Programming In Java

BOOK 2

Inheritance and Data Types

CONTENTS

1. CLASS HIERARCHY AND INHERITANCE

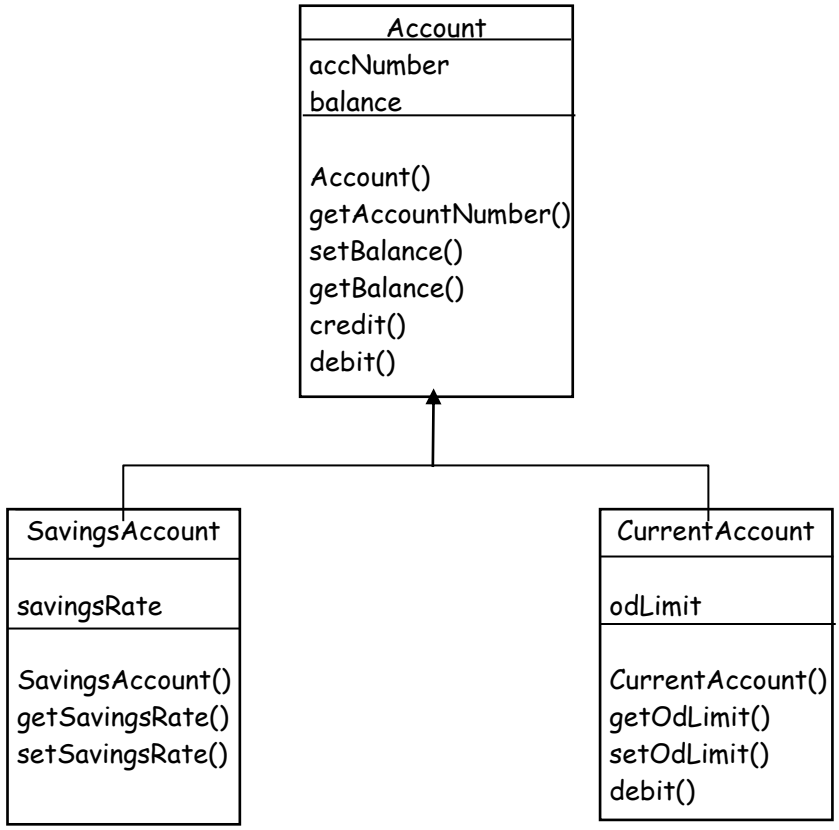
At the root of the Java Class hierarchy is the class Object. Every Class in the system has Object as its ultimate parent. Every method defined in the Object class is available in every Java object. It provides a set of basic behaviors to all objects, but no basic state as there are no variables declared in the Object class.

1.1 Extending The Account Class and Inheritance

We now take a closer look at the Account class. In banking terms, Account is a rather general description. Normally we think in terms of a particular type of account. A savings account, is a special type of account, just as a current account is a special type of account. Whilst both types of account have differences they also have common properties and behaviour, such as both possessing a current balance and a reference number, and both responding to a credit operation or a debit operation.

The idea of specialisation introduces a very important feature of object technology***... inheritance.*** Inheritance is a major feature. It allows new classes, called subclasses, to be constructed from existing classes, the superclasses. The derived subclass inherits ALL the properties and methods of the superclass and adds new characteristics of it's own. In practical terms this means that when a subclass is created, the properties and methods that belong to the superclass do not have to be generated again. Data and methods of the superclass become available to the subclass(es). This has major implications and benefits for the fast production of reliable software systems. Software can be reused.

Using our general Account class we can create two specialised subclasses. A Savings account and a Current account. The inheritance relationships are illustrated with the following class hierarchy,



1.2 Changes to Some Existing Classes

To accommodate our new account class hierarchy we have made a number of changes to the existing Branch class. Since there are now two possible types of account that a Customer can hold this forces a slight change in the addCustomer() and createAccount() methods. Both need modifications to handle the possible different types of account.

1.2.1 The Revised Branch Class

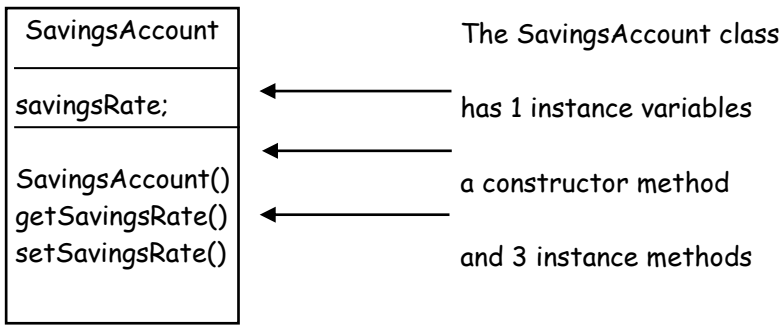
The Branch class changes are shown in bold. We have taken the opportunity to add some constants to the class. They represent the initial savings rate applied to new Savings accounts and the initial overdraft limit applied to new Current accounts.

```
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, char accType) {
        theCustomerList.addElement(aCustomer);
        this.createAccount(aCustomer,accType);
    }

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

    public Enumeration getCustomers() {
        return theCustomerList.elements();
    }

    public void createAccount(Customer aCustomer, char type) {
        Account aNewAccount = null;
        int tempInt = new Integer(nextAccountNumber).intValue();
        ++tempInt;
        nextAccountNumber = new Integer(tempInt).toString();
        switch (type) {
            case 'S' :    aNewAccount = new SavingsAccount(nextAccountNumber,0,RATE);
                          break;
            case 'C' :    aNewAccount = new CurrentAccount(nextAccountNumber,0,OD_LIMIT);
                          break;
        }
        aCustomer.addAccount(aNewAccount);
    }

    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;
    }
}
```

1.3 The Two Account Subclasses

1.3.1 The SavingsAccount Class

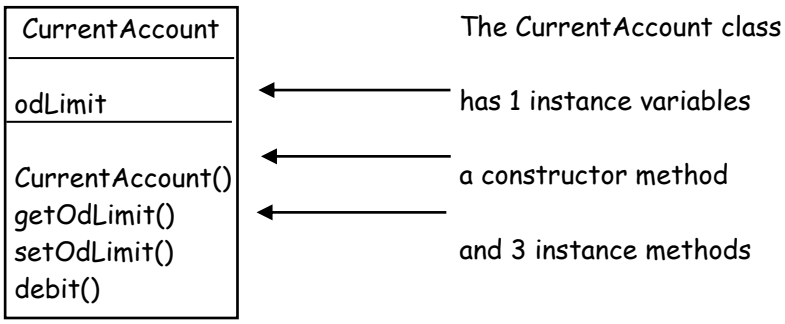



```
public class SavingsAccount extends Account {
    private int savingsRate;
```

```
    public SavingsAccount(String accN,int initBal,int aRate) {
        super(accN,initBal);
        this.setSavingsRate(aRate);
    }

    public int getSavingsRate() {
        return savingsRate;
    }

    public void setSavingsRate(int aRate) {
        savingsRate = aRate;
    }
}
```

Here we use encapsulation to set the savings rate, rather than attempt to refer directly to the savingsRate instance variable... see later comments on this

1.3.2 The CurrentAccount Class




```
public class CurrentAccount extends Account {
    private int odLimit;

    public CurrentAccount(String accN,int initBal,int aLimit) {
        super(accN,initBal);
        this.setOdLimit(aLimit);
    }

    public int getOdLimit() {
        return odLimit;
    }

    public void setOdLimit(int aLimit) {
        odLimit = aLimit;
    }

    public boolean debit(int anAmount) {
        int currentBalance = this.getBalance();
        if (currentBalance + odLimit - anAmount >= 0) { // can't exceed odLimit...
            currentBalance -= anAmount;
            this.setBalance(currentBalance);
        return true;
        }
        else return false;
    }
}
```

1.4 Overriding Methods - The debit() Method

The debit() method is an example of **method overriding**. It appears in the Account class and the CurrentAccount class. It is defined in the Account superclass but the CurrentAccount subclass requires it's own specialised version, since the rules for a debit on a savings account are different to the rules for a debit on a current account. The CurrentAccount subclass implements the debit() method in a way that is appropriate to it's own requirements. You could view the Account superclass as a kind of abstract class in the sense that we will never actually create, that is use, an instance of the Account class. We are only interested in it's properties so that we can re-use them to create the classes Savings and Current, which will actually be instantiated. Another way of saying this is that we will never use an Account object. Our banking system will only ever produce either Savings account objects or Current account objects. Java uses the ideas of abstractions and makes use of classes called abstract classes. An abstract class type is a class specifically designed to provide inheritable characteristics for it's descendent class types. The purpose of an abstract class is to have descendants and not instances. Note that Account is not really a 'true' Java abstract class. We have only drawn some comparisons. True abstract classes will be introduced later in these notes.

From the Account class...

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

From the CurrentAccount class...

```
public boolean debit(int anAmount) {
        int n = anAmount - odLimit;
        return super.debit(n);



        int currentBalance = this.getBalance();

        if (currentBalance + odLimit - anAmount >= 0) { // can't exceed odLimit...
            currentBalance -= anAmount;
            this.setBalance(currentBalance);
        return true;
        }
        else return false;
    }
```

1.5 Testing The System

```
import java.util.*;
public class TestBank {
    private Bank theBank;

    public TestBank() {
        // Create the bank...
        theBank = new Bank("NatWest");

        // Next create two branches...
        Branch blackBurn = new Branch("Blackburn","1000");
        Branch burnley = new Branch("Burnley","4000");
        // Add the branches to the Bank...
        theBank.addBranch(blackBurn);
        theBank.addBranch(burnley);
        // Now create some customers...
        Customer customerJeff = new Customer("jeff");
```

```
        Customer customerPat = new Customer("pat");
        // Add the customers to the branches...
        blackBurn.addCustomer(customerJeff,'S');
        burnley.addCustomer(customerPat,'C');
        // Create some extra accounts for the customers...
        blackBurn.createAccount(customerJeff,'C');
    // credit some accounts...
        blackBurn.credit("1001",300);

        this.displayBankDetails();

        // Test the Branch transfer method...
        Account fromAccount = blackBurn.getAccount("1001");
        Account toAccount = burnley.getAccount("4001");
        boolean transferOk = blackBurn.transfer(fromAccount,toAccount,100);
        this.displayBankDetails();
    }

    public void displayBankDetails() {
        // Now display the details of the Bank
        System.out.println("----------------------------------------");
        System.out.println(theBank.getName());
        Enumeration theBranches = theBank.getBranches();
        Branch nextBranch;
        String nextBranchName;
        // Process each branch in turn...
        while (theBranches.hasMoreElements()) {
            nextBranch = (Branch)theBranches.nextElement();
            nextBranchName = nextBranch.getName();
            System.out.println("\t" + nextBranchName);
            Enumeration theCustomers = nextBranch.getCustomers();
            Customer nextCustomer;
            // Process each customer in turn
            while (theCustomers.hasMoreElements()) {
                nextCustomer = (Customer)theCustomers.nextElement();
                System.out.println("\t\t" + nextCustomer.getName());
                // Process each account in turn
                Account nextAccount;
                Enumeration theAccounts = nextCustomer.getAccounts();
                while (theAccounts.hasMoreElements()) {
                    nextAccount = (Account)theAccounts.nextElement();
                    this.displayAccountDetails(nextAccount);
                } // end of accounts
            } // end of customers
        } // end of branches
    }

    public void displayAccountDetails(Account anAccount) {
        String accType = anAccount.getClass().getName();
        System.out.print("\t\t\t" + accType + " : " + anAccount.getAccountNumber() + " : " +
anAccount.getBalance());
        if (accType.equals("SavingsAccount")) {
            SavingsAccount theAccount = (SavingsAccount)anAccount;
            System.out.println(" : " + theAccount.getSavingsRate());
        }
        else {
            CurrentAccount theAccount = (CurrentAccount)anAccount;
            System.out.println(" : " + theAccount.getOdLimit());
        }
    }
```

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

1.5.1 Exercise

Complete the testing of the system by adding new code to the bank test class. In particular the debit() methods of the Savings account class and the Current account class need to be fully tested.

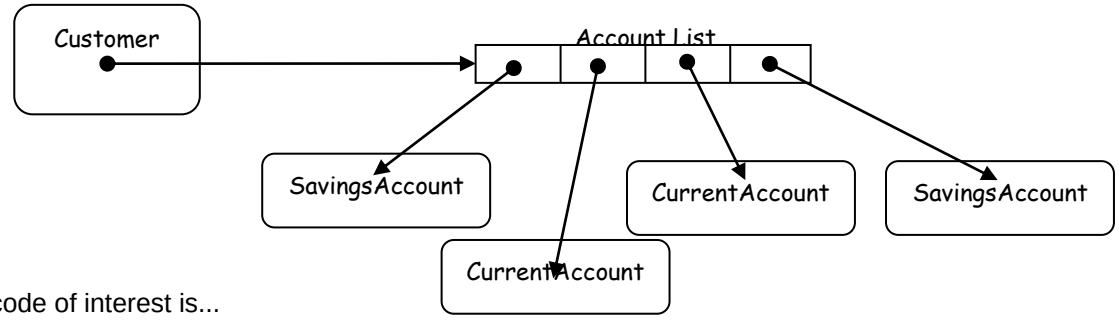
1.6 Substitutability

Substitutabilty means that an object of one type is legitimately substitutable for an object of another type. This is allowed in Java when objects are related through inheritance. An object of the child class may be substituted for an object of the parent class. This is because the subclass object has everything that the superclass object would have, plus some possible extra features as well. So using the subclass object as if it were a superclass object won't cause any problems. This property of Java (and other object-oriented languages) is sometimes known as *subtype polymorphism*.

public class A {
  public void m1(A a) {}
}
	public class B extends A{}	public class Test {
		  public static void main(String s[]) {
		  A a = new B();
		  B b = new B();
		  a.m1(b);
		  }
		}

1.6.1 An Example From The Banking System

There are some interesting happenings in the BankTest class...
As we are retrieving the accounts for a particular customer from the Vector of accounts, we have to note that the list may be a mixture of SavingsAccount objects and CurrentAccount objects. So each account that we retrieve may be of one kind or another.



The code of interest is...

```
Account nextAccount;
Enumeration theAccounts = nextCustomer.getAccounts();
      while (theAccounts.hasMoreElements()) {
            nextAccount = (Account)theAccounts.nextElement();
            this.displayAccountDetails(nextAccount);
      } // end of accounts
```

here we are retrieving the account objects from the customer account list. Within the loop each account is referenced by nextAccount. Whether it is a SavingsAccount object or a CurrentAccount object is sorted out in the called method displayAccountDetails().

```
    public void displayAccountDetails(Account anAccount) {
        String accType = anAccount.getClass().getName();
        System.out.print("\t\t\t" + accType + " : " + etc...;
        if (accType.equals("SavingsAccount")) {
            SavingsAccount theAccount = (SavingsAccount)anAccount;
            System.out.println(" : " + theAccount.getSavingsRate());
```

```
        }
        else {
            CurrentAccount theAccount = (CurrentAccount)anAccount;
            System.out.println(" : " + theAccount.getOdLimit());
        }
    }
```

A key line is... String accType = anAccount.getClass().getName();
This line sets the string accType to either "SavingsAccount" or "CurrentAccount".
Once we have done this we can do the necessary casting and send appropriate messages to the relevant object.

1.6.2 Another Example of Substitution

In the TestBank class the following code is used to make a transfer from account 1001 which is a Savings account to account 4001 which is a Current account.

```
 // Test the Branch transfer method...
Account fromAccount = blackBurn.getAccount("1001");
Account toAccount = burnley.getAccount("4001");
boolean transferOk = blackBurn.transfer(fromAccount,toAccount,100);
this.displayBankDetails();
```

In the Branch class the method getAccount() is defined as returning an Account object, but in the code above it actually returns with a Savings account object in the first line and a Current account object in the second line.

Also the transfer() method is defined in the Branch class as taking two Account objects as arguments, but here it takes a Savings account object and a Current account object....

There is more on this issue of substitutability later in the chapter.

1.7 More on Encapsulation (Accessor Methods vs Direct Reference)

You may have noticed that the instance variables in the Account superclass have been declared as private... so how then do the subclasses gain access to these variables.

```
public class Account {
    private int accNumber;
    private int balance;
    private char type;
```

There is some debate within the object world that accessing instance variables directly may be ‘bad practice’ and accessor methods should always be used to set and return instance variables values. The argument for using accessor methods, often referred to as setter and getter methods, rather than direct access is based around encapsulation and 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.

A policy adopted by many developers is to use direct reference to instance variables only in the accessor methods themselves. Any other references, ie in the other methods, will only be through the accessor methods. Hence updates will be focused on these setter and getter methods.
Put another way… any direct reference to an instance variable other than in a setter/getter method, should be done through the accessor method.

Adopting a policy like this means that when any updates are required we know exactly where to look to make the changes. System built with this policy will exhibit strong encapsulation.

Encapsulation also ensures that an object's state is changed in a manner known to the object. When we use a method to effect a change to a state (instance) variable the object is aware of the change and may wish to modify other parts of it's state. Accessing the instance variable directly is not likely to have the same affect on changing other state variables.

Many OO writers suggest that 'normal' practice when building a family of inherited classes is to make instance variables *protected*… giving sub-classes direct access to super class instance variables. However there are good reasons in many cases to drop this practice and to make super class instance variables *private* and to provide sub-classes with access to these super class instance variables through 'accessor' methods. Any internal data (attribute) changes to super classes should then leave sub-classes without the need for change. If in a sub-class we make direct reference to a (protected) super class instance variable and then change the internal representation of the super class instance variable we may then have to make corresponding changes in the sub-classes. If on the other hand, sub-classes do not make direct reference to super class instance variables then no changes should be necessary… providing of course that the super class interface (ie the methods) remains the same.

If you now look carefully at the debit() method in the Savings account class you will see encapsulation being used on the balance instance variable,

```
public boolean debit(int anAmount) {
        int currentBalance = this.getBalance();
        if (currentBalance - anAmount >= 0) {  // savings accounts cannot be negative...
            currentBalance -= anAmount;
            this.setBalance(currentBalance);
        return true;
        }
        else return false;
    }
```

Here we use encapsulation to access the balance variable, rather than attempt to refer directly to the balance.

If for some reason we don't want to use encapsulation (???) then in order to be able to refer to the instance variables directly in the subclasses we would have to declare the instance variables as protected...

```
public class Account {
    protected int accNumber;
    protected int balance;
    protected char type;
```

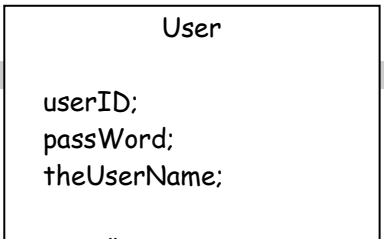
Unfortunately Java does not enforce encapsulation... it does allow you to access the variables directly... without using the appropriate method.

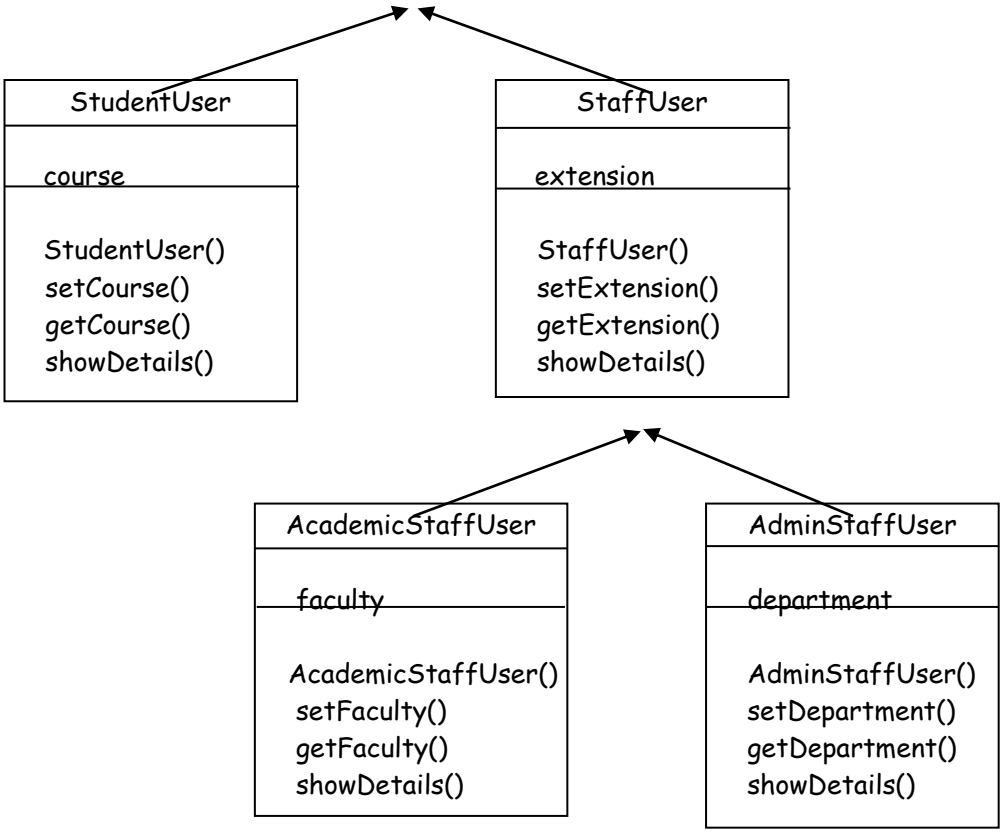
```
public class Demo {
    public int x;
    public Demo(int p) {
        x = p;
    }

    public int getX() {
        return x;
    }
}
```

```
public class DemoTest {
    public static void main(String args[]) {
        Demo anObj = new Demo(6);
        System.out.println(anObj.x);
    }
}
```

1.8 Another Example of Class Hierarchy

In this class hierarchy the class User is the superclass of the two subclasses StudentUser and StaffUser. The class StaffUser is the superclass of the subclasses AcademicStaffUser and AdminStaffUser. Notice that the superclass serves as a general description and that the subclasses are more specialised. This means for example, that all users have common properties and behaviour but additionally have requirements for specialised properties and specialised behaviour of their own.





The details of the code are as follows,

public class User {

```
    private String userID;
    private String passWord;
    private String theUserName;

    public User(String aName) {
        this.setUserId(aName);
        this.setPassWord();
        theUserName = aName;
    }

    private void setUserId(String str) {
        userID = str.substring(2);
    }

    public String getUserId() {
        return userID;
    }

    public void setPassWord() {
        // bit clumsy... we have to create a StringBuffer object from the
        // name string, reverse it and convert it back to a String object
        passWord = new StringBuffer(this.getUserName()).reverse().toString();
    }

    public String getPassWord() {
        return passWord;
    }

    public String getUserName() {
        return theUserName;
    }

    public void showDetails() {
        System.out.println("UserId     = " + this.getUserId());
        System.out.println("passWord   = " + this.getPassWord());
        System.out.println("User Name  = " + this.getUserName());
    }
}

public class StaffUser extends User {
    private int extension;

    public StaffUser(String aName, int anExtension) {
        super(aName);
        this.setExtension(anExtension);
    }
    public void setExtension(int anExtension) {
        extension = anExtension;
    }
,   public int getExtension() {
        return extension;
    }
    public void showDetails() {
        super.showDetails();
        System.out.println("Ext        = " + this.getExtension());
    }
}


public class AcademicStaffUser extends StaffUser {
```

```
    private String faculty;

    public AcademicStaffUser(String aName, int anExtension, String aFaculty) {
        super(aName, anExtension);
        this.setFaculty(aFaculty);
    }

    public void setFaculty(String aFaculty) {
        faculty = aFaculty;
    }

    public String getFaculty() {
        return faculty;
    }

    public void showDetails() {
        super.showDetails();
        System.out.println("Faculty = " + this.getFaculty());
    }
}
```

```
public class AdminStaffUser extends StaffUser {
    private String department;

    public AdminStaffUser(String aName, int anExtension, String aDepartment) {
        super(aName, anExtension);
        this.setDepartment(aDepartment);
    }

    public void setDepartment(String aDepartment) {
        department = aDepartment;
    }

    public String getDepartment() {
        return department;
    }

    public void showDetails() {
        super.showDetails();
        System.out.println("Admin Dept = " + this.getDepartment());
    }
}
```

```
public class StudentUser extends User {
    private String course;

    public StudentUser(String theName, String theCourse) {
        super(theName);
        this.setCourse(theCourse);
    }

    public void setCourse(String aCourse) {
        course = aCourse;
    }

    public String getCourse() {
        return course;
    }
```

```
    public void showDetails() {
        super.showDetails();
        System.out.println("Course       = " + this.getCourse());
    }
}
```

1.9 Polymorphism

An action/behaviour that is shared throughout a class hierarchy is given a name and each class implements that action in a way that is appropriate to itself.

1.10 Polymorphism Through Method Overriding

The method showDetails() is an example of overriding. Each sub class modifies the code for the method showDetails() and thus behaves in a different way. Each object type applies the same meaning to the method, but acts in a different way.

anAcademicUser.showDetails(); aStudent.showDetails(); anAdminStaff.showDetails();

Sometimes, as illustrated in the example, you may not want to override a superclass method ‘completely’ but may just want to add some additional features in a subclass version of the method. This is sometimes called overriding by 'refinement'.

1.11 Whats the Difference between Overriding and Overloading

Overriding is associated with class hierarchies. It means giving a method in a subclass the same signature as a method in a superclass and then redefining the method in the subclass so that the subclass objects respond differently to superclass objects. There is no change to the return type and the parameters.

Overloading is associated with methods within the same class. It means giving methods the same identifier (name) but changing the parameters either in type or in number. Java differentiates overloaded methods based on the number and type of parameters, not on the return type. That is, if you try to create two methods with the same name and same parameters but a different return type the compiler with report this as an error. Overloading is not used very often... except in the case of constructor overloading... it is quite common for a class to have a number of 'overloaded constructors'.

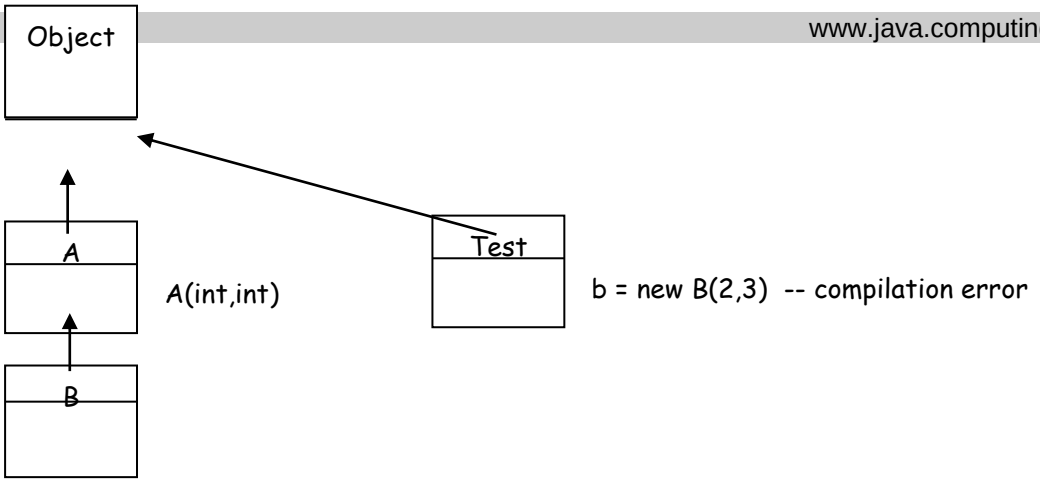
1.12 Constructors are not Inherited

1.12.1 Example 1

```
public class A {
    private int x, y;
    public A(int a, int b) {
        x = a; y = b;
    }
}
```

```
public class B extends A {

}
```

```
public class Test {
    public static void main(String args[]) {
        B b = new B(2,3);
    }
}
```



When Test.java was compiled the following error messages appeared,

```
Test.java:3: Wrong number of arguments in constructor.
        B b = new B(2,3);
              ^
.\B.java:1: No constructor matching A() found in class A.
public class B extends A {
           ^
```

So class B has not inherited the super class constructor

So what does this error message mean ??...

1.12.2 Example 2

If we now provide A with a zero argument constructor

```
public class A {
   private int x, y;
   public A(int a, int b) {
      x = a; y = b;
   }
   public A() {}
}
```

We now get a clean compile with test.java.
So we deduce that in the previous situation because B had no two int argument constructor defined (in fact it had no constructor defined), the compiler assumed it was looking for a zero argument constructor. It couldn't find one in class B so it looked in class A for a zero argument constructor A() and again it didn't find one. All it found was the two argument constructor which didn't match the zero argument constructor it was looking for.

1.13 A note on The Default Constructor

The default constructor is only provided if the class contains no constructor declarations. This default constructor is then provided in the sense that the compiler will look for a zero-argument constructor in the superclass, and provided one is found it will be invoked. If the superclass has no zero-argument constructor, a compile-time error may occur (this will happen, for example, if the superclass has no zero-argument constructor declaration, but has at least one other constructor declaration). If the superclass in turn has no constructor declarations, then the compiler will attempt to provide a default by looking in its superclass, and so on.

If no superclass has a constructor (of any kind) then eventually the Object class will be reached and the zero argument constructor of the Object class will be invoked.

If super() is not called in a subclass constructor, the compiler will begin a search, from that constructor, of the superclasses looking for a zero argument constructor. The search will stop when the first superclass is reached where there is

- either
  no non-zero argument constructor and a constructor with arguments, in which case a compile error occurs
- or

there is a zero argument constructor and the search stops

Note : If a subclass has more than one constructor, this searching process is carried out for each constructor.

1.14 Constructors are not Overriden

Since constructors not are inherited we cannot speak of overriding a constructor. In the example below the constructor of class A is not overriden in class B. The class B is simply defining a two argument constructor.

```
public class A {
    protected int x, y;
    public A(int a, int b) {
        x = a; y = b;
    }
}
```

```
public class B extends A {
    public B(int p, int q) {
        x = p; y = q;
        // more code...
    }
}
```

```
public class Test {
    public static void main(String args[]) {
        B b = new B(2,3);
    }
}
```

No compilation problems here...

If we want class B to have access to the constructor of class A then we have to make an explicit call to the superclass constructor with super(i,j). This is the usual way of doing things.

```
public class A {
    protected int x, y;
    public A(int a, int b) {
        x = a; y = b;
    }
}
```

```
public class B extends A {
    public B(int p, int q) {
        super(p, q);
        // more code...
    }
}
```

```
public class Test {
    public static void main(String
args[]) {
        B b = new B(2,3);
    }
}
```

No compilation problems here...

1.14.1 Constructors and Order of Intialisation In a Class Hierarchy

Every constructor invokes its superclass constructor first. This gives a chain of constructor calls up the class hierarchy to class Object, then completion of the constructors on the unwinding of the chain. The effect is to make an Object object, then to extend it down the levels until you have an object of the required class.

Not all of the superclass constructor calls are explicit. If there is no constructor in the code at all, the system adds a zero-argument constructor that just calls the superclass constructor. If there is a constructor, but the superclass constructor is not explicitly invoked at the start of it, then the system adds a call to the superclass constructor there.

The default behaviour for Java constructors is bizzare, and can often cause confusion. Basically the classloader initialises all local variables, then initialises variables in the superclass, executes the superclass constructor, before it calls it's own constructor.
This can (and does) lead to bizarre behavior. If you call local methods in your constructor, which are later overriden by a subclass, the superclass constructor would call the non overridden method, as the subclass would not be instantiated. You should get into the habit of only calling final methods from constructors, it'll save hours of hassle. In some senses this behaviour is good, because it does always mean that all objects are initialised before use.

1.15 Some Categories of Inheritance

Tim Budd in his book Understanding Object Oriented Programming with Java, identifies a number of different kinds of inheritance.

Inheritance for Extension (the usual one...)
This occurs when the child class only adds new behaviour to the parent class and does not modify or alter any of the inherited features.

Inheritance for Specialisation
The child class is more specialised than the parent class but still satisfies the specification of the parent's behaviour in all relevant aspects and therefore its objects are substitutable for those of the parent. The child class will override some of the parent's behaviour so that the behaviour is more specialised in certain respects.

Inheritance for Specification
The parent class defines the behaviour that is to be implemented in the child class but not in the parent class. In this case the parent class is an *interface* and the child class *implements* the parent rather than extending the parent.

Inheritance for Limitation
The child class restricts the use of some of the behaviour of the parent class. Probably by overriding methods as 'null' methods and therefore has less behaviour than the parent class.

Inheritance for Construction
The child class makes use of some of the behaviour of the parent class but is clearly not a sensible subtype of the parent class.

2. SCOPE - ACCESS MODIFIERS

2.1 Classes

The class modifiers are public, abstract and final. An abstract class provides an abstract class declaration that cannot be instantiated. Abstract classes are used as building blocks for the declaration of subclasses. A class that is declared as public can be accessed outside its package. If a class is not declared as public it can only be accessed within its package. A final class cannot be subclassed.

2.2 Instance Variables and Instance Methods

Java provides extensive and complex access mechanisms for classes, instance variables and methods. Here is just a taster...

- Any instance variable or method declared public becomes visible (ie accessible) to the class containing the declarations and to any other class. This is the most 'open' of all access modifiers.
- Any instance variable or method declared protected becomes visible (ie accessible) to the class containing the declarations, to any of its subclasses and to other classes in the same package.
- Any instance variable or method declared private is visible (ie accessible) only to the class containing the declarations. This is the most stringent modifier.

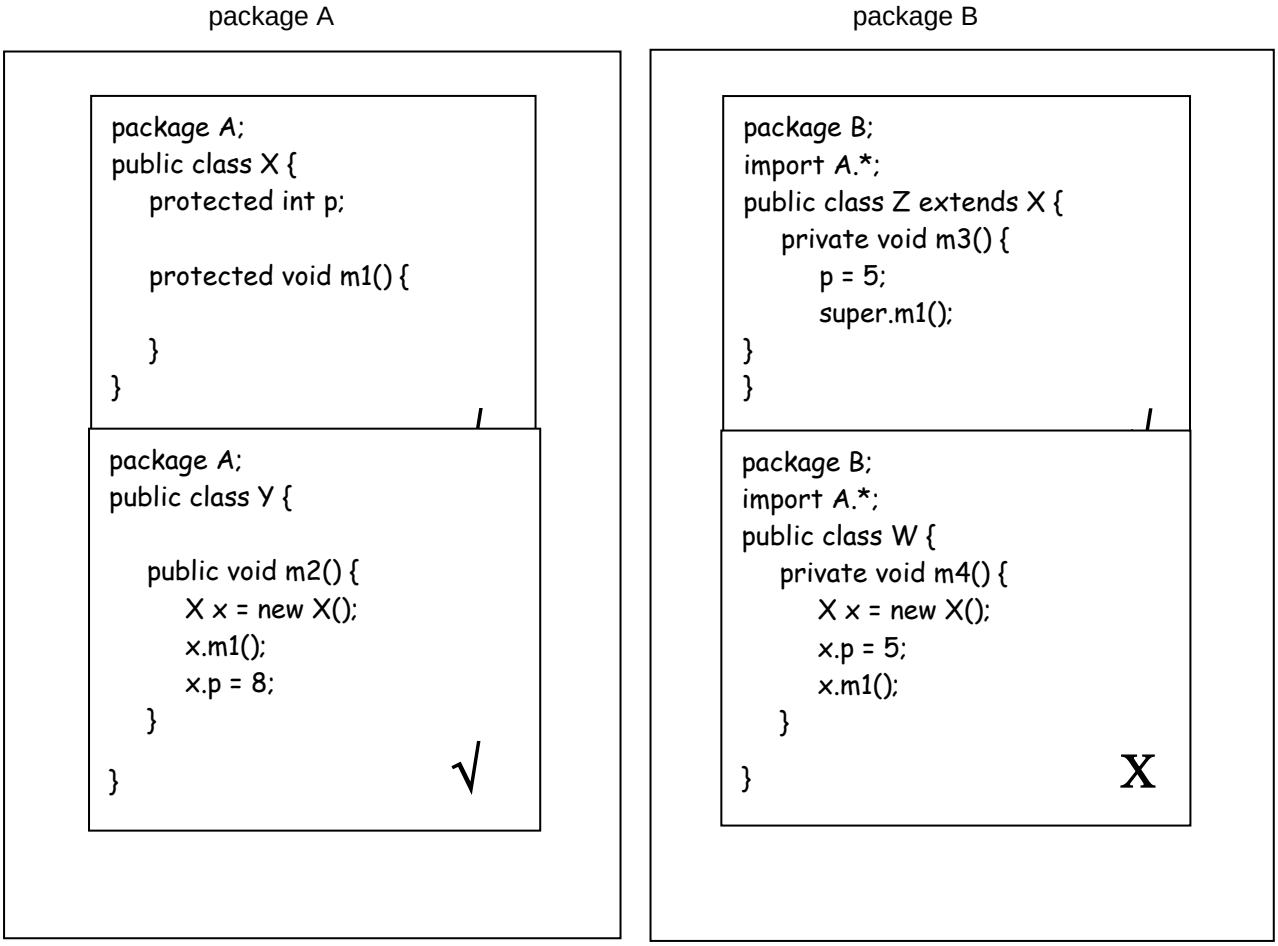
Since one of the key features in object technology is encapsulation, the norm is to make instance variables private and methods public.

Sometimes there are times when methods in a class are made private. This usually means that they perform some 'housekeeping' necessary only within the class. They are not visible outside the class and hence do not form part of the class interface. In other words you cannot send them as messages to objects. A term often associated with a private method is to call it a 'helper method'.

Note : if you leave out the modifier then you are effectively declaring the instance variable or method to have a default status... this means that only classes within the same package (a related group of classes) can see the variable or the method.

scope	instance variable	instance method	class variable	class
public	accessible in all classes	accessible in all classes	accessible in all classes	accessible in all classes
protected	accessible in the defining class, sub-classes, and classes in the same package	accessible in the defining class, sub-classes and classes in the same package	accessible in the defining class, sub-classes and classes in the same package	n/a
private	accessible only in the defining class	accessible only in the defining class	accessible only in the defining class	applies to inner classes
default	accessible to classes within the same package	accessible to classes within the same package	accessible to classes within the same package	accessible to classes within the same package
abstract	n/a	Must be implemented in a subclass	n/a	Must be inherited. Cannot be instantiated
final	n/a	Cannot be overridden	Serves as a constant	cannot be subclassed

The protected case is illustrated in the following example,



3. JAVA APPLICATIONS AND COMMAND LINE ARGUMENTS

The method main() is the first method to be executed in any Java application. All Java applications contain a main() method as the starting point for the application. Like all Java methods, main() can be passed

parameters. When a Java application is run the application can be passed parameters at the command level. These parameters are then available for use and processing within the application. It is possible to pass parameters to applications by appending them to the command line when you run your Java application. For example if we have a Java application that uses three files, then to run the application we might do something like,

java **concat** Pascal.doc Delphi.doc Update.doc

The intention perhaps being that files Pascal.doc and Delphi.doc are to be concatenated by the application to form the new Update.doc. The three file names can then be used by the application code.

```
public class ArgDemo {
    public static void main(String arg[]) {
        if (arg.length > 0) {
            for (int i = 0; i < arg.length; ++i)
                print(arg[i]);
        }
        else print("No passed arguments...");
    }

    private static void print(String msg) {
        System.out.println(msg);
    }
}
```

The command line, **java ArgDemo one two three**, then prints

one
two
three

the command line **java ArgDemo** prints

No passed arguments...

4. SOME BITS AND PIECES

4.1 Finalizer Methods

A finalizer method is the opposite of a constructor. A finalizer method is called just before the object is destroyed by the garbage collector of Java.

```
protected void finalizer() {
    // Code you want to carry out out just before the object is destroyed...
    // Often some sort of cleaning up code
}
```

Protected is the normal access modifier for this method since we only want it to be available to objects of the class (or subclasses).
One word of warning when using finalizer() methods... they will only get called when the Java garbage collector runs. This means that they may not execute when you expect them to. If you want your tidy up code to be executed at the particular point in your program sequence you may be better to have a method cleanUp() explicitly for this purpose.

4.2 Reference and Value Parameters

You should remember that when you pass an object as a parameter to a method you are in fact passing a reference to the object, not the object itself, nor a copy of the object. This of course means that anything you do to the object within the method affects the 'original' object outside the method, because you are dealing with the same object. Primitive types (int, float, char etc) are *passed by value* since java does not treat these scalars as objects. This is illustrated in the following example.

```
import java.util.Stack;
public class DemoClass {
   public void doSomething(StringBuffer target) {
      target.reverse();
```

```
class Tester {
    public static void main(String args[]) {
        DemoClass d = new DemoClass();
        StringBuffer theStr = new StringBuffer("Jeff");
```

```
    }                                                    d.doSomething(theStr);
}                                                        System.out.println(theStr);
                                                     }
                                                 }
```

Output in this case would be the string “ffej”.
If this is rather obvious… fine but you should be aware of the possibilities.

4.3 Danger - Reference Semantics - Shared objects

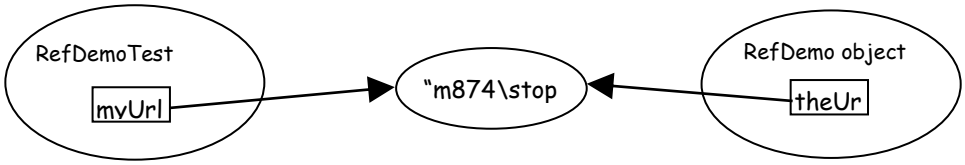
Can you spot the two major ‘weaknesses’ in the RefDemo1 class design.

```
public class RefDemoTest {
    public static void main(String args[]) {
        StringBuffer myUrl, mySbPtr;
        myUrl = new StringBuffer("m874\\stop");
        RefDemo rd = new RefDemo(myUrl,21);
        rd.showUrl();
        myUrl = myUrl.append("\\tmas");
        rd.showUrl();
    }
```

```
output is…
theUrl = m874\stop
theUrl = m874\stop\tmas
```

```
public class RefDemo {
    private StringBuffer theUrl;
    private int theKey;

    public RefDemo(StringBuffer aUrl, int aKey) {
        theUrl = aUrl;
        theKey = aKey;
    }

    public StringBuffer getUrl() {
        return theUrl;
    }

    public int getKey() {
        return theKey;
    }

    public void showUrl() {
        System.out.println("theUrl = " + theUrl);
    }
}
```

The first problem lies in the constructor method. The assignment theUrl = aUrl produces the situation below,



The client object, in this case the test class object, and the server object, the RefDemo object, both share the same StringBuffer object. Changes effected by the reference myUrl are picked up from with the RefDemo object and hence the state of this object is changed from without.

The way round this problem is to allow the server (container) object to make it’s own copy of the incoming object rather than just copy the reference to the incoming object and hence still reference (ie point to) the ‘original’ copy. This is done by changing the constructor method.


```
public class RefDemo {
    private StringBuffer theUrl;
    private int theKey;

    public RefDemo(StringBuffer aUrl, int aKey) {
        theUrl = new
StringBUffer(aUrl.toString());
        theKey = aKey;
    }

    public StringBuffer getUrl() {
        return theUrl;
    }

    public int getKey() {
        return theKey;
    }

    public void showUrl() {
        System.out.println("theUrl = " + theUrl);
    }
}
```

The revised class
Note : it was necessary to convert the parameter aUrl to a string first because the StringBuffer constructor used requires a string as it's parameter.

Of course the description of 'weakness in the class design' may not be appropriate if indeed the intention was for the two objects to share the same StringBuffer object, but you should be aware of the situations that reference semantics can produce.

The other problem is to do with the getUrl() method. With reference semantics is that it is relatively easy to write code that can violate the encapsulation of an object. By returning a reference to an object's data we effectively tell the world where the data item is stored and thus allow access to the data item outside the object. Anytime we allow references to 'float around' then items pointed to by those references become exposed. Clearly by returning a reference we are telling the 'world' where we have hidden our data.

In the method getUrl() we return a reference to theUrl object so the object receiving the return now has access to our private data item. Our data hiding is now lost. To avoid this loss of encapsulation we can make a copy of the data item and return the reference to this copy, thus preserving the integrity of our private data item… this should fool any would be attempts to break the encapsulation.

```
public StringBuffer getUrl() {
    return new StringBuffer(theUrl.toString());
}
```

Returning theKey variable in the getKey() method is not a problem since this is simply a scalar... not a referenced object.

So yes… it is easy to break the data hiding, but bad method design means that it has been broken from the inside not from the outside. It's also easy to do other silly things in programming... as we all know to our cost.

4.4 Representing Objects Texturally - the toString() method

The method toString() can be used to display a textural representation of the current state of an object. Generally it's purpose is to show the current values of the instance variables - ie the object's state, in some sensible readable (text) format and any other information that may be relevant to the user of the object.

```
public class ToStringDemo {
    private int theNumber;
    private String theName;
    public ToStringDemo(int aNumber, String aString) {
        theNumber = aNumber;
        theName = aString;
    }

    public String toString() {
        String aString;
        aString = "This is a textural representation of the state of the current object\n";
        aString += "It's purpose is to show the current values of the instance variables - ie the object's state, in
some sensible readable format and any other information that may be relevent to the user of the object.\n";
        aString += "theNumber = " + theNumber + "\n";
        aString += "theName   = " + theName;
        return aString;
    }
}

public class Tester {
    public static void main(String args[]) {
        ToStringDemo myDemo = new ToStringDemo(0,"Shirt,blue");
        System.out.println(myDemo.toString());
    }
}
```

4.5 Some Decimal, Currency and Percentage Formatting Tips

A DecimalFormat object is created and becomes responsible for the formatting of the double. We can use the NumberFormat class to handle currency and percentage formatting.

```
import java.text.*;
public class TestNumberFormat {
    public static void main(String args[]) {
        double x = 123.43/9.99;
        DecimalFormat df = new DecimalFormat("#.00"); // format to 2dp
        String formatString;
        formatString = df.format(x);        // we ask the df object to format the double x
        System.out.println("unformatted -> " + x);
        System.out.println("formatted   -> " + formatString);
        x = 135.0/144.3;
        System.out.println(x);
        System.out.println(NumberFormat.getCurrencyInstance().format(x));
        System.out.println(NumberFormat.getPercentInstance().format(x));
    }
}
```

4.6 Comparing Objects for Equality

Since Java uses reference semantics when dealing with objects, you have to be careful when you want to compare two objects for equality. When we ask if two objects of the same class are the same we are usually asking if their corresponding instance variables have the same values. It is worth stating that if you have objects and you want to compare their instance variables for equality then the most likely way of doing this is to include an equals() method in the class definition.

4.7 Equality and Equivalence of Objects

When we say two objects are equal we normally mean that there are two different objects but that they have the same state. That is, their corresponding instance variables are equal.
When we speak of object equivalance we normally mean there is only one object but it might be referenced by two (or more) variables.

4.7.1 Example 1

```
public class Book {
    private String author;
    private String isbn;
    private String title;
    public Book(String anAuthor, String anISBN, String aTitle) {
        this.setAuthor(anAuthor);
        this.setISBN(anISBN);
        this.setTitle(aTitle);
    }
    public void setAuthor(String anAuthor) {
        author = anAuthor;
    }
    public void setISBN(String anISBN) {
        isbn = anISBN;
    }
    public void setTitle(String aTitle) {
        title = aTitle;
    }
}
```

If we now create some Book objects how would you answer the question and decide… are the two book objects the same ?

All the classes in the Java API have equals already defined. However, if you have developed a new class and need to carry out some equality comparison of two objects defined by the class then you will need to define a method equals for that class that overrides the equals() method that all classes inherit from the Object superclass. Normally the code for this method carries out some element-by-element equality comparison of the instance variables.
A starting point is found in the superclass Object. An equals() method is found in this superclass and so it is available to all subclasses through inheritance. It's interface is

```
public boolean equals(Object obj)
```

It is used by the instances of the Book class in the following example,

```
public class BookTest {
    public static void main(String args[]) {
        Book book_A = new Book("Tolkein","09-08-07","The Hobbit");
        Book book_B = book_A;
        Book book_C = new Book("Tolkein","09-08-07","The Hobbit");
        if (book_A.equals(book_B)) System.out.println("yes… book_A = book_B");
        if (book_A.equals(book_C)) System.out.println("yes… book_A = book_C");
    }
}
```

Output from the program was…. yes… book_A = book_B

So book_A is not equal to book_C ?

"book_A and book_B are equivalent"…

That is there is only one Book object but it is referenced by two variables book_A and book_B. So we say the object book_A is equivalent to the object book_B.

The equals() method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any reference values x and y, this method returns true if and only if x and y refer to the same object (x ==y has the value true).

This means that if we write

```
if (book_A == book_B) System.out.println("yes… book_A = book_B");
```

the output will also be… yes… book_A = book_B

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 mean do thay 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 book_A equal book_C… to be yes.

To test if two objects are equal (ie have the same state) each subclass of Object must override and implement its own version of equals().

In object work we often take the meaning of **same** to mean equivalence (ie only one object) and the meaning of **equal** to mean different objects having the same state.

Hence to test if two Book objects are equal we shall have to implement an equals method in the class. The following method was added to the Book class,

```
public boolean equals(Book inBook) {
    return (this.getAuthor().equals(inBook.getAuthor()) &&
            this.getISBN().equals(inBook.getISBN()) &&
            this.getTitle().equals(inBook.getTitle()));
}
```

The output from the BookTest program now is… yes… book_A = book_B
yes… book_A = book_C

4.7.2 Another Example

```
public class A {
    private int theInt;
    private String theString;

    public A(int anInt, String aString) {
        theInt = anInt;
        theString = aString;
    }
}

public class EqualsDemo {
    public static void main(String args[]) {
        A x = new A(1,"jeff");
        A y = new A(1,"jeff");
        A z = x;
        if (x.equals(y)) System.out.println("java is great");
        if (x.equals(z)) System.out.println("but C++ is better");
    }
}
```

- What's the output from this program ?
- What would have to be done to get the 'correct' output if it was state which determined equality of 2 objects?

The equals() method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any reference values x and y, this method returns true if and only if x and y refer to the same object (that is x == y has the value true). Only one object is involved... but two references.

Another way... if p and q are object references, and equals() is inherited from the Object class, p.equals(q) returns true if p and q refer to the same (unique) object and would return false if p referenced one object and q referenced another (different) object, even though the two objects has the same state (ie all instance variables the same).

4.7.3 Yet another equals() Example

Here we compare two objects of a completely different type. We compare a type A object with a type B object. Why does the code compile and does it make sense to attempt to do the comparison this way.

```
public class Test {
    public static void main(String a[]) {
        if (new A().equals(new B())) System.out.println("jeff");
    }
}
class A {}
class B {}
```

4.8 Converting Between ints and char Scalars

Like a number of other languages Java allows a ‘free’ conversion between int and char scalars.

```
public class ADemo {
    private int x, y, z;
    private char c;
    public ADemo() {
        x = '.';
        System.out.println("Unicode value of '.' = " + new Integer('.').toString());
        y = 21;
        c = 'A';
        System.out.println("Unicode value of 'A' = " + new Integer('A').toString());
        z = x + y + c;
        System.out.println(z);
        System.out.println('A' + 'B' + 'C');
        System.out.println(50 + 'A');
        System.out.println('A');
        c = ('A' + 'B') / 2;
        System.out.println(c);
        System.out.println(('A' + 'B' + 'C') / 3);
        System.out.println((char)(('A' + 'B' + 'C') / 3));
    }
    public static void main(String args[]) {
        ADemo testObject = new ADemo();
    }
}
```

Output was…

```
Unicode value of '.' = 46
Unicode value of 'A' = 65
132
198
115
A
A
66
```

4.9 Object Wrappers - Converting between Scalars and Objects

The basic scalar data types such as integers, floats, doubles, chars and booleans are not objects. However there will be many situations where you will need to treat the scalars as objects and hence Java makes provision for this with the Wrapper feature. Wrappers allow, for example, a simple scalar integer data type to be converted into an Integer object so that it can then be treated as an object for processing. It will then be able to understand messages and work (communicate) with other objects.

4.9.1 Converting an int scalar to a String object

int x = 123;

String aString = String.valueOf(x);	Using the valueOf() method from the String class
String bString = new Integer(x).toString();	Use an Integer wrapper, then convert it to a String
String cString = "" + 123;	Whenever the + operator appears in an expression where at least one operand is a string the other operands are converted to strings also. So the concatenation of the empty string and the int 123 produces the String "123"

4.9.2 Converting an int scalar to an Integer object

```
int x = 987;
Integer intObj = new Integer(x);
```

4.9.3 Converting a String Object to an int scalar

```
int x = Integer.parseInt(aString);
```

4.9.4 Converting a char scalar to a Character Object

```
char c = 'a';
Character charObj = new Character(c);
```

4.9.5 Checking if a char value is a letter or a digit

```
c = 'a';
if (Character.isLetter(c)) System.out.println("yes a letter");
c = '9';
if (Character.isDigit(c)) System.out.println("yes a digit");
```

4.9.6 Converting Case

```
System.out.println(Character.toUpperCase(c));
```

4.10 State Variables and Working variables

Recall from previous work that the state of an object is the sum total of all it's instance variables. This is why instance variables are sometimes called state variables. You should remember this when you are designing a class. Consider the following,

```
public class AirCraft {
    private Seat[] planeSeats ;        // an array of seat objects representing the state of the AirCraft object
    private int LoopCounter;       // used as Loop control variable
    private int RowIndx;           // used to count seat row 1-n
    private int SeatIndx;          // seat index position A-F
    private int TotalCount;        // Integer that counts the number of finds used by helper method
    private int FoundCounter;      // The counter that steps through the new array used by helper  method

    public AirCraft {
        etc...
```

You should distinguish between state variables (instance variables) and 'working variables', which are usually local variables declared within the method where they are used. Of those declared above only the first one is really a state (instance) variable. The others should be removed and declared where they are required in the various methods.

4.11 How Can I Read Data From the Keyboard ?

As usual with OOP we create an object to do the reading for us. In this approach I have defined a KeyBoardReader class and then in the test class I have created a KeyBoardReader object and used it to get the input from the keyboard. I have given two versions of the KeyBoardReader class; it doesn't matter which you use. There are some features in the classes that you may not have yet studied. If you want to use the class then just use the class to get keyboard input, and worry about the details later. There are of course other ways of getting keyboard input...

Version 1

```
import java.io.*;
public class KeyBoardReader {
  private String thePrompt;

  public KeyBoardReader(char aPrompt) {
    thePrompt = aPrompt + " ";
  }

  public String getInput() {
     byte[] buffer = new byte[80];
    System.out.print(thePrompt);
     try {
       System.in.read(buffer);
     }
    catch (IOException e) {}
       return new String(buffer);
  }
}
```

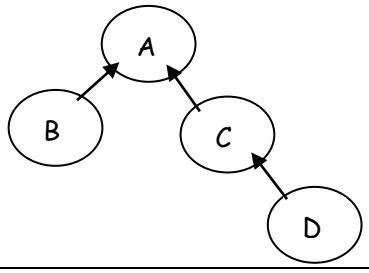
Version 2

```
import java.io.*;
public class KeyBoardReader {
  private InputStreamReader isr;
  private BufferedReader br;
  private String thePrompt;

  public KeyBoardReader(char aPrompt) {
     isr = new InputStreamReader(System.in);
     br = new BufferedReader(isr);
    thePrompt = aPrompt + " ";
  }

  public String getInput() {
     String retVal = null;
     System.out.print(thePrompt);
     try {
        retVal = br.readLine();
      }
      catch (IOException e) {}
      return retVal;
  }
}
```

```
public class TestKeyBoardReader {
    public static void main(String args[]) {
        KeyBoardReader aKBReader = new KeyBoardReader('?');
        String input = aKBReader.getInput();
        System.out.println("From keyboard : " + input);
        while (1==1);
  }
}
```

4.12 More on Polymorphism, Substitutabilty and Late Binding

Substitutabilty means that an object of one type is legitimately substitutable for an object of another type. This is allowed in Java when objects are related through inheritance. An object of the child class may be substituted for an object of the parent class. This is because the subclass object has everything that the superclass object would have, plus some possible extra features as well. So using the subclass object as if it were a superclass object won't cause any problems. This property of Java (and other object-oriented languages) is sometimes known as *subtype polymorphism*.



The getMessage() method is an example of overriding; a polymorhpic message; same message signature, but means different things to each object

```
public abstract class A {
  public abstract String getMessage();
}
```

```
public class B extends A {
  public String getMessage() {
        return "this is a message from B...";
  }
}
```

```
public class C extends A {
  public String getMessage() {
        return "this is a message from C...";
  }
}
```

```
public class D extends C {
  public String getMessage() {
        return "this is a message from D...";
  }
}
```

```
import java.util.*;
public class Test {
    public static void main(String args[]) {
        Vector theList = new Vector();
        B b = new B();    theList.addElement(b);
        C c = new C();    theList.addElement(c);
        D d = new D();    theList.addElement(d);

        A nextObject;
        for (int i = 0; i < theList.size(); ++i) {
           nextObject = (A)theList.elementAt(i);
          System.out.println(nextObject.getMessage());
        }
    }
}
```

The code nextObject.getMessage() binds the getMessage() method to the object. Since nextObject can be one of 3 different types the binding does not take place until the type of the object is known... and this can only be determined when the code is running. Hence the term 'late binding'.

This late binding is confirmed from the output,

this is a message from B...
this is a message from C...
this is a message from D...

When the getMessage() message is sent to nextObject, the type is sorted out and the appropriate version of the message is used.

You might also like to consider the following. Note the re-declaration form of the 3 objects.

```
import java.util.*;
public class Test {
```

```
    public static void main(String args[]) {
        Vector theList = new Vector();
        A b = new B();     theList.addElement(b);
        A c = new C();     theList.addElement(c);
        A d = new D();     theList.addElement(d);

        A nextObject;
        for (int i = 0; i < theList.size(); ++i) {
            nextObject = (A)theList.elementAt(i);
           System.out.println(nextObject.getMessage());
        }
    }
}
```

The variables b,c and d are all declared to refer to an object of type A but following the object creation (new ...) actually refer to objects of type B, C and D respectively. This is allowed by substitutabilty. The actual objects created are not A objects, but are in turn, a B object, a C object and a D object.

The reference type and the object type don't always match. The rule is

- if the reference type is a class, the object it refers to must be either that same type, or a subclass of that type.
- if the reference type is an interface, the object it refers to must be an object from a class which implements the interface, either directly or through inheritance.

Output, and the reasons for it, are just the same as before.