CHAPTER: 08

Implementing classes

Objectives:

By the end of this chapter you should be able to:

- *design classes using the notation of the **Unified Modeling Language (UML)**;*
- *write the Java code for a specified class;*
- *explain the difference between* `public` *and* `private` *access to attributes and methods;*
- *explain the meaning of the term **encapsulation**;*
- *explain the use of the* `static` *keyword;*
- *pass objects as parameters;*•

8.1 Introduction

This chapter is arguably the most important so far, because it is here that you are going to learn how to develop the classes that you need for your programs. You are already familiar with the concept of a class, and the idea that we can create objects that belong to a class; in the last chapter you saw how to create and use objects, and you saw how we could use the methods of a class without knowing anything about how they work.

In this chapter you will look inside the classes you have studied to see how they are constructed, and how you can write classes of your own. We start with the `Rectangle` class.

8.2 Designing classes in UML notation

In the last chapter you saw that a class consists of:

- a set of **attributes** (the data);
- a set of **methods** that can access or change those attributes.

When we design a class we must, of course, consider what data the class needs to hold, and what methods are needed to access that data. The `Rectangle` class that we develop here will need to hold two items of data – the length and the height of the rectangle; these will have to be real numbers, so `double` would be the appropriate type for each of these two attributes. You have already seen the

methods that we provided for this class in table 7.1 in the previous chapter.

When we design classes, it is very useful to start off by using a diagrammatic notation. The usual way this is done is by making use of the notation of the **Unified Modeling Language (UML)**[1]. In this notation, a class is represented by a box divided into three sections. The first section provides the name of the class, the second section lists the attributes, and the third section lists the methods. The UML class diagram for the `Rectangle` class is shown in figure 8.1.

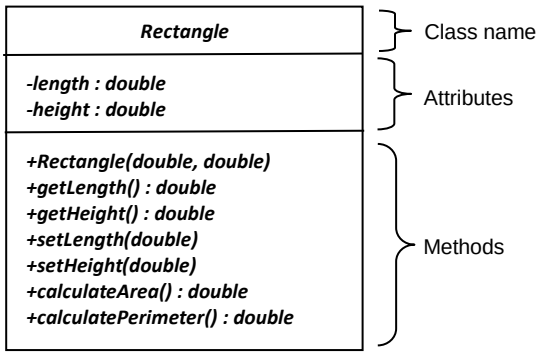

Fig 8.1. The design of the *Rectangle* class

You can see that the UML notation requires us to indicate the names of the attributes along with their types, separated by a colon.

In the last chapter we introduced you to the concept of *encapsulation* or information-hiding. This is the technique of making attributes accessible only to the methods of the same class, and it is this feature of object-oriented languages that has contributed to object-orientation becoming the standard way of programming in today's world. By restricting access in this way, programmers can keep the data in their classes "sealed off" from other classes, because they are the ones in control of how it is actually accessed.

The way our `Rectangle` class has been set up means that you cannot directly use the `length` and `height` attributes in another program. If you want to find out the area of the rectangle in, say, the **`main`** method of another program then you can't do this by a accessing the `length` and `height` data directly, because access to these attributes is denied.

Instead we would, as you know, call the `calculateArea` method of the `Rectangle` object. We design our classes like this because doing so means that no-one can inadvertently *change* the values of `length` and `height` - our data is kept secure. If access to these attributes were not restricted in this way, then the `length` and `height` data could inadvertently be changed. Instead we limit access of the `Rectangle` class to its methods. This is illustrated in figure 8.2 below:

[1] Simon Bennett et al., *Schaum's Outline UML (3rd edn)*, McGraw-Hill 2006.

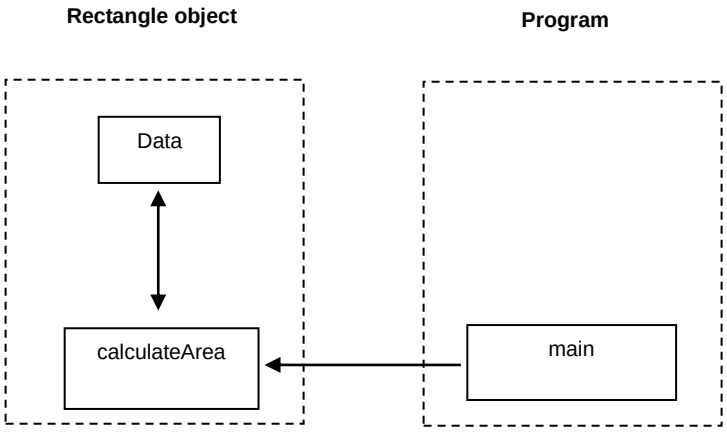

Fig 8.2 Encapsulation requires data be kept hidden inside an object

The plus and minus signs that you can see in the UML diagram in figure 8.1 are all to do with this idea of encapsulation; a minus sign means that the attribute or method is **private** - that is, it is accessible *only to methods of the same class*. A plus sign means that it is **public** - it is accessible to methods of other classes. Normally we make the attributes private, and the methods public, in this way achieving encapsulation. You will see how it is done in a Java class in the next section.

Now let's consider the notation for the methods. You can see from the diagram that the parameter types are given in brackets - for example:

+setLength(double)

The return types are placed after the brackets, preceded by a colon - for example:

+getLength() : double

Where there is no return type, nothing appears after the brackets, as in the `setlength` and `getLength` methods.

The first method, `Rectangle`, is the constructor. As we know the constructor always has the same name as the class, and in this case it requires two parameters of type **`double`**:

+Rectangle(double, double)

You should note that a constructor *never has a return type*. In fact you will see later that in Java we don't even put the word **`void`** in front of a constructor; if we did the compiler would think it was a regular method.

As you saw in the previous chapter, we have provided our `Rectangle` class with methods for reading and writing to the attributes – and it is conventional to begin the name of such methods with `get-` and `set-` respectively. However, it is not always the case that we choose to supply methods such as `setLength` and `setHeight`, which allow us to *change* the attributes. Sometimes we set up our class so that the only way that we can assign values to the attributes is via the constructor. This would mean that the values of the `length` and `height` could be set only at the time a new `Rectangle` object

was created, and could not be changed after that. Whether or not you want to provide a means of writing to individual attributes depends on the nature of the system you are developing and should be discussed with potential users. However, we believe that it is a good policy to provide write access to only those attributes that clearly require to be changed during the object's lifetime, and we have taken this approach throughout this book. In this case we have included "set" methods for `length` and `height` because we are going to need them in chapter 10.

8.3 Implementing classes in Java

8.3.1 The *Rectangle* class

Now that we have the basic design of the `Rectangle` class we can go ahead and write the Java code for it. We present the code here - when you have had a look at it we will discuss it.

The *Rectangle* class

```
public class Rectangle
{
   // the attributes
   private double length;
   private double height;

   // the methods

   // the constructor
   public Rectangle(double lengthIn, double heightIn)
   {
      length = lengthIn;
      height = heightIn;
   }

   // this method allows us to read the length attribute
   public double getLength()
   {
      return length;
   }

   // this method allows us to read the height attribute
   public double getHeight()
   {
      return height;
   }

   // this method allows us to write to the length attribute
   public void setLength(double lengthIn)
   {
      length = lengthIn;
   }

   // this method allows us to write to the height attribute
   public void setHeight(double heightIn)
   {
      height = heightIn;
   }

   // this method returns the area of the rectangle
   public double calculateArea()
   {
      return length * height;
   }
```

```
   // this method returns the perimeter of the rectangle
   public double calculatePerimeter()
   {
      return 2 * (length + height);
    }
}
```

Let's take a closer look at this. The first line declares the `Rectangle` class:

```
public class Rectangle
```

Next comes the attributes. A `Rectangle` object will need attributes to hold values for the length and the height of the rectangle, and these will be of type **`double`**. The declaration of the attributes in the `Rectangle` class took the following form in our UML diagram:

-length : double
-height : double

In Java this is implemented as:

```
private double length;
private double height;
```

As you can see, attributes are declared like any other variables, except that they are declared *outside* of any method, and they also have an additional word in front of them - the word **`private`**, corresponding to the minus sign in the UML notation. In Java, this keyword is used to restrict the scope of the attributes to methods of this class only, as we described above.

You should note that the attributes of a class are accessible to *all* the methods of the class – unlike *local* variables, which are accessible only to the methods in which they are declared.

Figure 8.1 makes it clear which methods we need to define within our `Rectangle` class. First comes the constructor. You should recall that it has the same name as the class, and, unlike any other method, it has no return type - not even **`void`**! It looks like this:

```
public Rectangle(double lengthIn, double heightIn)
{
    length = lengthIn;
    height = heightIn;
}
```

The first thing to notice is that this method is declared as **`public`**. Unlike the attributes, we want our methods to be accessible from outside so that they can be called by methods of other classes.

In our class we are defining the constructor so that when a new `Rectangle` object is created (with the keyword **`new`**) then not only do we get some space reserved in memory, but we also get some other stuff

occurring; in this case two assignment statements are executed. The first assigns the value of the parameter `lengthIn` to the `length` attribute, and the second assigns the value of the parameter `heightIn` to the `height` attribute. Remember, the attributes are visible to all the methods of the class.

When we define a constructor like this in a class it is termed a *user-defined*[2] constructor. If we don't define our own constructor, then one is automatically provided for us - this is referred to as the **default** constructor. The default constructor takes no parameters and when it is used to create an object - for example in a line like this:

```
Rectangle myRectangle = new Rectangle();
```

then all that happens is that memory is reserved for the new object - no other processing takes place. Any attributes will be given initial values according to the rules that we give you later in section 8.5.

One more thing about constructors: once we have defined our own constructors, this default constructor is no longer automatically available. If we want it to be available then we have to re-define it explicitly. In the `Rectangle` case we would define it as:

```
public Rectangle()
{
}
```

You can see that just like regular methods, constructers can be overloaded, and we can define several constructors in one class. When we create an object it will be clear from the parameter list which constructor we are referring to.

Now let's take a look at the definition of the next method, `getLength`. The purpose of this method is simply to send back the value of the `length` attribute. In the UML diagram it was declared as:

+getLength() : double

In Java this becomes:

```
public double getLength()
{
    return length;
}
```

Once again you can see that the method has been declared as **`public`** (indicated by the plus sign in UML), enabling it to be accessed by methods of other classes.

The next method, `getHeight`, behaves in the same way in respect of the `height` attribute.

Next comes the `setLength` method:

[2] Here the word *user* is referring to the person *writing* the program, not the person using it!

+setLength(double)

We implement this as:

```
public void setLength(double lengthIn)
{
    length = lengthIn;
}
```

This method does not return a value, so its return type is **`void`**. However, it does require a parameter of type **`double`** that it will assign to the `length` attribute. The body of the method consists of a single line which assigns the value of `lengthIn` to the `length` attribute.

The next method, `setHeight`, behaves in the same way in respect of the `height` attribute.

After this comes the `calculateArea` method:

+calculateArea() : double

We implement this as:

```
public double calculateArea()
{
    return length * height;
}
```

Once again there are no formal parameters, as this method does not need any data in order to do its job; it returns a **`double`**. The actual code is just one line, namely the statement that returns the area of the rectangle, calculated by multiplying the value of the `length` attribute by the value of the `height` attribute.

The `calculatePerimeter` method is similar and thus the definition of the `Rectangle` class is now complete.

One important thing to note here. Unlike some of the methods we developed in chapter 5, the methods that we have defined here deal only with the basic *functionality* of the class - they do not include any routines that deal with input or output. That is because the methods of chapter 5 were only being used by the class in which they were written - but now our methods will be used by other classes that we cannot as yet predict. So when developing a class we should always strive to restrict our methods to the essential functions that define the class (in this case, for example, calculating the area and perimeter of the rectangle), and to exclude anything that is concerned with the input or output functions of a program. If we do this, then our class can be used in any sort of application, regardless of whether is it a simple console application like the ones we have developed so far, or a complex graphical application like the ones you will come across later in this book.

8.3.2 The *BankAccount* class

The UML class diagram for the `BankAccount` class, which we used in the previous chapter, is shown in figure 8.3.

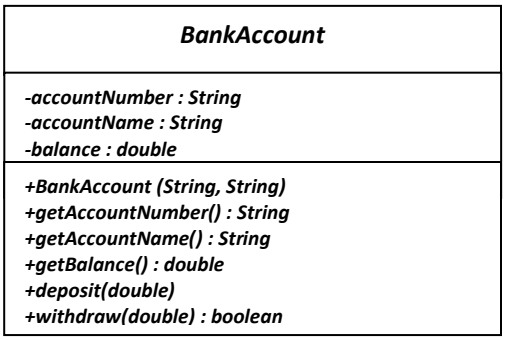

Fig 8.3. The design of *BankAccount* class

You will notice here that *accountNumber* and *accountName* are declared as *Strings*; it is perfectly possible for the attributes of one class to be objects of another class.

We can now inspect the code for this class:

The *BankAccount* class

```
public class BankAccount
{
   // the attributes
   private String accountNumber;
   private String accountName;
   private double balance;

   // the methods

   // the constructor
   public BankAccount(String numberIn, String nameIn)
   {
      accountNumber = numberIn;
      accountName = nameIn;
      balance = 0;
   }

   // methods to read the attributes
   public String getAccountName()
   {
       return accountName;
   }

   public String getAccountNumber()
   {
       return accountNumber;
```

```
    }

    public double getBalance()
    {
        return balance;
    }

    // methods to deposit and withdraw money
    public void deposit(double amountIn)
    {
        balance = balance + amountIn;
    }
    public boolean withdraw(double amountIn)
    {
        if(amountIn > balance)
        {
            return false; // no withdrawal was made
        }
        else
        {
            balance = balance - amountIn;
            return true; // money was withdrawn successfully
        }
    }
}
```

Now that we are getting the idea of how to define a class in Java, we do not need to go into so much detail in our analysis and explanation.

The first three lines declare the attributes of the class, and are as we would expect:

```
private String accountNumber;
private String accountName;
private double balance;
```

Now the constructor:

```
public BankAccount(String numberIn, String nameIn)
{
    accountNumber = numberIn;
    accountName = nameIn;
    balance = 0;
}
```

You can see that when a new object of the `BankAccount` class is created, the `accountName` and `accountNumber` will be assigned the values of the parameters passed to the method. In this case, the `balance` will be assigned the value zero; this makes sense because when someone opens a new account there is a zero balance until a deposit is made[3].

[3] You would be right in thinking that the balance attribute would automatically be assigned a value of zero if we did not specifically do that here. However it is good practice always to ensure that variables are initialized with the values that we require - particularly because in many other programming languages attributes are not initialized as they are in Java.

The next three methods, `getAccountNumber`, `getAccountName` and `getBalance`, are all set up so that the values of the corresponding attributes (which of course have been declared as **`private`**) can be read.

After these we have the `deposit` method:

```
public void deposit(double amountIn)
{
   balance = balance + amountIn;
}
```

This method does not return a value; it is therefore declared to be of type **`void`**. It does however require that a value is sent in (the amount to be deposited), and therefore has one parameter – of type **`double`** – in the brackets. As you would expect with this method, the action consists of adding the deposit to the `balance` attribute of the `BankAccount` object.

Now the `withdraw` method:

```
public boolean withdraw(double amountIn)
{
       if(amountIn > balance)
       {
          return false; // no withdrawal was made
       }
       else
       {
          balance = balance - amountIn;
          return true; // money was withdrawn successfully
       }
 }
```

The amount is subtracted only if there are sufficient funds - in other words if the amount to be withdrawn is no bigger than the balance. If this is not the case then a value of **`false`** is returned and the method terminates. Otherwise the amount is subtracted from the balance and a value of **`true`** is returned. The return type of the method therefore is **`boolean`**.

8.4 The *static* keyword

You have already seen the keyword **`static`** in front of the names of methods in some Java classes. A word such as this (as well as the words **`public`** and **`private`**) is called a **modifier**. A modifier determines the particular way a class, attribute or method is accessed.

Let's explore what this **`static`** modifier does. Consider the `BankAccount` class that we discussed in the previous section. Say we wanted to have an additional method which added interest, at the current rate, to the customer's balance. It would be useful to have an attribute called `interestRate` to hold the value of the current rate of interest. But of course the interest rate is the same for any customer – and if it changes, we want it to change for every customer in the bank; in other words for every object of the class. We can achieve this by declaring the variable as **`static`**. An attribute declared as **`static`** is a

class attribute; any changes that are made to it are made to all the objects in the class.

It would make sense if there were a way to access this attribute without reference to a specific object; and so there is! All we have to do is to declare methods such as `setInterestRate` and `getInterestRate` as **`static`**. This makes a method into a *class* method; it does not refer to any specific object. As you will see in program 8.1 we can call a class method by using the class name instead of the object name.

We have rewritten our `BankAccount` class, and called it `BankAccount2`. We have included three new methods as well as the new **`static`** attribute `interestRate`. The first two of these – `setInterestRate` and `getInterestRate` – are the methods that allow us to read and write to our new attribute. These have been declared as **`static`**. The third – `addInterest` – is the method that adds the interest to the customer's balance. As can be seen in figure 8.4, the UML notation is to underline static attributes and methods.

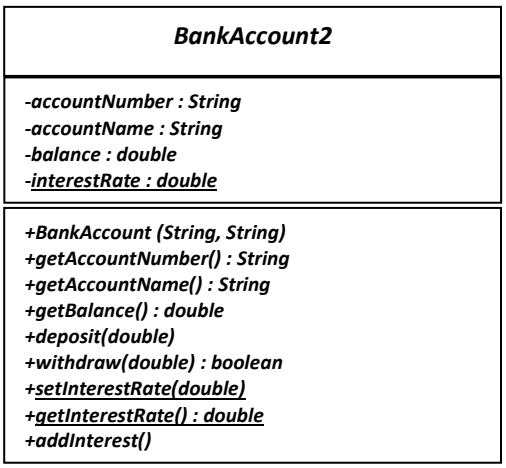

Fig 8.4. The design of the *BankAccount2* class

Here is the code for the class. The new items have been emboldened.

The modified *BankAccount* class

```
public class BankAccount2
{
   private String accountNumber;
   private String accountName;
   private double balance;
   private static double interestRate;

   public BankAccount2(String numberIn, String nameIn)
   {
      accountNumber = numberIn;
      accountName = nameIn;
      balance = 0;
   }

   public String getAccountName()
   {
      return accountName;
```

```
   }

   public String getAccountNumber()
   {
      return accountNumber;
   }

   public double getBalance()
   {
      return balance;
   }

   public void deposit(double amountIn)
   {
      balance = balance + amountIn;
   }

    public boolean withdraw(double amountIn)
    {
        if(amountIn > balance)
        {
            return false;
        }
        else
        {
            balance = balance - amountIn;
            return true;
        }
    }

   public static void setInterestRate(double rateIn)
   {
      interestRate = rateIn;
   }

   public static double getInterestRate()
   {
      return interestRate;
   }

   public void addInterest()
   {
      balance = balance + (balance * interestRate)/100;
   }
}
```

Program 8.1 uses this modified version of the `BankAccount` class.

Program 8.1

```
public class BankAccountTester2
{
  public static void main(String[] args)
  {
     // create a bank account
     BankAccount2 account1 = new BankAccount2("99786754","Gayle Forcewind");
     // create another bank account
     BankAccount2 account2 = new BankAccount2("99887776","Stan Dandy-Liver");
     // make a deposit into the first account
     account1.deposit(1000);
     // make a deposit into the second account
     account2.deposit(2000);
     // set the interest rate
     BankAccount2.setInterestRate(10);
```

```
    // add interest to accounts
    account1.addInterest();
    account2.addInterest();
    // display the account details
    System.out.println("Account number: " + account1.getAccountNumber());
    System.out.println("Account name: " + account1.getAccountName());
    System.out.println("Interest Rate " + BankAccount2.getInterestRate());
    System.out.println("Current balance: " + account1.getBalance());
    System.out.println(); // blank line
    System.out.println("Account number: " + account2.getAccountNumber());
    System.out.println("Account name: " + account2.getAccountName());
    System.out.println("Interest Rate " + BankAccount2.getInterestRate());
    System.out.println("Current balance: " + account2.getBalance());
  }
}
```

Take a closer look at the first four lines of the `main` method of program 8.1. We have created two new bank accounts which we have called `account1` and `account2`, and have assigned account numbers and names to them at the time they were created (via the constructor). We have then deposited amounts of 1000 and 2000 respectively into each of these accounts.

Now look at the next line:

```
BankAccount2.setInterestRate(10);
```

This line sets the interest rate to 10. Because `setInterestRate` has been declared as a **`static`** method, we have been able to call it by using the class name `BankAccount2`. Because `interestRate` has been declared as a **`static`** attribute this change is effective for any object of the class.

Therefore, when we add interest to each account as we do with the next two lines

```
account1.addInterest();
account2.addInterest();
```

we should expect it to be calculated with an interest rate of 10, giving us new balances of 1100 and 2200 respectively.

This is exactly what we get, as can be seen from the output below:

```
Account number: 99786754
Account name: Gayle Forcewind
Interest Rate 10.0
Current balance: 1100.0

Account number: 99887776
Account name: Stan Dandy-Liver
Interest Rate 10.0
```

```
Current balance: 2200.0
```

Class methods can be very useful indeed and we shall see further examples of them in this chapter. Of course, we have always declared our `main` method, and other methods within the same class as the `main` method, as **`static`** – because these methods belong to the class and not to a specific object.

8.5 Initializing attributes

Looking back at the `BankAccount2` class in the previous section, some of you might have been asking yourselves what would happen if we called the `getInterestRate` method before the interest rate had been set using the `setInterestRate` method. In fact, the answer is that a value of zero would be returned. This is because, while Java does not give an initial value to *local* variables (which is why you get a compiler error if you try to use an uninitialized variable), Java always initializes attributes. Numerical attributes such as **`int`** and **`double`** are initialized to zero; **`boolean`** attributes are initialized to **`false`** and objects are initialized to **`null`**. Character attributes are given an initial Unicode value of zero.

Despite the above, it is nonetheless good programming practice always to give an initial value to your attributes, rather than leave it to the compiler. One very good reason for this is that you cannot assume that every programming language initializes variables in the same way – if you were using C++, for example, the initial value of any variable is completely a matter of chance – and you won't get a compiler error to warn you! In the `BankAccount2` class, it would have done no harm at all to have initialized the `interestRate` variable when it was declared:

```
private static double interestRate = 0;
```

In fact, one technique you could use is to give the `interestRate` attribute some special initial value (such as a negative value) to indicate to the user of this class that the interest rate had not been set. You will see another example where this technique can be used in question 2 of the programming exercises.

8.6 Passing objects as parameters

In chapter 5 it was made clear that when a variable is passed to a method it is simply the *value* of that variable that is passed – and that therefore a method cannot change the value of the original variable. In chapter 6 you found out that in the case of an array it is the value of the memory location (a *reference*) that is passed and consequently the value of the original array elements can be changed by the called method.

What about objects? Let's write a little program (program 8.2) to test this out.

Program 8.2

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

```
  {
   // create new bank account
   BankAccount testAccount = new BankAccount("1", "Ann T Dote");
   test(testAccount); // send the account to the test method
   System.out.println("Account Number: " + testAccount.getAccountNumber());
   System.out.println("Account Name: " + testAccount.getAccountName());
   System.out.println("Balance: " + testAccount.getBalance());
  }

  // a method that makes a deposit in the bank account
  static void test(BankAccount accountIn)
  {
   accountIn.deposit(2500);
  }
}
```

The output from this program is as follows:

```
Account Number: 1
Account Name: Ann T Dote
Balance: 2500.0
```

You can see that the deposit has successfully been made - in other words the attribute of the object has actually been changed.. This is because what was sent to the method was, of course, a *reference* to the original `BankAccount` object, `testAccount`. Thus `accountIn` is a *copy* of the `testAccount` reference and so points to the original object and invokes that object's methods. So the following line of code:

```
accountIn.deposit(2500);
```

calls the `deposit` method of the original `BankAccount` object.

You might think this is a very good thing, and will make life easier for you as a programmer. However, you need a word of caution here. It is very easy inadvertently to allow a method to change an object's attributes, so you need to take care – more about this in the second semester.

8.7 The benefits of object-oriented programming

In this chapter and the previous one you have seen how to create classes and use them as data types in your programs. You have seen how the process of building classes enables us to hide data within a class. Programming languages based on classes and objects - in other words object-oriented languages - have brought a number of benefits, and are now the standard. Below we have summarized some of the benefits that this has brought us.

- As we have demonstrated, the ability to encapsulate data within a class has enabled us to build far more secure systems.

- The object-oriented approach makes it far easier for us to *re-use* classes again and again. Having defined a `BankAccount` class or a `Student` class for example, we can use them in many different programs without having to write a new class each time. In the next chapter you will also see how it is possible to refine existing classes to meet additional needs by the technique known as **inheritance**. If systems can be assembled from re-usable objects, this leads to far higher productivity.

- With the object-oriented approach it is possible to define and use classes which are not yet complete. They can then be extended without upsetting the operation of other classes. This greatly improves the testing process. We can easily build prototypes without having to build a whole system before testing it and letting the user of the system see it.

- The object-oriented approach makes it far easier to make changes to systems once they have been completed. Whole classes can be replaced, or new classes can easily be added.

- The object-oriented way of doing things is a far more "natural" approach. We base our programs on objects that exist in the real world - students, bank accounts, customers and so on.

- The modular nature of object-oriented programming improves the whole development process. The modular approach means that the old methodologies whereby systems were first analysed, then designed, and then implemented and tested were able give way to new methods whereby these processed were far more integrated and systems were developed far more rapidly.