranch(Branch aBranch) {
        theBranchList[numBranches++] = aBranch;
    }

    public Branch[] getBranches() {
        return theBranchList;
    }
    public int getNumBranches() {
        return numBranches;
    }
}
```

2.15 The Structure of a Java Program

A typical Java program, or application as it is often called, will have the following structure,



One of the classes must contain a main() method. The application is run with interpretation commencing at the main() method. The Java run time system will search out the main() method and begin execution at this point and use the facilities of the other classes when it needs to. Another feature of this approach is that each class can be defined in a separate file. Notice that this means a minimum Java program (application) would consist of just one class definition file containing a main() method.

You can also define several classes within the same file with one of the classes containing your main() method.

```
class1 {

}

class2 {
  main()
}

class3 {
```

If you do this you must ensure that you save the file with a name the same as the class containing your main() method otherwise the interpreter will not be able to find to find the main() method to commence interpretation and execution of your application.
In the example opposite the file would have to be saved as Class2.java.

In this case the Java compiler (javac) will actually produce three separate class files
class1.class
class2.class
class3.class

2.16 Overloading The Object Constructor

In Java the method that gets overloaded the most is probably the constructor method. Overloading constructors allows you to build new objects in different initial states. Overloading effectively means having different versions of the same method in the same class... each version doing a different job. You can also overload other methods but this is not something you will need to do very often; not nearly as often as you will want to overload constructors. The following modification to the Branch class gives it three constructors,

The 3rd one is a funny one… it's that word **this** again. When used in this context it refers to the current class and calls the constructor with one string argument. If an instance of the Branch is created without any arguments being passed to the constructor such as

Branch blackBurn = new Branch();
then rather than some error occurring the use of **this** in this context acts as a kind of default creation... whether you want this to happen is of course another matter...

```
public class Branch {
    private static final String START_NUMBER = "1000";

    private String theName;
    private Customer[] theCustomerList;
    private int numCustomers;
    private String nextAccountNumber;

    public Branch(String aName, String startAccount) {
        theName = aName;
        theCustomerList = new Customer[10]; // no more than 10 customers...
        numCustomers = 0;
        nextAccountNumber = startAccount;
    }

    public Branch(String aName) {
        theName = aName;
        theCustomerList = new Customer[10]; // no more than 10 customers...
        numCustomers = 0;
        nextAccountNumber = START_NUMBER;
    }

    public Branch() {
        this("No Name Given");
    }
```

Take a look at the String class in the java.lang package... this has 7 constructors.

rest of the class code...

2.17 Class Constants

Also demonstrated in the above code, this is how we define class constants in Java

```
private static final String START_NUMBER = 1000;
```

3. SOME JAVA BASICS

3.1 Scalars and Objects

Java treats all the usual simple data types as scalars, not objects. The scalar types are,

byte, short, int , long float, double char, boolean

All are manipulated as in many other languages. All other data types, as you define them and use them are treated as objects. In this sense Java is not quite a pure object oriented language as say Smalltalk. Sometimes you will need to treat integers, reals etc as objects and Java makes provision for this with the Wrapper feature. Wrappers allow, for example, a simple integer data type to be converted into an Integer object so that it can them be treated as an object for processing. See the later section on Object Wrappers for more details.

3.2 Literals

Literal is a programming language term, which essentially means that what you type is what you get. For example, if you type 100 in a Java program, you automatically get an integer with the value 100. If you type 'a', you get a character with the value a. Literals may seem intuitive most of the time, but there are some special cases of literals in Java for different kinds of numbers, characters, strings, and boolean values.

3.2.1 Number Literals

There are several integer literals. For example, 100 is an integer literal of type int (although you can assign it to a variable of type byte or short because it's small enough to fit into those types). An integer literal larger than an int is automatically of type long. You also can force a smaller number to a long by appending an L or 1 to that number (for example, 4L is a long integer of value 4). Negative integers are preceded by a minus sign-for example, -45.

Integers can also be expressed as octal or hexadecimal: a leading 0 indicates that a number is octal-for example, 0777 or 0004. A leading 0x (or 0X) means that it is in hex (0xFF, 0XAF45).

Floating-point literals usually have two parts: the integer part and the decimal part-for example, 5.677777. Floating-point literals result in a floating-point number of type double, regardless of the precision of that number. You can force the number to the type float by appending the letter f (or F) to that number-for example, 2.56F. You can use exponents in floating-point literals using the letter e or E followed by the exponent (which can be a negative number): 10e45 or .36E-2.

3.2.2 Boolean Literals

Boolean literals consist of the keywords true and false. These keywords can be used anywhere you need a test or as the only possible values for boolean variables.

3.2.3 Character Literals

Character literals are expressed bya single character surrounded by single quotes: ' a' , , # ' , , 3' , and so on. Characters are stored as 16-bit Unicode characters.

There are also String literals. These are dealt with later in the notes.

3.2.4 A Possible Problem When Using Floating-point Literals

From the above... "Floating-point literals result in a floating-point number of type double, regardless of the precision of that number. You can force the number to the type float by appending the letter f (or F) to that number-for example, 2.56F."

This means that if you attempt to do something like,
    float y = 15.456;
You will get a compilation error something like the following,

Incompatible type for declaration. Explicit cast needed to convert double to float.
        float y = 15.456;

This occurs because by default the literal 15.456 is of type double. There are three ways out of the problem...

- declare y as a double double y = 15.456;
- append f after the literal float y = 15.456f;
- use a cast float y = (float)15.456;

3.3 Some of the more common Java operators

The Arithmetic Operators,

+ - * /	as usual, (integer division truncates fractional part)
%	modulo (for integers),

The Increment and Decrement Operators,

++x	is equivalent to the more familiar x = x + 1;
--y	is equivalent to the more familiar y = y - 1;
y = ++x	x is increased by 1 and then its value is assigned to y
y = x++	x is assigned to y and then the value of x is increased by 1

Note that the assignment, x = 6/4 results in setting x to 1.

3.3.1 The Relational Operators,

> >= < <=	as usual,
==	equal to, (test for equality),
!=	not equal to.

3.3.2 The Logical Operators,

&&	AND
||	OR
!	NEGATION.

3.3.3 The Assignment Operator

Assignment is = (a single equals sign), so the statement, area = length * breadth, assigns the result of length * breadth to the variable area.

3.3.4 The if () else Construct

This control construct has the format,

```
if (exp) statement;
if (x == 5) y = y*p;

    if (x > 0) {
        y = 3*p;
        System.out.println(y + p);
    }
    else {
        y = 3*q;
        System.out.println(y + q);
    };
```

```
if (exp) statement 1; else statement 2;
if ( x != 0) p = y/x; else p = 0;
```

A common source of logic error is...

```
int x;
x = 5;
if (x = 3) System.out.println("x equals 3");
else System.out.println("x equals 5");
```

the above code will ALWAYS produce
x equals 3.....why ?

3.3.5 The ? : Construct

This selection construct has the form

```
variable = (boolean expression) ? p : q;
```

If the boolean expression is true the variable is set to p, otherwise the variable is set to q.

It is of course equivalent to

```
if (boolean expression) x = p; else x = q;
```

3.3.6 The switch Construct

The multi-selection construct of Java is the switch construct. The form is as follows...

```
switch (exp1) {
        case <constant exp2>: statement;
        case <constant exp3>: statement;
                etc...
        default               : statement;
};
```

exp1 is evaluated and control is transferred to whichever case matches. If no match takes place the default case is executed.

```
switch (3*x-y) {
    case 1 :  System.out.println("\nDenary Multiplication");
              range = denary;
              < statements >
              break;
        case 2 :  System.out.println("\nOctal Multiplication");
              range = octal;
              < statements >
              break;
        default :  System.out.println("\nError message...");
} // End of switch
```

3.3.7 The while and do while Constructs

These control constructs have the format,

```
while (exp is true)  {statement;}

count = 0;
while (i <= someval) {
        ++count;
        statements;
        ++i;
}
```

```
do {statements;} while (exp is true);

    i = count = 0;
do {
    ++count;
    statements;
    ++i;
} while (i <= someval);
```

3.3.8 The for Construct

This control construct has the format,

for (exp1; exp2; exp3) statement;	

for (i = 0; i <= 20; ++i) {
  System.out.println("i squared = " + x*x);
} // End of for loop
	for (x = 0, j = 1; x != 200; ++x, ++j) statement;

```
watch this...   for (i = 1; i < end; ++i);          // the semi-colon after the bracket ) will cause
                        sum = sum +i;          // big trouble if  it is intended to repeatedly add i to sum
```

Note : The for construct is really a disguised while construct. The loop continues while the 'centre' condition remains true... so for example the following are equivalent,

```
for (i = 0; i <= 20; ++i) {
    statements
}
```

```
i = 0;
while (i <=20)  {
    statements;
    ++i;
}
```

You can also increment/decrement in steps other than 1, eg

```
for (int i = 0; i < 20; i += 5) System.out.println(i);
```

3.3.9 The break Statement

```
for (i = 0; i < MAX; ++i) {
    if (i == VAL) break;
    statement_1; statement_2;
}
statement 3;
```

When variable i is equal to VAL the break statement is executed causing control to pass to statement 3. If i does not equal VAL, statements 1 and 2 are executed.

3.3.10 The continue Statement

```
for (i = 0; i < MAX; ++i) {
    if (i == VAL) {
        ++count;
        continue;
    };
    statement1
    statement 2;
}
```

If i equals VAL the variable count is incremented and the continue statement causes the for loop to execute again, missing out the statements 1 and 2.

You will probably gather that both ***break*** and ***continue*** are really disguised ***goto*** statements. They are however *restricted* gotos in that they jump only forward to known points and hence do not contradict the concepts of structured programming.

3.4 Arrays

Arrays in Java are not pure objects. That is they do not understand messages and are treated very much in the same way as in non object oriented languages.

3.4.1 Declaring Arrays

int[] num; declares num to be array of type int.

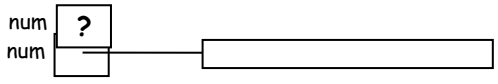
Dice d[]; declares d to be an array of type Dice (we assume that dice is a user defined class type)

The declaration can also be written as int num[] either style is used.

3.4.2 Array Allocation

Declaring an array does not allocate space for the array. To allocate storage for an array you need to use the new operator to create an array of a specified size,

```
int num[];
num = new int[20]  ;
```



In Java it is usual to combine the declaration and allocation as follows,

int theTable[] = new int[100]; declares and creates an integer array to hold 100 elements (max)

String theNames[] = new String[20]; declares and creates a String array of 20 elements (max)

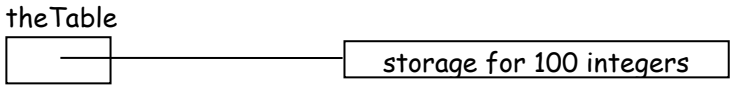
int aBlock[][] = new int[10][20]; declares and creates a 2 dimensional integer array

[10 rows, 20 cols]

Actually Java only supports single dimensional arrays. Hence the last example actually declares *aBlock* to be an array of 10 arrays each of 20 elements.

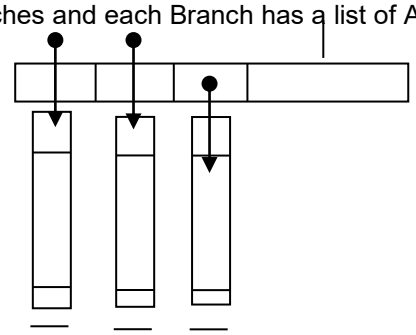
The second way to allocate storage to an array is with a static initialiser,
int buffer[] = {1,2,3,4,5}; declares and allocates an integer array of 5 elements
Reference to the elements is in the usual way...
theTable[3] = buffer[0]; theBlock[0][4] = theTable[7];

Following on from the description of reference semantics the following code
int theTable[] = new int[100]; actually produces the situation as follows



You can find the length of an array with theArray.length
Note length is not a message… it is in fact an attribute of the array.

Arrays of Arrays
The statement... "a bank has a list of Branches and each Branch has a list of Accounts" can be modeled as...
Branch theBank[];
Account aBranch[];



3.5 Strings

A String is an object, but there are facilities in Java for manipulating String objects that do not appear to be object oriented but which have been included probably to conform to facilities in other, non object oriented languages (C++ probably…).

When the javac compiler meets a declaration such as
String aCountry = "Scotland";
"Scotland" is a string literal and aCountry is a reference to it.

the compiler does the work of creating the String object and then assigns the address of the newly created String object to the variable aCountry.

As a matter of good OO practice you are advised to always think and treat Strings as objects. In which case, to assign a new string value to aCountry would take something like

aCountry = new String("England");

Strings are **immutable**. That means, that once created, a String can never be altered in any way. All the methods that appear to alter a String actually return a new String - leaving the original untouched.

The compiler makes sure that each String constant actually results in a String object. This means that when we do something like,
aCountry = "Scotland";
and then later
aCountry = "England";

we have not 'changed the value' of the String object but we have created a new String object ("England") and made aCountry point to it. If you like we have changed the value of the reference, not the 'original' String object.

3.5.1 Testing if Two Strings are Equal

The problems with doing this are part of a wider picture when testing if two objects are the 'same'. This wider issue is discussed later in these notes.

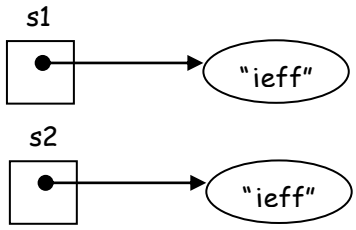
3.5.2 A note on the == operator and the equals() method for String Objects

Newcomers to the Java language often confuse the use of the operator = = and the method equals() when comparing two Strings. The former, since it is an operator, is used to compare scalars . When it is used to compare scalars it carries out a comparison on their values. If it used in the context of comparing String objects it is not really comparing the objects at all. It simply compares the addresses of the two objects. The method equals() must be used to compare the equality of String objects.

Not understanding the difference between the operator == and the method equals() can lead to some very subtle programming errors, so be awake. It's all to do with reference semantics.

Try this example out for yourself and try to explain the outputs.

```
public class A {
    public static void main(String args[]) {
        String  s1, s2;
        s1 = new String("jeff"); s2 = new String("jeff");
        if (s1 == s2) System.out.println("oh dear... what's gone wrong");
        if (s1.equals(s2)) System.out.println("Good... thats better");
    }
}
```

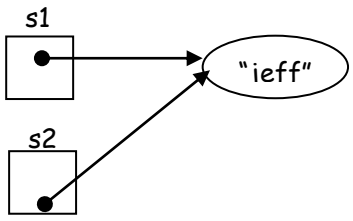


The output is, Good... thats better

s1 is not equal to s2. They are not String objects but each is a variable whose value will be the address of a String object. They have different values since they hold different address values. However the String object pointed to by s1 and the String object pointed to by s2 have the same state and in this sense are the 'same'. In our software model of the real world we would probably want the answer to 'is s1 equal to s2' to be yes, they are the same.

If we now use the alternative approach of creating a String object, that of creating a String literal,

```
public class A {
    public static void main(String args[]) {
        String  s1, s2;
        //s1 = new String("jeff"); s2 = new String("jeff");
        s1 = "jeff";
        s2 = "jeff";
        if (s1 == s2) System.out.println("oh dear... what's gone wrong");
        if (s1.equals(s2)) System.out.println("Good... thats better");
    }
}
```



The output this time is, oh dear... what's gone wrong
Good... thats better

The reason the string-equality bug is such a common pitfall for novice Java programmers is that simple tests to show its existence appear to confirm that the code is all right. The reason for this is that the Java compiler is rather clever with String literals and what it does with them confuses the issue of equal references versus equal values (as well as the issue of scalars versus objects). Java pools its String literals. The effect of this is that if you use the String literal "jeff" in two places, Java will not make two String objects containing the string "jeff", it will instead use the same object twice, giving you two references to the same object. Consequently, in order to demonstrate that two different String objects can contain the same String values, you can't simply make two Strings objects with equal values directly from String literals, because they will turn out to have the same references as well and "==" will behave like "equals" in that case.

Strings created by the use of a String literal undergo a process known as interning. That is, the compiler will check to see if that String has already be created, and if it has, returns a reference to the original String rather than to a new one. This only really bothers us if we use the equality operator '==' which tests if the objects compared are actually the same object.

In a more general context, if you want to test that two objects of the same class are 'equal' you have to write your own equals() method to override the one defined in the Object superclass.

4. CLASS VARIABLES AND CLASS METHODS

4.1 Class Variables and Class Methods Explained

Class (static) variables are a kind of global variable. They are available to every object instanced from the class. Java requires every variable to be declared within a class and so uses the keyword static in front of a variable to indicate that the variable is to be shared by all instances of the class. No matter how many instances there are of a class there is only one copy of the static (class) variable.

Class (static) methods are methods that are sent to classes. They are messages sent to classes rather than to instances of the class. It is the class that understands the message. An object will not understand the (class) method.

```
public class DemoClass {
// Class variables and constants
    private static final int MAX_OBJECTS = 3;
    private static int objectCount = 0;
    // Instance variables
    private int objectNumber;
    // possibly  more instance variables...

    public DemoClass() {
        if (this.canCreateObject()) {
            objectNumber = ++objectCount;
            // initialisation of other instance variables...
        }
    }

    // class methods
    public static int getObjectCount() {
        return objectCount;
    }

    public static boolean canCreateObject() {
        return objectCount <= MAX_OBJECTS;
    }

    public static void report() {
        System.out.println("Too many objects...");
    }

    // instance methods
    public int getNumber() {
        return objectNumber;
    }

    public void whoAmI() {
        System.out.println("You are object number " + this.getNumber());
```

The keyword **final** makes MAX_OBJECTS a constant. Note when the class variable is initialised… it cannot be done in the constructor otherwise initialisation would be done every time a new object is instanced!

```
    }
}
public class TestDemo{
    public static void main(String args[]) {
        int count;
        DemoClass obj1, obj2, obj3,obj4;
        obj1 = new DemoClass();
        if (!DemoClass.canCreateObject()) DemoClass.report();
        obj2 = new DemoClass();
        if (!DemoClass.canCreateObject()) DemoClass.report();
        count = DemoClass.getObjectCount();
        System.out.println("Current number of objects = " + count);

        obj3 = new DemoClass();
        if (!DemoClass.canCreateObject()) DemoClass.report();

        count = DemoClass.getObjectCount();
        System.out.println("Current number of objects = " + count);

        obj2.whoAmI();

        obj4 = new DemoClass();
        if (!DemoClass.canCreateObject()) DemoClass.report();
        else obj4.whoAmI();
    }
}
```

canCreateObject() and report() are class methods and hence the receiver of these methods is the Class DemoClass

4.2 Class (static) methods can only make changes to class variables

Class (static) methods can only make changes to class variables (static variables) or variables local to the class method. In other words a class method cannot change an instance variable. If you try to do something like

```
    public static void incGlobalVar(int n) {
        globalVar += n;
        instanceVar += n;
    }
```

You will get an error message,
Can't make a static reference to nonstatic variable instanceVar in class DemoClass.

```
        instanceVar += n;
        ^
```

This seems reasonable... we cannot have a class changing the state of any of its instances

4.3 Class Methods in The Java Classes

There are many examples of class methods throughout the Java classes, so be aware of them and use them. For example if you wanted to convert an integer to a string you might do something like the code below

```
public class A {
    public static void main(String args[]) {
        int x = 864;
        String aString = String.valueOf(x);
        System.out.println(aString.length());
    }
}
```

If you look in the java.lang package at the String class you will find an entry as follows,

public static String valueOf(int i)
Returns a String object that represents the value of the specified integer.
Parameters:
i - the integer

This tells us that the method **valueOf**() is a class (static) method and is used as the Java code above demonstrates.

4.4 The System.out Class Variable

A familiar example of a class variable can be seen in the line of Java code

```
System.out.println(aString);
```

the variable **out** is a static (class) variable of the System class. If you look at its definition within the System class you will see that it is defined as a PrintStream type. That is **out** is defined as an instance of the PrintStream class. Hence **out** is a PrintStream object to which we send the println() message. If we write the line as

```
(System.out).println(aString);
```

then maybe the point is further made. As a general rule remember that the dot operator is evaluated left to right.

5. THE VECTOR CLASS

5.1 Vectors are Like Arrays, but…

The Vector class implements a growable array of objects. Like an array, it contains components that can be accessed using an integer index. However, the size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created. The Vector class operates by creating an initial storage capacity and then adding to this capacity as needed. The access methods provided by the Vector class support array-like operations and operations related to the size of Vector objects. The array-like operations allow elements to be added, deleted and inserted. They also allow tests to be performed on the contents of Vectors and specific elements to be retrieved.

A Vector can be used to store different Object types. Clearly accessing and processing the elements then becomes a more complex task as the type of an element has to first of all be determined before sensible processing can be carried out. All in all a very useful data structure…

5.2 A Simple Vector Example

```
public class Student {
    private String name;
    private String pin;

    public Student(String aName, String aPin) {
        name = aName;
        pin = aPin;
    }

    public String getName() {
        return name;
    }

    public String getPin() {
        return pin;
    }
}


import java.util.*;
public class Register {
    private Vector theList;
```

```
    public Register() {
        theList = new Vector();
    }

    public void addStudent(Student aStudent) {
        theList.addElement(aStudent);
    }

    public boolean removeStudent(Student aStudent) {
        return theList.removeElement(aStudent);
    }

    public int getPositionOf(Student aStudent) {
        int thePosition = theList.indexOf(aStudent);
        return thePosition;   // -1 if not found...
    }

    public void show() {
        Student nextStudent;
        for (int i = 0; i < theList.size(); ++i) {
            nextStudent = (Student)theList.elementAt(i);
            System.out.println(nextStudent.getName());
        }
    }

    public static void main(String arg[]) {
        Register rg = new Register();
        Student jeff = new Student("Jeff","1234");
        Student pat = new Student("Pat","4321");
        Student john = new Student("John","4567");
        Student debbie = new Student("Debbie","6789");
        rg.addStudent(jeff); rg.addStudent(pat);
        rg.addStudent(john); rg.addStudent(debbie);
        rg.show();
        if (rg.removeStudent(jeff)) System.out.println("Ok student removed");
        else System.out.println("Student not found...");
        rg.show();
        System.out.println("Position of Pat = " + rg.getPositionOf(pat));

        System.out.println("Position of Jenny = " + rg.getPositionOf(new Student("jenny","2222")));
    }
}
```

See later comments on this indexOf() method

A vector stores its elements as Object type. It doesn't matter what type the incoming object is, it is stored as an Object type. This means that when we retrieve an object from a Vector we have to cast the object back to the 'original' type. What happens if we don't know what type to cast to??? see the Wrapper example later...

5.3 The elements() Method

One of the methods of the Vector class deserves some special mention. Consider this alternative implementation of the show() method from the above Register class.

```
    public void show() {
        Student nextStudent;
        Enumeration e = theList.elements();
        while (e.hasMoreElements()) {
            nextStudent = (Student)e.nextElement();
            System.out.println(nextStudent.getName());
        }
    }
```

The elements() method of the Vector class produces an Enumeration object. The Enumeration object returns with a list of the elements of the vector, in this case the list of names stored in the vector. The Enumeration object (e) has two methods to allow us to access the list,

public boolean hasMoreElements()
Tests if this enumeration contains more elements (names in this case)
Returns true if this enumeration contains more elements (names),
false otherwise.

public Object nextElement()
Returns the next element (name) of this enumeration.

These two methods allow the elements of the vector object to be scanned as the above example demonstrates.

5.4 Use of Object Wrappers with Vectors

Vectors store objects. Hence when using a vector to store scalars (int, chars etc) a number of problems arise in that the scalars have to be converted to objects before adding to the vector and then converted from objects back to scalars when they are removed from the vector. This is where the object wrapper facility is vital.

```
import java.util.*;
public class WrapperDemo {
    public static void main(String args[]) {
        Vector theVector = new Vector();
        String aString = new String("Jeff");
        // theVector.addElement(5);
        theVector.addElement(new Integer(5));
        theVector.addElement(new Character('A'));
        theVector.addElement(aString);

        Object obj;
        Class aClass;
        String str;
        int anInt;
        for (int i = 0; i < theVector.size(); ++i)  {
            obj = theVector.elementAt(i);
            aClass = obj.getClass();
            str = aClass.getName();
            System.out.println("Object type is -> " + str);
            if (str.equals("java.lang.Integer")) {
                anInt = ((Integer)theVector.elementAt(i)).intValue();
                System.out.println("Some arithmetic... " + (2*anInt));
            }
        } // end for
    }
}
```

Can't do this since a Vector stores objects and 5 is a scalar

So we have to wrap the scalar within an Integer object

Likewise with a char scalar...

But… we're ok with an object…

the lines

```
obj = theVector.elementAt(i);
aClass = obj.getClass();
str = obj.getClass().getName();
```

need some further explanation…

- theVector.elementAt(i) returns the object at the ith position in the Vector
- this object is then sent the message getClass() which returns the class of the object as a Class type
- this Class type is then sent the message getName() which returns the name of the class as a String object type.

Output from a run was...
```
    Object type is -> java.lang.Integer
    Some arithmetic... 10
    Object type is -> java.lang.Character
    Object type is -> java.lang.String
```

5.5 The Vector and the equals() method

Consider the method from the Register class. It makes use of the indexOf() method from the Vector class (theList was defined as... theList = new Vector()).

```
    public int getPositionOf(Student aStudent) {
        int thePosition = theList.indexOf(aStudent);
        return thePosition;   // -1 if not found...
    }
```

The Java API entry for the indexOf() method from the Vector class is as follows,

```
 public final int indexOf(Object elem)
    Searches for the first occurrence of the given argument, testing for equality using the equals method.
    Parameters:
        elem - an object.
    Returns:
        the index of the first occurrence of the argument in this vector; returns -1 if the object is not found.
```

This implies that the indexOf() method searches the element of the Vector to find if it's argument is stored in the Vector. It does this by comparing the argument with each element in turn until it finds a match or it reaches the end of the Vector. This means that it has to compare for equality between two stored objects. Hence it has to be able to compare two Student objects to see if they are the same. If two Student objects are to be compared then we must include an equals() method as part of the Student class interface. All classes in the Java API are provided with a 'default' equals() method. They all inherit one from the Object superclass. However for the equals() method to work properly for any particular class that class has to override the method from the Object class. The reason for this is that the inherited equals() does not really work the way we might reasonably expect it to.

The version of equals() inherited from the Object class does not compare the states of two objects, in this case two Student objects, but simply compares if they are the same object. It works like the == operator in that it simply tests if the variables referencing the two objects are equal. That is, do the two variables point to the same object. This can be confirmed by the following example.

```
import java.util.*;
public class EqualsTester {
    public static void main(String arg[]) {
        Student x = new Student("Jeff","1234");
        Student y = new Student("Jeff","1234");
        // Before we add an equals() method to the Student class...
        // the output is ????
        // After we add an equals() method to the Student class...
        // the output is Yes... the same student...
        if (x.equals(y)) System.out.println("Yes... the same student...");
        else System.out.println("????");
    }
}
```

The equals() method that we need to include in the Student class is,

```
    public boolean equals(Object obj) {
        Student aStudent = (Student)obj;
```

```
        return    (   this.getName().equals(aStudent.getName()) &&
                      this.getPin().equals(aStudent.getPin()));
    }
```

Generally in software applications when we ask if two 'things' are equal we usually mean do they have the same value? Now in object technology we transfer the word value to the word state. If we now ask are two objects equal we generally mean do they have the same state. Two objects will be equal if their corresponding instance variables are equal. In 'everyday' terms we would expect the question… does Student x equal Student y… to be yes.

In developing an equals method for the Student class we may have been tempted to write something like

```
    public boolean equals(Student aStudent) {
        return    (   this.getName().equals(aStudent.getName()) &&
                      this.getPin().equals(aStudent.getPin()))
    }
```

This however will not work in the context that we want it to. Certainly it will return true or false depending whether the two students are the 'same' or not. We have not overridden the equals() method from the Object class but simply given the Student class a new method calls equals. The failure is due to the fact that the indexOf() method will search the Student class for an equals() method that takes an Object as an argument. This is how it was designed by the Sun Java engineers. It will not find one and therefor will search the superclass (Object) where it will find the matching equals() method and hence we are back to the original problem. So the moral is... we must override the equals() method not just overload it .

5.6 The Banking Classes Updated to Use Vectors and Encapsulation

Since we have replaced the array data structure with a Vector data structure there is no need to carry an instance variable to count the current numBranches, numCustomers and numAccount is the respective classes. These values can be retrieved when required by using the size() method from the Vector class. This method returns the current number of elements stored in the Vector.

5.6.1 The Account Class

```
public class Account {
    private String accNumber;
    private int balance;

    public Account(String accN,int initBal) {
        accNumber = accN;
        this.setBalance(initBal);
    }

    public String getAccountNumber() {
        return accNumber;
    }

    public void setBalance(int anAmount) {
        balance = anAmount;
    }

    public int getBalance() {
        return balance;
    }

    public void credit(int anAmount) {
        this.setBalance(this.getBalance() + anAmount);
    }

    public boolean debit(int anAmount) {
```

```
        if (this.getBalance() - anAmount >= 0) {
            this.setBalance(this.getBalance() - anAmount);
            return true;
        }
        else return false;
    }
}
```

5.6.2 The Customer Class

```
import java.util.*;
public class Customer {
    private String theName;
    private Vector theAccountsList;

    public Customer(String aName) {
        theName = aName;
        theAccountsList = new Vector();
    }

    public String getName() {
        return theName;
    }

    public int getNumAccounts() {
        return theAccountsList.size();
    }

    public void addAccount(Account anAccount) {
        theAccountsList.addElement(anAccount);
    }

    public Vector getAccounts() {
        return theAccountsList;
    }

    public Account getAccount(String anAccountNumber) {
        Enumeration accList = theAccountsList.elements();
        Account nextAccount = null;
        while (accList.hasMoreElements()) {
            nextAccount = (Account)accList.nextElement();
            if (nextAccount.getAccountNumber().equals(anAccountNumber))
                return nextAccount;
        }
        return nextAccount;
    }
}
```

5.6.3 The Branch Class

```
import java.util.*;
public class Branch {
    private static int RATE = 5;
    private static int OD_LIMIT = 100;

    private String theName;
    private Vector theCustomerList;
    private String nextAccountNumber;

    public Branch(String aName, String startAccount) {
```

```
        theName = aName;
        theCustomerList = new Vector();
        nextAccountNumber = startAccount;
    }

    public String getName() {
        return theName;
    }

    public void addCustomer(Customer aCustomer) {
        theCustomerList.addElement(aCustomer);
    }

    public int getNumCustomers() {
        return theCustomerList.size();
    }

    public Vector getCustomers() {
        return theCustomerList;
    }

    public void createAccount(Customer aCustomer) {
        Account aNewAccount = new Account(nextAccountNumber,0);
        aCustomer.addAccount(aNewAccount);
        int tempInt = new Integer(nextAccountNumber).intValue();
        ++tempInt;
        nextAccountNumber = new Integer(tempInt).toString();
    }

    public void credit(String anAccountNumber, int anAmount) {
        Account targetAccount = this.getAccount(anAccountNumber);
        targetAccount.credit(anAmount);
    }

    public boolean debit(String anAccountNumber, int anAmount) {
        Account targetAccount = this.getAccount(anAccountNumber);
        boolean debitOk = targetAccount.debit(anAmount);
        return debitOk;
    }

    public Account getAccount(String accNumber) {
        Account aTempAccount = null;
        Customer nextCustomer;
        Enumeration custList = theCustomerList.elements();
        while (custList.hasMoreElements()) {
            nextCustomer = (Customer)custList.nextElement();
            aTempAccount = nextCustomer.getAccount(accNumber);
            if (aTempAccount != null) break;
        }
        return aTempAccount;
    }



    public Customer getCustomer(String aName) {
        Customer nextCustomer = null;
        Enumeration e = theCustomerList.elements();
        while (e.hasMoreElements()) {
            nextCustomer = (Customer)e.nextElement();
```

```
            if (aName.equals(nextCustomer.getName())) {
                break;
            }
        }
        return nextCustomer;
    }

    public boolean transfer(Account fromAccount,Account toAccount,int anAmount) {
        if (fromAccount.debit(anAmount)) {
            toAccount.credit(anAmount);
            return true;
        }
        else return false;
    }
}
```

5.6.4 The Bank Class

```
import java.util.*;
public class Bank {
    private String theName;
    private Vector theBranchList;

    public Bank(String aName) {
        theName = aName;
        theBranchList = new Vector();
    }

    public String getName() {
        return theName;
    }

    public void addBranch(Branch aBranch) {
        theBranchList.addElement(aBranch);
    }

    public Enumeration getBranches() {
        return theBranchList.elements();
    }

    public int getNumBranches() {
        return theBranchList.size();
    }
}
```