Methods

We have already come across the methods **Main**, **WriteLine** and **ReadLine**. **Main** is the method we write which is where our program starts. **WriteLine** and **ReadLine** were provided by the creators of C# to give us a way of displaying text and reading information from the user.

This is what methods are all about. Your programs will contain methods that you create to solve parts of the problem and they will also use methods that have been provided by other people. In this section we are going to consider why methods are useful and how you can create your own.

Methods So Far

In the glazing program above we spend a lot of time checking the values of inputs and making sure that they are in certain ranges. We have exactly the same piece of code to check widths and heights. If we added a third thing to read, for example frame thickness, we would have to copy the code a third time. This is not very efficient; it makes the program bigger and harder to write. What we would like to do is write the checking code once and then use it at each point in the program. To do this you need to define a method to do the work for you.

Method and Laziness

We have already established that a good programmer is creatively lazy. One of the tenets of this is that a programmer will try to do a given job once and once only.

Up until now all our programs have been in a single method. The method is the block of code which follows the main part in our program. However, C# lets us create other methods which are used when our program runs. Methods give us two new weapons:

1. We can use methods to let us re-use a piece of code which we have written.
2. We can also use methods to break down a large task into a number of smaller ones.

We will need both of these when we start to write larger programs. Again, as with lots of features of the C# language, methods don't actually make things possible, but they do help with the organisation of our programs.

Essentially you take a block of code and give it a name. Then you can refer to this block of code to do something for you. As a silly example:

```
using System ;
class MethodDemo {
   static void doit () {
       Console.WriteLine ("Hello");
   }
   public static void Main () {
       doit();
       doit();
   }
}
```

In the main method I make two calls of **doit**. Each time I call the method the code in the block which is the body of the method is executed. In this case it contains a single statement which prints **"Hello"** on the console. The result of running the above program would be:

```
Hello
Hello
```

So, we can use methods to save us from writing the same code twice. We simply put the code inside a method body and then call it when we need it.

Parameters

At this point methods are useful because they let us use the same block of statements at many points in the program. However, they become more useful if we allow them to have parameters.

A parameter is a means of passing a value into a method call. The method is given the data to work on. As an example, consider the code below:

```
using System ;
class MethodDemo {
   static void silly ( int i) {
       Console.WriteLine ( "i is : " + i ) ;
   }
   public static void Main () {
       silly ( 101 ) ;
       silly ( 500 ) ;
   }
}
```

The method **`silly`** has a single integer parameter. Within the block of code which is the body of this method we can use the parameter **`i`** as if it was an integer variable. When the method starts the value supplied for the parameter is copied into it. This means that when the program runs we get output like this:

```
i is : 101
i is : 500
```

Return values

A method can also return a value:

```
using System ;
class ReturnDemo {

   static int sillyReturnPlus ( int i) {
       i = i + 1;
       Console.WriteLine ( "i is : " + i ) ;
       return i;
   }

   public static void Main () {
       int res;
       res = sillyReturnPlus (5);
       Console.WriteLine ( "res is : " + res ) ;
   }
}
```

The method **`sillyReturnPlus`** takes the value of the parameter and returns it plus one.

A Useful Method

If we put these two abilities together we can start to write genuinely useful methods:

```
class MethodDemo {

   static double readValue (
       string prompt, // prompt for the user
       double low,    // lowest allowed value
       double high    // highest allowed value
       ) {
       double result = 0;
       do {
           Console.WriteLine (prompt +
              "between " + low +
               " and " + high );
           string resultString = Console.ReadLine ();
           result = double.Parse(resultString);
       } while ( (result < low) || (result > high) );
       return result ;
   }

   public static void Main () {
       double age = readValue (
           "Enter the age : ", 0, 100.0 ) ;
       Console.WriteLine ( "Age is : " + age ) ;
   }
}
```

The **readValue** method is told the prompt to use and the lowest and the highest allowed values. It can then be used to read values and make sure that they are in range. We can use this method to read the ages of people as in the example above. However, we can also use exactly the same method to read in the width and height of our windows:

```
double windowWidth = readValue (
   "Enter width of window: ", MIN_WIDTH, MAX_WIDTH) ;
```

Programmer's Point:Design with methods

Methods are a very useful part of the programmer's toolkit. They form an important part of the development process. Once you have worked out what the customer wants and gathered your metadata you can start thinking about how you are going to break the program down into methods. Often you find that as you write the code you are repeating a particular action. If you do this you should consider taking that action and moving it into a method. There are two reasons why this is a good idea:

1. It saves you writing the same code twice.
2. If a fault is found in the code you only have to fix it in one place.

Moving code around and creating methods is called *refactoring*. This will be an important part of the Software Engineering we do later.

Method Limitations

The method is very good for getting work done, but it is a bit limited because of the way that it works. For example, if I want to write a method which reads in the name and age of a person I have a problem. From what we have seen of methods, they can only return one value. So I could write a method which returns the name of a person, or write on which returns an age. But not both values at the same time. This limitation is because, unless you specify otherwise, only the **value** of a parameter is passed into a call to a method.

Parameter Passing By Value

So, what do I mean by "passing parameters by value". Consider:

```
static void addOneToParam ( int i) {
    i = i + 1;
    Console.WriteLine ( "i is : " + i ) ;
}
```

The method **addOneToParam** adds one to the parameter, prints the result out and then returns:

```
int test = 20 ;
addOneToParam(test);
Console.WriteLine ( "test is : " + test ) ;
```

The piece of C# above calls the method with the variable **test** as the parameter. When it runs it prints out the following:

```
i is : 21
test is : 20
```

It is **very important** that you understand what is happening here. The **value** of **test** is being used in the call of **addOneToParam**. The program works out the result of the expression to be passed into the method call as a parameter. It then passes this value into the call. This means that you can write calls like:

```
test = 20 ;
addOneToParam(test + 99);
```

This would print out:

```
i is : 120
```

Pass by value is very safe, because nothing the method does will affect variables in the code which calls it. However, it is a limitation when we want to create a method which returns more than one value.

Parameter Passing By Reference

It is very important that you understand how references work. If you don't understand these you can't call yourself a proper programmer!

Fortunately C# provides a way that, rather than sending the value of a variable into a method, instead a reference to that variable is supplied instead. Inside the method, rather than using the value of the variable the reference is used to get the actual variable itself. Effectively the thing that is passed into the method is the position or address of the variable in memory, rather than the content of the variable.

So, rather than passing in "20" in our above call the compiler will generate code which passes in "memory location 5023" instead (assuming that the variable test is actually stored at 5023). This memory location is used by the method, instead of the value. In other words:

"If you pass by reference, changes to the parameter change the variable whose reference you passed"

Consider the code:

```
static void addOneToRefParam ( ref int i)
{
    i = i + 1;
    Console.WriteLine ( "i is : " + i ) ;
}
```

Note that the keyword **ref** has been added to the information about the parameter.

```
test = 20 ;
addOneToRefParam(ref test);
Console.WriteLine ( "test is : " + test ) ;
```

The code above makes a call to the new method, and also has the word **ref** in front of the parameter. In this case the output is as follows:

```
i is : 6
test is : 6
```

In this case the method call has made changes to the content of the variable. Note that C# is careful about when a parameter is a reference. You have to put the word **ref** in the method heading and also in the call of the method.

Programmer's Point:Document your side-effects

A change by a method to something around it is called a *side effect* of the method. Generally speaking you have to be careful with these, as someone reading your program has to know that your method has made changes in this way.

Passing Parameters as "out" references

When you pass a parameter as a reference you are giving the method complete control of it. Sometimes you don't want this. Instead you want to just allow the method to change the variable. This is the case when we want to read in the name and age of a user. The original value of the parameters is of no interest to the method. Instead it just wants to deliver results to them. In this case I can replace the **ref** with the keyword **out**:

```
static void readPerson ( out string name, out int age )
{
    name = readString ( "Enter your name : " ) ;
    age = readInt ( "Enter your age : ", 0, 100 ) ;
}
```

The method **readPerson** reads the name and the age of a person. Note that it uses two more methods that I have created, **readString** and **readInt**.

I can call **readPerson** as follows:

```
string name ;
int age ;
readPerson ( out name, out age ) ;
```

Note that I must use the **out** keyword in the call of the method as well.

The **readPerson** method will read the person and deliver the information into the two variables.

Programmer's Point:Languages can help programmers

The out keyword is a nice example of how the design of a programming language can make programs safer and easier to write. It makes sure that a programmer can't use the value of the parameter in the method. It also allows the compiler to make sure that somewhere in the method the output parameters are assigned values. This is very useful. It means that if I mark the parameters as out I **must** have given them a value for the program to compile. This makes it harder for me to get the program wrong, as I am protected against forgetting to do that part of the job.

Method Libraries

The first thing that a good programmer will do when they start writing code is to create a set of libraries which can be used to make their job easier. In the code above I have written a couple of library methods which I can use to read values of different types:

```
static string readString ( string prompt )
{
    string result ;
    do
    {
           Console.Write ( prompt ) ;
           result = Console.ReadLine ();
    } while ( result == "" ) ;
    return result ;
}

static int readInt ( string prompt, int low, int high )
{
    int result ;

    do
    {      string intString = readString (prompt) ;
           result = int.Parse(intString);
    } while ( ( result < low ) || ( result > high ) );

    return result;
}
```

The **readString** method will read text and make sure that the user does not enter empty text. The **readInt** method reads a number within a particular range. Note how I have rather cleverly used my **readString** method in my **readInt** one, so that the user can't enter an empty string when a number is required.

Programmer's Point:Always consider the failure behaviours

Whenever you write a method you should give some thought to the ways that it could fail. If the method involves talking to the user it is possible that the user may wish to abandon the method, or that the user does something that may cause it to fail. You need to consider whether or not the method should deal with the problem itself or pass the error onto the system which tried to use it.

If the method deals with the error itself this may lead to problems because the user may have no way of canceling a command. If the method passes the error on to the code which called it you have to have a method by which an error condition can be delivered to the caller. I often solve this problem by having my methods return a code value. If the return value is 0 this means that the method returned correctly. If the return value is non-zero this means that the method did not work and the value being returned is an error which identifies what went wrong. This adds another dimension to program design, in that you also have to consider how the code that you write can fail, as well as making sure that it does the required job! We are going to discuss error management later

Variables and Scope

We have seen that when we want to store a quantity in our program we can create a variable to hold this information. The C# compiler makes sure that the correctly sized chunk of memory is used to hold the value and it also makes sure that we only ever use that value correctly. The C# compiler also looks after the part of a program within which a variable has an existence. This is called the *scope* of a variable.

Scope and blocks

We have already seen that a block is a number of statements which are enclosed in curly brackets. Any block can contain any number of *local* variables, i.e. variables which are local to that block.

The scope of a local variable is the block within which the variable is declared. As far as the C# language is concerned you can declare a variable at any point in the block, but you **must** declare it before you use it. When the execution of the program moves outside a block any local variables which are declared in the block are automatically discarded. The methods that we have created have often contained local variables; the variable **`result`** in the **`readInt`** method is local to the method block.

Nested Blocks

We have seen that in C# the programmer can create blocks inside blocks. Each of these *nested* blocks can have its own set of local variables:

```
{
    int i ;
    {
            int j ;
    }
}
```

The variable **`j`** has the scope of the inner block. This means that only statements in the inner block which are after the declaration can use this variable. In other words the code:

```
{
    int i ;
    {
            int j ;
    }
    j = 99 ;
}
```

- would cause an error, as the variable **`j`** does not exist at this point in the program.

In order to keep you from confusing yourself by creating two versions of a variable with the same name C# has an additional rule about the variables in the inner blocks:

```
{
    int i ;
    {
            int i ;
    }
}
```

This is not a valid program because C# does not let a variable in an inner block have the same name as one in an outer block. This is because inside the inner block there is the possibility that you may use the "inner" version of **`i`** when you intend to use the outer one. In order to remove this possibility the compiler refuses to allow this. Note that this is in contrast to the situation in other languages, for example C++, where this behaviour is allowed.

If is however perfectly acceptable to reuse a variable name in successive blocks because in this situation there is no way that one variable can be confused with another.

```
{
    int i ;
}
{
    int i ;
    {
            int j ;
    }
}
```

The first incarnation of **i** has been destroyed before the second, so this code is OK.

For loop local variables

A special kind of variable can be used when you create a for loop construction. This allows you to declare a control variable which exists for the duration of the loop itself:

```
for ( int i = 0 ; i < 10 ; i = i + 1 ) {
    Console.WriteLine ( "Hello") ;
}
```

The variable **i** is declared and initialized at the start of the for loop and only exists for the duration of the block itself.

Programmer's Point:Plan your variable use

You should plan your use of variables in your programs. You should decide which variables are only required for use in local blocks and which should be shared over the entire class.

Arrays

We now know how to create programs that can read values in, calculate results and print them. Our programs can also make decisions based on the values supplied by the user and also repeat actions a given number of times.

It turns out that you now know about nearly all the language features that are required to implement every program that has ever been written. Only one thing is missing, and that is the ability to create programs which store large amounts of data. Arrays are one way to do this, and we are going to find out about them next.

Why We Need Arrays

Your fame as a programmer is now beginning to spread far and wide. The next person to come and see you is the chap in charge of the local cricket team. He would like to you write a program for him which allows the analysis of cricket results. What he wants is quite simple; given a list of cricket scores he wants a list of them in ascending order.

"This is easy" you think. Having agreed the specification and the price you sit down that night and start writing the program. The first thing to do is define how the data is to be stored:

```
int score1, score2, score3, score4, score5, score6, score7
    score8, score9, score10, score11 ;
```

Now you can start putting the data into each variable:

```
Console.WriteLine("Enter the scores ");
string score1String = Console.ReadLine();
int score1 = int.Parse(score1String);
string score2String = Console.ReadLine();
int score2 = int.Parse(score2String);
string score3String = Console.ReadLine();
int score3 = int.Parse(score3String);
string score4String = Console.ReadLine();
int score4 = int.Parse(score4String);
string score5String = Console.ReadLine();
int score5 = int.Parse(score5String);
string score6String = Console.ReadLine();
int score6 = int.Parse(score6String);
string score7String = Console.ReadLine();
int score7 = int.Parse(score7String);
string score8String = Console.ReadLine();
int score8 = int.Parse(score8String);
string score9String = Console.ReadLine();
int score9 = int.Parse(score9String);
string score10String = Console.ReadLine();
int score10 = int.Parse(score10String);
string score11String = Console.ReadLine();
int score11 = int.Parse(score11String);
```

All we have to do next is sort them..... Hmmmm..... This is awful! There seems to be no way of doing it. Just deciding whether **`score1`** is the largest value would take an if construction with 10 comparisons! Clearly there has to be a better way of doing this, after all, we know that computers are very good at sorting this kind of thing.

C# provides us with a thing called an *array*. An array allows us to declare a whole row of a particular kind of box. We can then use things called *subscripts* to indicate which box in the row that we want to use. Consider the following:

```
using System;
 class ArrayDemo {
     public static void Main ()
     {
         int [] scores = new int [11]  ;
         for ( int i=0; i<11; i=i+1) {
             string scoreString = Console.ReadLine();
             scores [i] = int.Parse(scoreString);
         }
     }
 }
```

The **`int [] scores`** part tells the compiler that we want to create an array variable. You can think of this as a tag which can be made to refer to a given array.

The bit which makes the array itself is the **`new int [11]`**. When C# sees this it says "aha! What we need here is an array". It then gets some pieces of wood and makes a long thin box with 11 compartments in it, each large enough to hold a single integer. It then paints the whole box red - because boxes which can hold integers are red. It then gets a piece of rope and ties the tag scores to this box. If you follow the rope from the scores tag you reach the array box. Actually, it probably doesn't use wood or rope, but you should get a picture of what is going on here.

Array Elements

Each compartment in the box is called an element. In the program you identify which element you mean by putting its number in square brackets **`[ ]`** after the array name. This part is called the *subscript*. Note that the thing which makes arrays so wonderful is the fact that you can specify an element by using a variable, as well as a constant. In fact you can use any expression which returns an integer result as a subscript, i.e.

```
scores [i+1]
```

- is quite OK. (as long as you don't fall off the end of the array)

Array Element Numbering

C# numbers the boxes starting at 0. This means that you specify the first element of the array by giving the subscript 0. There is consequently no element **`scores [11]`**. If you look at the part of the program which reads the values into the array you will see that we only count from 0 to 10. This is very important. An attempt to go outside the array bounds of **`scores`** cause your program to fail as it runs.

Large Arrays

The real power of arrays comes from our being able to use a variable to specify the required element. By running the variable through a range of values we can then scan through an array with a very small program; indeed to change the program to read in 1000 scores we only have to make a couple of changes:

```
using System;
class ArrayDemo {
    public static void Main ()
    {
        int [] scores = new int [1000]  ;
        for ( int i=0; i<1000; i=i+1) {
            string scoreString = Console.ReadLine();
            scores [i] = int.Parse(scoreString);
        }
    }
}
```

The variable **`i`** now ranges from 0 to 999, vastly increasing the amount of data we are storing.

Managing Array Sizes

A good trick when working with arrays is to make use of constant variables to hold the size of the array. This has two significant benefits:

- It makes the program easier to understand
- It makes the program easier to change

A constant variable is given a value when it is declared. This value can then only be read by the program, never updated. So, if I wanted to write a scores program that could be easily changed for any size of team I could write:

```
using System;
 class ArrayDemo {
     public static void Main ()
     {
         const int SCORE_SIZE = 1000;
         int [] scores = new int [SCORE_SIZE]  ;
         for ( int i=0; i < SCORE_SIZE; i=i+1) {
             string scoreString = Console.ReadLine();
             scores [i] = int.Parse(scoreString);
         }
     }
 }
```

The variable **`SCORE_SIZE`** is an integer which has been marked with **`const`**. This means that it cannot be changed by statements within the program. It will never have a value other than 1000. There is a convention that constants of this kind are given in LARGE LETTERS with an underscore between words.

Everywhere I previously used a fixed value to represent the size of the array I now use my constant instead. This means that if the size of the team changes I just have to change the value assigned when the constant is declared and then re-compile the program. The other benefit of this is that the **for** loop now looks a lot more meaningful. Since the value of **i** is now going from **0** to **SCORE_SIZE** it is more obvious to the reader that it is working through the score array.

Creating a Two Dimensional Array

However, sometimes we want to hold more than just a row. Sometimes we want a grid. We can do this by creating a *two dimensional* array. You can think of this as an "array of arrays" if you like (but only if this doesn't make your head hurt). For example, to hold the board for a game of noughts and crosses we could use:

```
int [,] board = new int [3,3];
board [1,1] = 1;
```

This looks very like our one dimensional array, but there are some important differences. The **[,]** now has a comma. The presence of a comma implies something each side of it. This means that the array now has two dimensions, rather than just one. So when we give the size of the board we must supply two dimensions rather than just one. Then, when we want to specify an element we have to give two subscript values. In the code above I've set the value in the middle square (the best one) to 1.

In the example above the array is square (i.e. the same dimension across as up). We can change this if we like:

```
int [,] board = new int [3,10];
```

- but this would make it rather hard to play a sensible game on...

You can think of a two dimensional array as a grid if you like. The subscripts become x and y values which indicate a particular value in the grid.

More than Two Dimensions

Once in a blue moon you may need to use more than two dimensions. If you go to three dimensions you can think in terms of a pile of grids if you like, with the third dimension (which you could call z) giving you the particular grid. If we wanted to play three dimensional noughts and crosses in a board which is a cube we can declare the array to do it as follows:

```
int [,,] board = new int [3,3,3];
board [1,1,1] = 1;
```

This code creates a three dimensional board and then gets the highly valuable location right in the middle of the game cube.

You can go to more than three dimensions if you like, in that C# does not have a problem with this. However, you might have big problems because this is very hard to understand and visualise.

Programmer's Point:Keep your dimensions low

In all my years of programming I've never had to use anything more than three dimensions. If you find yourself having lots of dimensions I would suggest that you are trying to do things the wrong way and should step back from the problem. It may be that you can get a much more efficient solution by creating a struct and then making an array of the structure items. We will talk about structures later.

Switching

We now know nearly everything you need to know about constructing a program in the C# language. You may find it rather surprising, but there is really very little left to know about programming itself. Most of the rest of C is concerned with making the business of programming simpler. A good example of this is the **switch** construction.

Making Multiple Decisions

Suppose you are refining your double glazing program to allow your customer to select from a pre-defined range of windows. You ask something like

```
Enter the type of window:

1 = casement
2 = standard
3 = patio door
```

Your program can then calculate the cost of the appropriate window by selecting type and giving the size. Each method asks the relevant questions and works out the price of that kind of item.

When you come to write the program you will probably end up with something like:

```
static void handleCasement ()
{
    Console.WriteLine("Handle Casement");
}

static void handleStandard ()
{
    Console.WriteLine("Handle Standard");
}
static void handlePatio ()
{
    Console.WriteLine("Handle patio");
}
```

These methods are the ones which will eventually deal with each type of window. At the moment they just print out that they have been called. Later you will go on and fill the code in (this is actually quite a good way to construct your programs).

Selecting using the `if` construction

When you come to perform the actual selection you end up with code which looks a bit like this:

```
int selection ;
selection = readInt ( "Window Type : ", 1, 3 ) ;

if ( selection == 1 )
{
    handleCasement();
}
else
{
    if ( selection == 2 )
    {
            handleStandard();
    }
```

```
    else
    {
            if ( selection == 3 )
            {
                    handlePatio() ;
            }
            else
            {
                    Console.WriteLine ( "Invalid number" );
            }
    }
}
```

This would work OK, but is rather clumsy. You have to write a large number of **if** constructions to activate each option.

The switch construction

Because you have to do this a lot C# contains a special construction to allow you to select one option from a number of them based on a particular value. This is called the *switch* construction. If you write the above using it your program would look like this.

```
switch (selection)
{
    case 1 : handleCasement ();
             break ;
    case 2 : handleStandard () ;
             break ;
    case 3 : handlePatio () ;
             break ;
    default :
             Console.WriteLine ( "Invalid number" ) ;
             break ;
}
```

The **switch** construction takes a value which it uses to decide which option to perform. It executes the **case** which matches the value of the **switch** variable. Of course this means that the type of the cases that you use must match the switch selection value although, in true C# tradition, the compiler will give you an error if you make a mistake. The **break** statement after the call of the relevant method is to stop the program running on and performing the code which follows. In the same way as you break out of a loop, when the **break** is reached the **switch** is finished and the program continues running at the statement after the switch.

Another other useful feature is the **default** option. This gives the switch somewhere to go if the switch value doesn't match any of the cases available; in our case (sorry!) we put out an appropriate message.

You can use the switch construction with types other than numbers if you wish:

```
switch (command)
{
    case "casement" : handleCasement ();
             break ;
    case "standard" : handleStandard () ;
             break ;
    case "patio" : handlePatio () ;
             break ;
    default :
             Console.WriteLine ( "Invalid command" ) ;
             break ;
}
```

This switch uses a string to control the selection of the cases. However, your users would not thank you for doing this, since it means that they have to type in the complete

name of the option, and of course if they type a character wrong the command is not recognised.

Programmer's Point:switches are a good idea

Switches make a program easier to understand as well as quicker to write. It is also easier to add extra commands if you use a switch since it are just a matter of putting in another case. However, I'd advise against putting large amounts of program code into a switch case. Instead you should put a call to a method as I have above.

Our Case Study: Friendly Bank

The bulk of the text is based on a case study which will allow you to see the features of C# in a strong context. You are taking the role of a programmer who will be using the language to create a solution for a customer.

The program we are making is for a bank, the "United Friendly and Really Nice Bank of Lovely People ™", otherwise known as the Friendly Bank. We will be creating the entire bank application using C# and will be exploring the features of C# that make this easy.

Bank System Scope

The *scope* of a system is a description of the things that the system is going to do. This is also, by implication, a statement of what the system will **not** do. This equally as important, as a customer will not usually have clear idea of what you are doing and may well expect you to deliver things that you have no intention of providing. By setting out the scope at the beginning you can make sure that there are no unpleasant surprises later on.

At the moment we are simply concerned with managing the account information in the bank. The bank manager has told us that customers of the bank each have an account which holds their name, address, account number, balance and overdraft value. There are many thousands of customers and the manager has also told us that there are also a number of different types of accounts (and that new types of account are invented from time to time). The system must also generate warning letters and statements as required. The scope does not include to telephone or web based banking, yet.

Bank Notes

At the end of some sections there will be a description of how this new piece of C# will affect how we create our bank system. These notes should put the feature into a useful context.

Enumerated Types

These sound really posh. If anyone asks you what you learnt today you can say "I learnt how to use enumerated types" and they will be really impressed. Of course if they know

about programming they'll just say "Oh, you mean you've numbered some states" and not be that taken with it.

Enumeration and states

Enumerated sounds posh. But if you think of "enumerated" as just meaning "numbered" things get a bit easier. To understand what we are doing here we need to consider the problem which these types are intended to solve.

We know that if we want to hold an integer value we can use an **`int`** type. If we want to hold something which is either true or false we can use a **`bool`**. However, sometimes we want to hold a range of particular values or states.

Sample states

Enumerated types are very useful when storing *state* information. States are not quite the same as other items such as the name of a customer or the balance of their account.

For example, if I am writing a program to play the game Battleships (where squares of the "sea" hold different types of craft which can be attacked) I may decide that a given square of the sea can have the following thing in it:

- Empty sea
- Attacked
- Battleship
- Cruiser
- Submarine
- Rowing boat

If you think about it, I am sort of assembling more metadata here, in that I have decided that I need to keep track of the sea and then I have worked out exactly what I can put in it. I could do something with numbers if I like:

- Empty sea = 1
- Attacked = 2
- Battleship = 3
- Cruiser = 4
- Submarine = 5
- Rowing boat = 6

However, this would mean that I have to keep track of the values myself and remember that if we get the value 7 in a sea location this is clearly wrong.

C# has a way in which we can create a type which has just a particular set of possible values. These types are called "enumerated types":

```
enum SeaState {
    EmptySea,
    Attacked,
    Battleship,
    Cruiser,
    Submarine,
    RowingBoat
} ;
```

I have created a type called **`SeaState`** which can be used to hold the state of a particular part of the sea. It can only have the given values above, and must be managed solely in terms of these named enumerations. For example I must write:

```
SeaState openSea ;
openSea = SeaState.EmptySea;
```

My variable **openSea** is only able to hold values which represent the state of the sea contents. Of course C# itself will actually represent these states as particular numeric values, but how these are managed is not a problem for me.

Creating an enum type

The **enum** type must be created outside any program block and is held within the enclosing class:

```
using System;

class EnumDemonstration {

    enum TrafficLight {
        Red,
        RedAmber,
        Green,
        Amber
    } ;

    public static void Main () {
        TrafficLight light ;
        light = TrafficLight.Red;
    }
}
```

Every time that you have to hold something which can take a limited number of possible values, or states (for example **OnSale**, **UnderOffer**, **Sold**, **OffTheMarket** etc) then you should think in terms of using enumerated types to hold the values.

Programmer's Point:Use enumerated types

Enumerated types are another occasion where everyone benefits if you use them. The program becomes simpler to write, easier to understand and safer. You should therefore use them

For the bank you want to hold the state of an item as well as other information about the customer. For example, we could have the states "Frozen", "New", "Active", "Closed" and "Under Audit" as states for our bank account. If this is the case it is sensible to create an enumerated type which can hold these values and no others

```
enum accountState {
    New,
    Active,
    UnderAudit,
    Frozen,
    Closed
} ;
```

We now have a variable which can hold state information about an account in our bank.

Structures

Structures let us organise a set of individual values into a cohesive lump which we can map onto one of the items in the problem that we are working on. This is important in many applications.

What is a Structure?

Often when you are dealing with information you will want to hold a collection of different things about a particular item. The Friendly Bank has commissioned an account storage system and you can use structures to make this easier. Like any good programmer who has been on my course you would start by doing the following:

1. Establish precisely the specification, i.e. get in written form exactly what they expect your system to do.
2. Negotiate an extortionate fee.
3. Consider how you will go about storing the data.

A sample structure

From your specification you know that the program must hold the following:

- customer name - string
- customer address - string
- account number - integer value
- account balance - integer value
- overdraft limit - integer value

The Friendly Bank have told you that they will only be putting up to 50 people into your bank storage so, after a while you come up with the following:

```
const int MAX_CUST = 50;

string [] names = new string [MAX_CUST] ;
string [] addresses = new string [MAX_CUST] ;
int [] accountNos = new int [MAX_CUST] ;
int [] balances = new int [MAX_CUST] ;
int [] overdraft = new int [MAX_CUST] ;
```

What we have is an array for each single piece of data we want to store about a particular customer. If we were talking about a database (which is actually what we are writing), the lump of data for each customer would be called a record and an individual part of that lump, for example the overdraft value, would be called a field. In our program we are working on the basis that **`balance[0]`** holds the balance of the first customer in our database, **`overdraft [0]`** holds the overdraft of the first customer, and so on. (Remember that array subscript values start at 0).

This is all very well, and you could get a database system working with this data structure. However it would me much nicer to be able to lump your record together in a more definite way.

Creating a Structure

C# lets you create data structures. A structure is a collection of C# variables which you want to treat as a single entity. In C# a lump of data would be called a structure and each part of it would be called a field. To help us with our bank database we could create a structure which could hold all the information about a customer:

```
struct Account
{
    public string Name ;
    public string Address ;
    public int AccountNumber ;
    public int Balance ;
    public int Overdraft ;
} ;
```

This defines a structure, called **customer**, which contains all the required customer information. Having done this we can now define some variables:

```
Account RobsAccount ;
Account [] Bank = new Account [MAX_CUST];
```

The first declaration sets up a variable called **RobsAccount**, which can hold the information for a single customer. The second declaration sets up an entire array of customers, called **Bank** which can hold all the customers.

We refer to individual members of a structure by putting their name after the struct variable we are using with a . (full stop) separating them, for example:

```
RobsAccount.AccountNumber
```

- would refer to the integer field **AccountNumber** in the structured variable **RobsAccount**. (i.e. the **AccountNumber** value in **RobsAccount**)

You can do this with elements of an array of structures too, so that:

```
Bank [25].Name
```

- would be the string containing the name of the customer in the element with subscript **25**.

Using a Structure

A program which creates and sets up a structure looks like this:

```
using System;

class BankProgram {
    // structure
    struct Account
    {
        public string Name ;
        public string Address ;
        public int AccountNumber ;
        public int Balance ;
        public int Overdraft ;
    } ;
    // program
    public static void Main () {
        Account RobsAccount ;
        RobsAccount.Name = "Rob Miles";
        RobsAccount.Address = "His house";
        RobsAccount.AccountNumber = 1234;
        RobsAccount.Balance = 0;
        RobsAccount.Overdraft = -1000;
        Console.WriteLine ( "Name is : " + RobsAccount.Name ) ;
        Console.WriteLine ( "Balance is : " + RobsAccount.Balance ) ;
    }
}
```

Note how the structure is declared outside the **Main** method. This is so that it can be used by any methods in the class. Note also that once I have created my structure I can use it in the same way that I would use something like **int** or **float**.

This program doesn't create an array of structures, but it does show you how to access the various fields in a single structure variable.

Initial values in structures

When a structure is created as a local variable (i.e. in a block) the values in it are undefined. This means that if you try to use them in your program you will get a compilation error. This is exactly the same as if you use a variable in a program before giving it a value. In other words:

```
Account RobsAccount ;
Console.WriteLine ( "Name is : " + RobsAccount.Name ) ;
```

- would produce a compilation error. It is your job as programmer to make sure that you always put a value into a variable before you try to get something out of it.

Programmer's Point:Structures are crucial

In a commercial system it is common to spend a very long time designing the structures which make up the data storage. They are the fundamental building blocks of the program since they hold all the data upon which everything else is built. You can regard the design of the structures and the constraints on their content as another big chunk of metadata about a system that you create.

Enumerated Types in Structures

Since any given account instance will have a particular state it makes sense to add a state value to each one:

```
enum accountState {
    New,
    Active
    UnderAudit,
    Frozen,
    Closed
} ;

struct Account
{
    public accountState state ;
    public string Name ;
    public string Address ;
    public int AccountNumber ;
    public int Balance ;
    public int Overdraft ;
} ;
```

Objects, Structures and References

You have seen that if you want to store a block of information about a particular item you can bring all this together in a structure. Structures are useful, but we would like to be able to solve other problems when we write large programs:

- We want to make sure that a given item in our program cannot be placed into an invalid state, i.e. we don't want to have bank accounts with empty or incorrect account numbers.
- We want to be able to break a large system down into distinct and separate components which can be developed independently and interchanged with others which do the same task, i.e. we want to get one team of programmers working on accounts, another on cheques, another on credit cards etc.
- We want to make sure that the effort involved with making new types of bank account is as small as possible, i.e. if the bank decides to introduce a new high interest deposit account we want to be able to make use of existing deposit account

To do all these things we are going to have to start to consider programs from the point of view of object based design. This section should come with some kind of a health warning along the lines of "some of these ideas might hurt your head a bit at the start". But the following points are also very important:

- objects don't add any new behaviours to our programs – we know just about everything we need to know to write programs when we have learnt about statements, loops, conditions and arrays.
- objects are best regarded as a solution to the problem of design. They let us talk about systems in general terms. We can go back and refine how the objects actually do their tasks later.

You can write just about every program that has ever been written just by using the technologies that we have so far looked at. But objects allow us to work in a much nicer way. And so we are going to have to get the hang of them, like it or not…

Objects and Structures

In C# objects and structures have a lot in common. They can both hold data and contain methods. However, there is a crucial difference between the two. Structures are managed in terms of *value* whereas objects are managed in terms of *reference*.

It is very important that you understand the distinction between the two, for it has a big impact on the way that they are used.

Creating and Using a Structure

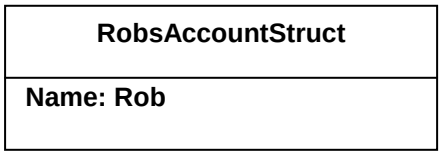
Consider the code:

```
class StructsAndObjects {

    struct AccountStruct
    {
           public string Name ;
    } ;

    public static void Main () {
           AccountStruct RobsAccountStruct ;
           RobsAccountStruct.Name = "Rob";
           Console.WriteLine ( RobsAccountStruct.Name );  }
}
```

This implements a very simple bank account, where we are only holding the name of the account holder. The **`Main`** method creates a structure variable called **`RobsAccountStruct`**.



It then sets the name property of the variable to the string "Rob". If we run this program it does exactly what you would expect, in that it prints out the name "Rob". If the structure contained other items about the bank account these would be stored in the structure as well, and I could use them in just the same way.

Creating and Using an Instance of a Class

We can make a tiny change to the program and convert the bank account to a class:

```
class StructsAndObjects {

    class Account
    {
           public string Name ;
    } ;

    public static void Main () {
           Account RobsAccount ;
           RobsAccount.Name = "Rob";
           Console.WriteLine (RobsAccount.Name );  }
}
```

The account information is now being held in a class, rather than a structure. I've changed the names to make it even clearer. The problem is that when we compile the program we get this:

```
ObjectDemo.cs(12,3): error CS0165: Use of unassigned local
variable ' RobsAccount'
```

So, what is going on?

To understand what is happening you need to know what is performed by the line:

```
Account RobsAccount;
```

This looks like a declaration of a variable called **RobsAccount**. But in the case of objects, this is not what it seems.

What you actually get when the program obeys that line is the creation of a *reference* called **RobsAccount**. Such references are allowed to *refer* to instances of the **Account**. You can think of them as a bit like a luggage tag, in that they can be tied to something with a piece of rope. If you have the tag you can then follow the rope to the object it is tied to.

But when we create a reference we don't actually get one of the things that it refers to. The compiler knows this, and so it gives me an error because the line:

```
RobsAccount.Name = "Rob";
```

- is an attempt to find the thing that is tied to this tag and set the name property to "Rob". Since the tag is presently not tied to anything our program would fail at this point. The compiler therefore says, in effect, "you are trying to follow a reference which does not refer to anything, therefore I am going to give you a 'variable undefined' error".

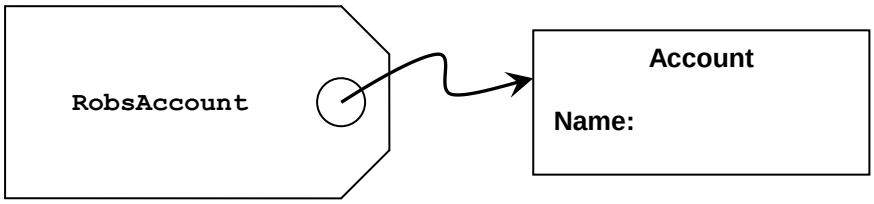
We solve the problem by creating an instance of the class and then connecting our tag to it. This is achieved by adding a line to our program:

```
class StructsAndObjects {

    class Account
    {
            public string Name ;
    } ;

    public static void Main () {
            AccountClass RobsAccount ;
            RobsAccount = new Account();
            RobsAccount.Name = "Rob";
            Console.WriteLine (RobsAccount.Name );  }
}
```

The line I have added creates a new **AccountClass** object and sets **RobsAccount** to refer to it.



We have seen this keyword **new** before. We use it to create arrays. This is because an array is actually implemented as an object, and so we use **new** to create it. The thing that new creates is an *object.* An object is an instance of a class. I'll repeat that in a posh font:

"An object is an instance of a class"

I have repeated this because it is very important that you understand this. A class provides the instructions to C# as to what is to be made, and what it can do. The **new** keyword causes C# to use the class information to actually make an instance. Note that in the above diagram I have called the object an **Account**, not **RobsAccount**. This is because the object instance does not have the identifier **RobsAccount**, it is simply the one which **RobsAccount** is connnected to at the moment.

References

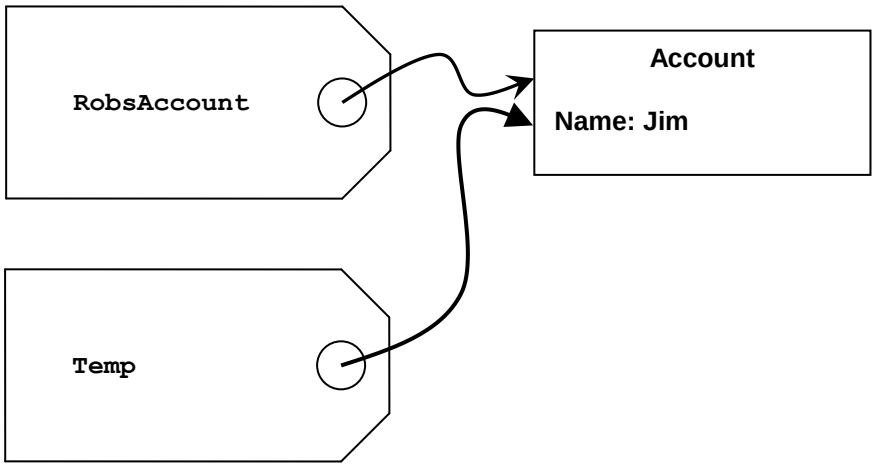
We now have to get used to the idea that if we want to use objects, we have to use references. The two come hand in hand and are inseparable. Structures are kind of useful, but for real object oriented satisfaction you have to have an object, and that means that we must manage our access to a particular object by making use of references to it. Actually this is not that painful in reality, in that you can treat a reference as if it really was the object just about all of the time, but you must remember that when you hold a reference you do not hold an instance, you hold a tag which is tied onto an instance…

Multiple References to an Instance

Perhaps another example of references would help at this point. Consider the following code:

```
Account RobsAccount ;
RobsAccount = new Account();
RobsAccount.Name = "Rob";
Console.WriteLine (RobsAccount.Name );
Account Temp ;
Temp = RobsAccount;
Temp.Name = "Jim";
Console.WriteLine (RobsAccount.Name );
```

The question is; what would the second call of **WriteLine** print out? If we draw a diagram the answer becomes clearer:



Both of the tags refer to the same instance of **Account**. This means that any changes which are made to the object that **Temp** refers to will also be reflected in the one that **RobsAccount** refers to, *because they are the same object*. This means that the program would print out Jim, since that is the name in the object that **RobsAccount** is referring to.

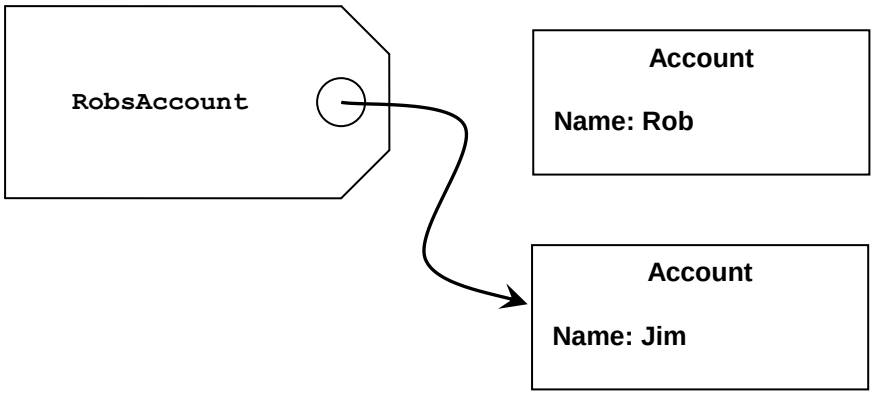
This indicates a trickiness with objects and references. There is no limit to the number of references that can be attached to a single instance, so you need to remember that changing the object that a reference refers to may well change that instance from the point of view of other objects.

No References to an Instance

Just to complete the confusion we need to consider what happens if an object has no references to it:

```
Account RobsAccount ;
RobsAccount = new Account();
RobsAccount.Name = "Rob";
Console.WriteLine (RobsAccount.Name );
RobsAccount = new Account();
RobsAccount.Name = "Jim";
Console.WriteLine (RobsAccount.Name );
```

This code makes an account instance, sets the name property of it to Rob and then makes another account instance. The reference **RobsAccount** is made to refer to the new item, which has the name set to Jim. The question is: What happens to the first instance? Again, this can be made clearer with a diagram:



The first instance is shown "hanging" in space, with nothing referring to it. As far as making use of data in the instance is concerned, it might as well not be there. Indeed the C# language implementation has a special process, called the "Garbage Collector" which is given the job of finding such useless items and disposing of them. Note that the compiler will not stop us from "letting go" of items like this.

You should also remember that you can get a similar effect when a reference to an instance goes out of scope:

```
{
    Account localVar ;
    localVar = new Account();
}
```

The variable **localVar** is local to the block. This means that when the program execution leaves the block the local variable is discarded. This means that the only reference to the account is also removed, meaning another job for the garbage collector..

Programmer's Point:Try to avoid the Garbage Collector

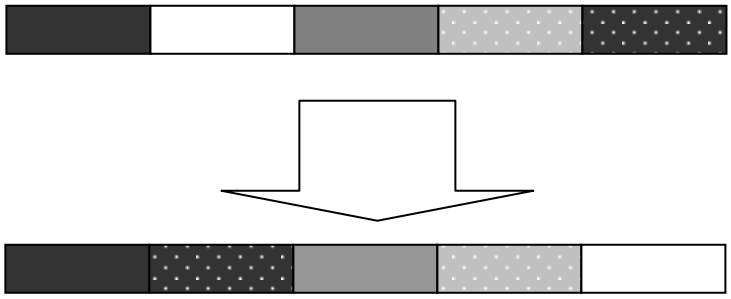
While it is sometimes reasonable to release items you have no further use for, you must remember that creating and disposing of objects will take up computing power. When I work with objects I worry about how much creating and destroying I am doing. Just because the objects are disposed of automatically doesn't mean that you should abuse the facility.

Why Bother with References?

References don't sound much fun at the moment. They seem to make it harder to create and use objects and may be the source of much confusion. So why do we bother with them?

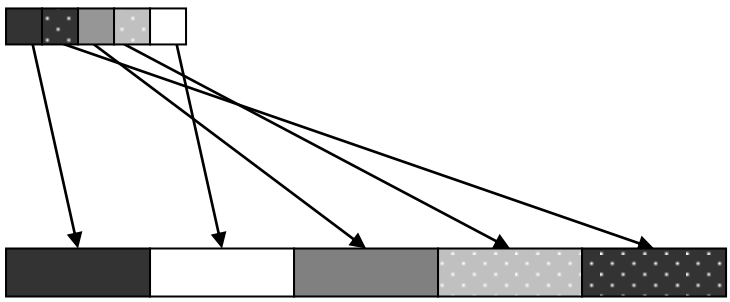
To answer this we can consider the Pacific Island of Yap. The currency in use on this island is based around 12 feet tall stones which weigh several hundred pounds each. The value of a "coin" in the Yap currency is directly related to the number of men who died in the boat bringing the rock to the island. When you pay someone with one of these coins you don't actually pick it up and give it to them. Instead you just say "The coin in the road on top of the hill is now yours". In other words they use references to manage objects that they don't want to have to move around.

That is why we use references in our programs. Consider a bank which contains many accounts. If we wanted to sort them into alphabetical order of customer name we have to move them all around.



Sorting by moving objects around

If we held the accounts as an array of structure items we would have to do a lot of work just to keep the list in order. The bank may well want to order the information in more than one way too, for example they might want to order it on both customer surname and also on account number. Without references this would be impossible. With references we just need to keep a number of arrays of references, each of which is ordered in a particular way:

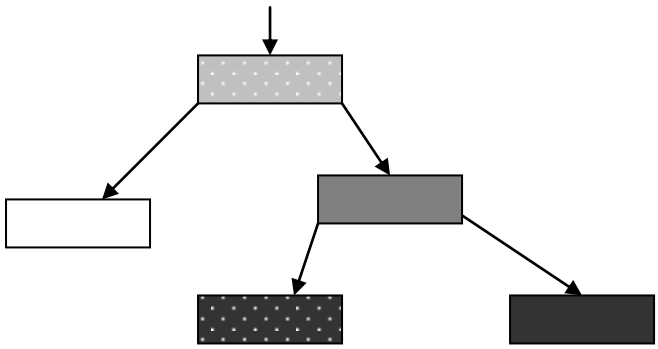


Sorting by using references.

If we just sort the references we don't have to move the large data items at all. New objects can be added without having to move any objects, instead the references can be moved around.

References and Data Structures

Our list of sorted references is all very good, but if we want to add something to our sorted list we still have to move the references around. We can get over this, and also speed up searching, by structuring our data into a tree form.



Sorting by use of a tree.

In the tree above each node has two references; one can refer to a node which is "lighter", the other to a node which is "darker". If I want a sorted list of the items I just have to go as far down the "lighter" side as I can and I will end up at the lightest. Then I go up to the one above that (which must be the next lightest). Then I go down the dark side (Luke) and repeat the process. The neat thing about this approach is also that adding new items is very easy; I just find the place on the tree that they need to be hung on and attach the reference there.

Searching is also very quick, in that I can look at each node and decide which way to look next until I either find what I am looking for or I find there is no reference in the required direction, in which case the item is not in the structure.

Programmer's Point:Data Structures are Important

This is not a data structures document, it is a programming document. If you don't get all the stuff about trees just yet, don't worry. Just remember that references are an important mechanism for building up structures of data and leave it at that. But some time in the future you are going to have to get your head around how to build structures using these things.

Reference Importance

The key to this way of working is that an object can contain references to other objects, as well as the data payload. We will consider this aspect of object use later; for now we just need to remember that the reference and the object are distinct and separate.

Bank Notes: References and Accounts

For a bank with many thousands of customers the use of references is crucial to the management of the data that they hold. The accounts will be held in the memory of the computer and, because of the size of each account and the number of accounts being stored, it will not be possible to move them around memory if we want to sort them.

This means that the only way to manipulate them is to leave them in the same place and have lists of references to them. The references are very small "tags" which can be used to locate the actual item in memory. Sorting a list of references is very easy, and it would also be possible to have several such lists. This means that we can offer the manager a view of his bank sorted by customer name and another view sorted in order of balance. And if the manager comes along with a need for a new structure or view we can create that in terms of references as well.

Designing With Objects

We are now going to start thinking in terms of objects. The reason that we do this is that we would like a way of making the design of our systems as easy as possible. This all comes back to the "creative laziness" that programmers are so famous for. The thing that we are trying to do here is best expressed as:

"Put off all the hard work for as long as we can, and if possible get someone else to do it."

Objects let us do this. If we return to our bank account we can see that there are a number of things that we need to be able to do with our bank account:

- pay money into the account
- draw money out of the account
- find the balance
- print out a statement
- change the address of the account holder
- print out the address of the account holder
- change the state of the account
- find the state of the account
- change the overdraft limit
- find the overdraft limit

Rather than saying "We need to do these operations on a bank account", object based design turns this on its head, a bit like President Kennedy did all those years ago:

"And so, my fellow Americans: ask not what your country can do for you—ask what you can do for your country" (huge cheers)

We don't do things to the bank account. Instead we ask it to do these things for us. The design of our banking application can be thought of in terms of identifying the objects that we are going to use to represent the information and then specifying what things they should be able to do. The really clever bit is that once we have decided what the bank account should do, we then might be able to get somebody else to make it do these things.

If our specification is correct and they implement it properly, we don't have to worry precisely how they made it work – we just have to sit back and take the credit for a job well done.

This brings us back to a couple of recurring themes in this document; *metadata* and *testing*. What a bank account object should be able to do is part of the metadata for this object. And once we have decided on the actions that the account must perform, the next thing we need to do is devise a way in which each of the actions can be tested.

In this section we are going to implement an object which has some of the behaviours of a proper bank account.

Programmer's Point:Not Everything Should Be Possible

Note that there are also some things that we should **not** be able to do with our bank account objects. The account number of an account is something which is unique to that account and should never change. We can get this behaviour by simply not providing a means by which it can be changed. It is important at design time that we identify what should not be possible, along with what should be done. We might even identify some things as being audited, in that an object will keep track of what has been done to it. That way we can easily find out if bad things are being done.

Data in Objects

So, we can consider our bank account in terms of what we want it to do for us. The first thing is to identify all the data items that we want to store in it. For the sake of simplicity, for now I'm just going to consider how I keep track of the balance of the accounts. This will let me describe all the techniques that are required without getting bogged down too much.

```
class Account
{
    public decimal Balance;
} ;
```

The **Account** class above holds the member that we need to store about the balance of our bank accounts. Members of a class which hold a value which describes some data which the class is holding are often called *properties.* I've used the decimal type for the account balance, since this is specially designed to hold financial values.

We have seen that each of the data items in a class is a *member* of it and stored as part of it. Each time I create an instance of the class I get all the members as well. We have already seen that it is very easy to create an instance of a class and set the value of a member:

```
Account RobsAccount ;
RobsAccount = new Account();
RobsAccount.Balance = 99;
```

The reason that this works is that the members of the object are all *public* and this means that anybody has direct access to them. This means that any programmer writing the application can do things like:

```
RobsAccount.Balance = 0;
```

- and take away all my money. If we are going to provide a way of stopping this from happening we need to protect the data inside our objects.

Member Protection inside objects

If objects are going to be useful we have to have a way of protecting the data within them. Ideally I want to get control when someone tries to change a value in my objects, and stop the change form being made if I don't like it. The posh word for this is *encapsulation*. I want all the important data hidden inside my object so that I have complete control over what is done with it. This technology is the key to my *defensive programming* approach which is geared to making sure that, whatever else happens, my part of the program does not go wrong.

For example, in our bank program we want to make sure that the balance is never changed in a manner that we can't control. The first thing we need to do is stop the outside world from playing with our balance value:

```
class Account
{
   private decimal balance;
} ;
```

The property is no longer marked as **public**. Instead it is now **private**. This means that the outside world no longer has direct access to it. If I write the code:

```
RobsAccount.balance = 0;
```

- I will get an error when I try to compile the program:

```
PrivateDemo.cs(13,3): error CS0122:
'PrivateMembers.Account.balance' is inaccessible due to its
protection level
```

The balance value is now held inside the object and is not visible to the outside world.

Changing `private` members

I can tell what you are thinking at this point. You are thinking "what is the point of making it private, now you can't change it at all". Well, thanks for the vote of confidence folks. It turns out that I can change the value, but only using code actually running in the class. Consider the program:

```
class Account
{
  private decimal balance = 0;

  public bool WithdrawFunds ( decimal amount )
  {
    if ( balance < amount )
    {
      return false ;
    }
    balance = balance - amount ;
    return true;
  }

} ;

class Bank {

  public static void Main () {
    Account RobsAccount;
    RobsAccount = new Account();
    if ( RobsAccount.WithdrawFunds (5) )
    {
      Console.WriteLine ( "Cash Withdrawn" ) ;
    }
    else
    {
      Console.WriteLine ( "Insufficient Funds" ) ;
    }
  }
}
```

This creates an account and then tries to draw five pounds out of it. This will of course fail, since the initial balance on my account is zero, but it shows how I go about providing access to members in an account. The method **`WithdrawFunds`** is a member of the **`Account`** class and can therefore access private members of the class.

Programmer's Point:Metadata makes Members and Methods

I haven't mentioned metadata for at least five minutes. So perhaps now is a good time. The metadata that I gather about my bank system will drive how I provide access to the members of my classes. In the code above the way that I am protecting the balance value is a reflection of how the customer wants me to make sure that this value is managed properly.

`public` Methods

You may have noticed that I made the **`WithdrawFunds`** method **`public`**. This means that code running outside the class can make calls to that method. This has got to be the case, since we want people to interact with our objects by calling methods in them. In general the rules are:

- if it is a data member (i.e. it holds data) of the class, make it **`private`**
- if it is a method member (i.e. it does something) make it **`public`**

Of course, the rules can be broken on special occasions. If you don't care about possible corruption of the member and you want your program to run as quickly as possible you can make a data member **`public`**. If you want to write a method which is only used inside a class and performs some special, secret, task you can make it **`private`**.

Programmer's Point:private data and public methods

If you look closely at the code I write (and I would advise you to do this – it is good stuff) you will find that when I write the name of a public item I use a capital letter to start the name (as in the case of **`WithdrawFunds`** to withdraw from our bank account). But I make the first letter of private members lower case (as in the case of the balance data member of our bank account). This makes it easy for someone reading my code, because they can see from the name of a class member whether or not it is public or private. The convention also extends to variables which are local to a block. These (for example the ubiquitous i) always start with a lower case letter.

A Complete Account Class

We can now create a bank account class which controls access to the balance value:

```
public class Account
{
      private decimal balance = 0;

      public bool WithdrawFunds ( decimal amount )
      {
            if ( balance < amount )
            {
                  return false ;
            }
            balance = balance - amount ;
            return true;
      }

      public void PayInFunds ( decimal amount )
      {
            balance = balance + amount ;
      }

      public decimal GetBalance ()
      {
            return balance;
      }
}
```

The bank account class that I have created above is quite well behaved. I have created three methods which I can use to interact with an account object. I can pay money in, find out how much is there and withdraw cash, for example:

```
Account test = new Account();
test.PayInFunds(50);
```

At the end of this set of statements the test account should have 50 pounds in it. If it does not my program is faulty. The method **`GetBalance`** is called an *accessor* since it allows access to data in my business object. I could write a little bit of code to test these methods:

```
Account test = new Account();
test.PayInFunds(50);
if ( test.GetBalance() != 50 ) {
    Console.WriteLine ( "Pay In test failed" );
}
```

My program now tests itself, in that it does something and then makes sure that the effect of that action is correct. Of course I must still read the output from all the tests, which is tedious. Later in the course we will consider the use of *unit tests* which make this much easier.

Programmer's Point:Test Driven Development – the only way

I love test driven development. If I ever write anything from now on you can bet your boots that I will write it using a test driven approach. This solves three problems that I can see:

Firstly you don't do the testing at the end of the project. This is usually the worst time to test, since you might be using code that you wrote some time back. If the bugs are in an old piece of code you have to go through the effort of remembering how it works. Far better to test the code as you write it, when you have the best possible understanding of what it is supposed to do.

The second good reason for using a test driven approach is that it lets you write code early in the project which will probably be useful later on. Many projects are doomed because people start writing code before they have a proper understanding of the problem. Writing the tests first is actually a really good way of refining your understanding. And there is a good chance that the tests that you write will be useful at some point too.

The final reason for using tests is that when I fix bugs in my program I need to be able to convince myself that the fixes I have applied have not broken some other part (about the most common way of introducing new faults into a program is to mend a bug). If I have a set of automatic tests that I can run after every bug fix I have a way of stopping this from happening.

Bank Notes: Protecting Account Members

The bank manager approves of our use of **`private`** and **`public`**. This means that other programmers (i.e. the people who write systems which use the **`Account`** class instances in the bank) will not be allowed unrestricted access to the very important information held inside them. This is very important.

Static Items

At the moment all the members that we have created in our class have been part of an instance of the class. This means that whenever we create an instance of the **`Account`** class we get a **`balance`** member. However, we can also create members which are held as part of the class, i.e. they exist outside of any particular instance.

Static class members

The **`static`** keyword lets us create members which are not held in an instance, but in the class itself.

It is very important that you learn what **`static`** means in the context of C# programs. We have used it lots in just about every program that we have ever written:

```
class AccountTest {
  public static void Main () {
    Account test = new Account();
    test.PayInFunds (50);
    Console.WriteLine ("Balance:" + test.GetBalance());
  }
}
```

The **AccountTest** class has a **static** member method called **Main**. We know that this is the method which is called to run the program. It is part of the **class AccountTest**. If I made fifty **AccountTest** instances, they would all share the same **Main** method. In terms of C# the keyword **static** flags a member as being part of the class, **not** part of an instance of the class. I will write that down again in a posh font, for it is important:

"A static member is a member of the class, not a member of an instance of the class"

I don't have to make an instance of the **AccountTest** class to be able to use the **Main** method. This is how my program actually gets to work, in that when it starts it has not made any instances of anything, and so this method **must** be there already, otherwise it cannot run.

Static does not mean "cannot be changed". I think this is time for more posh font stuff:

Static does not mean "cannot be changed".

Members of a class which have been made **static** can be used just like any other member of a class. Either a data member or a method can be made **static**.

Using a static data member of a class

Perhaps an example of **static** data would help at this point. Consider the interest rates of our bank accounts. The customer has told us that one of the members of the account class will need to be the interest rate on accounts. In the program we can implement this by adding another member to the class which holds the current interest rate:

```
public class Account {
  public decimal Balance ;
  public decimal InterestRateCharged ;
}
```

Now I can create accounts and set balances and interest rates on them. (of course if I was doing this properly I'd make this stuff private and provide methods etc, but I'm keeping things simple just now).

```
Account RobsAccount = new Account();
RobsAccount.Balance = 100;
RobsAccount.InterestRateChanged = 10;
```

The snag is; I've been told that the interest rate is held for all the accounts. If the interest rate changes it must change for **all** accounts. This means that to implement the change I'd have to go through all the accounts and update the rate. This would be tedious, and if I missed one account, possibly expensive.

I solve the problem by making the interest rate member **static**:

```
public class Account {
  public decimal balance ;
  public static decimal interestRateCharged ;
}
```

The interest rate is now part of the class, not part of any instance. This means that I have to change the way that I get hold of it:

```
Account RobsAccount = new Account();
RobsAccount.Balance = 100;
Account.InterestRateChanged = 10;
```

Since it is a member of the class I now have to use the class name to get hold of it instead of the name of the instance reference.

Programmer's Point:Static Data Members are Useful and Dangerous

When you are collecting metadata about your project you should look for things which can be made static. Things like the limits of values (the largest age that you are going to permit a person to have) can be made static. There might be a time where the age limit changes, and you don't want to have to update all the objects in your program.

But of course, as Spiderman's uncle said, "With great power comes great responsibility". You should be careful about how you provide access to static data items. A change to a single static value will affect your entire system. So they should always be made private and updated by means of method calls.

Using a `static` method in a class

We can make methods `static` too. We have been doing this for ages with the **`Main`** method. But you can also use them when designing your system. For example, we might have a method which decides whether or not someone is allowed to have a bank account. It would take in their age and income. It would then return true or false depending on whether these are acceptable or not:

```
public bool Allowed ( decimal income, int age ) {
  if ( ( income >= 10000 ) && ( age >= 18 ) ) {
      return true;
   }
   else {
      return false;
   }
}
```

This checks the age and income; you must be over 17 and have at least 1000 pounds income to be allowed an account. The snag is that, at the moment, we can't call the method until we have an **`Account`** instance. We can solve this by making the method `static`:

```
public static bool Allowed ( decimal income, int age ) {
  if ( ( income >= 10000 ) && ( age >= 18 ) ) {
      return true;
   }
   else {
      return false;
   }
}
```

Now the method is part of the class, not an instance of the class. I can now call the method by using the class name:

```
if ( Account.Allowed ( 25000, 21 ) ) {
  Console.WriteLine ( "Allowed Account");
}
```

This is nice because I have not had to make an instance of the account to find out if one is allowed..

Using member data in static methods

The **Allowed** method is OK, but of course I have hard wired the age and income methods into it. I might decide to make the method more flexible:

```
public class Account {

  private decimal minIncome = 10000;
  private int minAge = 18;

  public static bool Allowed(decimal income, int age) {
    if ( ( income >= minIncome) && ( age >= minAge) ) {
       return true;
    }
    else {
       return false;
    }
  }
}
```

This is a better design, in that I now have members of the class which set out the upper limits of the age and income. However it is a bad program, since the class above will not compile:

```
AccountManagement.cs(19,21): error CS0120: An object
reference is required for the nonstatic field, method, or
property 'Account.minIncome'

AccountManagement.cs(19,43): error CS0120: An object
reference is required for the nonstatic field, method, or
property  'Account.minAge'
```

As usual, the compiler is telling us exactly what is wrong; using language which makes our heads spin. What the compiler really means is that *"a static method is using a member of the class which is not static"*.

If that doesn't help, how about this: The members **minIncome** and **minAge** are held within *instances* of the **Account** class. However, a static method can run without an instance (since it is part of the class). The compiler is unhappy because in this situation the method would not have any members to play with. We can fix this (and get our program completely correct) by making the income and age limits **static** as well:

```
public class Account {

  private static decimal minIncome ;
  private static int minAge ;

  public static bool Allowed(decimal income, int age) {
    if ( ( income > minIncome) && ( age > minAge) ) {
       return true;
    }
    else {
       return false;
    }
  }
}
```

If you think about it, this makes perfect sense. The limit values should not be held as members of a class, since we want them to be the same for all instances of the class, therefore, making them **static** is what we should have done in the first place.

Programmer's Point:Static Method Members can be used to make Libraries

Sometimes in a development you need to provide a library of methods to do stuff. In the C# system itself there are a huge number of methods to perform maths functions, for example sin and cos. It makes sense to make these methods static, in that in this situation all we want is the method itself, not an instance of a class. Again, when you are building your system you should think about how you are going to make such methods available for your own use.

Bank Notes: Static Bank Information

The kind of problems that we can use **static** to solve in our bank are:

static member variable: the manager would like us to be able to set the interest rate for all the customer accounts at once. A single static member of the Account class will provide a variable which can be used inside all instances of the class. But because there is only a single copy of this value this can be changed and thereby adjust the interest rate for all the accounts. Any value which is held once for all classes (limits on values are another example of this) is best managed as a static value. The time it becomes impossible to use static is when the manager says "Oh, accounts for five year olds have a different interest rate from normal ones". At this point we know we can't use static because we need to hold different values for some of the instances.

static member method: the manager tells us that we need a method to determine whether or not a given person is allowed to have an account. I can't make this part of any Account instance because at the time it runs an account has not been generated. I must make it **static**, so that it can execute without an instance.

The Construction of Objects

We have seen that our objects are created when we use **new** to bring one into being:

```
test = new Account();
```

If you look closely at what is happening you might decide that what is happening looks quite a bit like a method call.

This is actually exactly what is happening. When an instance of a class is created the C# system makes a call to a *constructor* method in that class. The constructor method is a member of the class and it is there to let the programmer get control and set up the contents of the shiny new object. One of the rules of the C# game is that every single class **must** have a constructor method to be called when a new instance is created.

"But wait a minute", you say, "We've been making objects for a while and I've never had to provide a constructor method". This is because the C# compiler is, for a change, being friendly here. Rather than shout at you for not providing a constructor method, the compiler instead quietly creates a *default* one for you and uses that.

You might think this is strange, in that normally the compiler loses no time in telling you off when you don't do something, but in this case it is simply solving the problem without telling you. There are two ways to look at this:

- **nice compiler:** the compiler is trying to make life easier for you
- **evil compiler:** the compiler knows that if it does this automatically now you will suffer more later when you try to understand why you don't need to add one

How you regard the action of the compiler is up to you.

The Default Constructor

A constructor method has the same name as the class, but it does not return anything. It is called when we perform **new**. If you don't supply a constructor (and we haven't so far) the compiler creates one for us.

```
public class Account {

    public Account () {
    }
}
```

This is what the default constructor looks like. It is **public** so that it can be accessed from external classes who might want to make instances of the class. It accepts no parameters. If I create my own constructor the compiler assumes that I know what I'm doing and stops providing the default one. This can cause problems, which we will discuss later.

Our Own Constructor

For fun, we could make a constructor which just prints out that it has been called:

```
public class Account {


    public Account () {
      Console.WriteLine ( "We just made an account" );
    }
}
```

This constructor is not very constructive (ho ho) but it does let us know when it has been called. This means that when my program executes the line:

```
robsAccount = new Account();
```

- the program will print out the message:

```
We just made an account
```

Note that this is not very sensible, in that it will result in a lot of printing out which the user of the program might not appreciate, but it does show how the process works.

Feeding the Constructor Information

It is useful to be able to get control when an **Account** is created, but it would be even nicer to be able to feed information into the **Account** when I create it. As an example, I might want to set the name, address, and initial balance of an account holder when the account is created. In other words I want to do:

```
robsAccount = new Account("Rob Miles", "Hull", 0);
```

This could create a new account and set the name property to **Rob Miles**, the address to **Hull** and the initial balance to zero. It turns out that I can do this very easily, all I have to do is make the constructor method to accept these parameters and use them to set up the members of the class:

```
class Account {
  // private member data
  private string name;
  private string address;
  private decimal balance;

  // constructor
  public Account (string inName, string inAddress,
    decimal inBalance) {
    name = inName;
    address = inAddress;
    balance = inBalance;
  }
}
```

The constructor takes the values supplied in the parameters and uses them to set up the members of the **Account** instance that is being created. In this respect it behaves exactly as any other method call.

Note that adding a constructor like this has one very powerful ramification:

You **must** use the new constructor to make an instance of a class, i.e. the only way I can now make an **Account** object is by supplying a name, address and starting balance. If I try to do this:

```
robsAccount = new Account();
```

- the compiler will stop being nice to me and produce the error:

```
AccountTest.cs(9,27): error CS1501: No overload for method
'Account' takes '0' arguments
```

What the compiler is telling me is that there is no constructor in the class which does not have any parameters. In other words, the compiler only provides a default constructor if the programmer doesn't provide a constructor.

This can cause confusion if we have made use of the default constructor in our program and we then add one of our own. The default constructor is no longer supplied by the compiler and our program now fails to compile correctly. In that situation you have to either find all the calls to the default one and update them, or create a default constructor of your own for these calls to use. Of course you don't have to do this because your design of the program was so good that you never have this problem. Just like me, hem hem.

Overloading Constructors

Overload is an interesting word. In the context of the "Star Trek" science fiction series it is what they did to the warp engines in every other episode. In the context of a C# program it means:

"A method has the same name as another, but has a different set of parameters"

The compiler is quite happy for you to overload methods, because it can tell from the parameters given at the call of the method which one to use. In the context of the constructor of a class, what this means is that you can provide several different ways of constructing an instance of a class. For example, many (but not all) of your accounts will be created with a balance value of zero, i.e. nothing in the account. This means that we would like to be able to write

```
robsAccount = new Account("Rob Miles","Hull");
```

I've missed off the balance value, since I want to use the "default" one of zero. If your code does this the compiler simply looks for a constructor method which has two strings as parameters, and nothing else. Something a bit like this:

```
public Account (string inName, string inAddress){
  name = inName;
  address = inAddress;
  balance = 0;
}
```

Overloading a method name

In fact, you can overload any method name in your classes. This can be useful if you have a particular action which can be driven by a number of different items of data, for example you could provide several ways of setting the date of a transation:

```
SetDate ( int year, int month, int day)

SetDate ( int year, int julianDate )

SetDate ( string dateInMMDDYY)
```

A call of:

```
SetDate (23, 7, 2005 );
```

- would be matched up with the method which accepts three integer parameters and that code would be executed.

Constructor Management

If the **Account** class is going to have lots of constructor methods this can get very confusing for the programmer:

```
public Account (string inName, string inAddress,
  decimal inBalance) {
  name = inName;
  address = inAddress;
  balance = inBalance;
}

public Account (string inName, string inAddress){
  name = inName;
  address = inAddress;
  balance = 0;
}

public Account (string inName) {
  name = inName;
  address = "Not Supplied";
  balance = 0;
}
```

I've made three constructors for an **Account** instance. The first is supplied with all the information, the second is not given a balance and sets the value to 0. The third is not given the address either, and sets the address to "**Not Supplied**".

To do this I have had to duplicate code. Good programmers **hate** duplicating code. It is regarded as "dangerous extra work". The scary thing is that it is quite easy to do, just use the block copy command in the text editor and you can take the same piece of program and use it all over the place. But you should not do this. Because it is bad. If you need to change this piece of code you have to find every copy of the code and change it.

This happens more often than you'd think, even if you don't put a bug in your code, you still might find yourself having to change it because the specification changes. So, C# provides a way in which you can call one constructor from another. Consider:

```
public Account (string inName, string inAddress,
  decimal inBalance) {
  name = inName;
  address = inAddress;
  balance = inBalance;
}
public Account ( string inName, string inAddress ) :
  this (inName, inAddress, 0 ) {
}

public Account ( string inName ) :
  this (inName, "Not Supplied", 0 ) {
}
```

The keyword **this** means "another constructor in this class". As you can see in the code sample above, the highlighted bits of the code are calls to the first constructor. They simply pass the parameters which are supplied, along with any default values that we have created, on to the "proper" constructor to deal with. This means that the actual transfer of the values from the constructor into the object itself only happens in one method, and the other constructor methods just make calls to it.

The syntax of these calls is rather interesting, in that the call to the constructor takes place before the body of the constructor method. In fact it is outside the block completely. This is sensible, because it reflects exactly what is happening. The "**this**" constructor runs before the body of the other constructor is entered. In fact, in the code above, since the call of **this** does all the work, the body of the constructor can be empty.

Programmer's Point:Object Construction Should Be Planned

The way in which objects are constructed is something that you should plan carefully when you write your program. You should create one "master" constructor which handles the most comprehensive method of constructing the object. Then you should make all the other constructor methods use **this** to get hold of that method.

A constructor cannot fail

If you watch a James Bond movie there is usually a point at which 007 is told that the fate of the world is in his hands. Failure is not an option. Constructors are a bit like this. Constructors cannot fail. And this is a problem:

Whenever we have written methods in the past we have made sure that their behaviour is error checked so that the method cannot upset the state of our object. For example, attempts to withdraw negative amounts of money from a bank account should be rejected.

The whole basis of the way that we have allowed our objects to be manipulated is to make sure that they cannot be broken by the people using them. If you try to do

something stupid with a method call it should refuse to perform the action and return something which indicates that it could not do the job.

So we know that when we create a method which changes the data in an object we have to make sure that the change is always valid. For example, we would not let the following call succeed:

```
RobsAccount.PayInFunds (1234567890);
```

There will be an upper limit to the amount of cash you can pay in at once, so the **PayInFunds** method will refuse to pay the money in. But what is to stop the following:

```
RobsAccount = new Account ("Rob","Hull",1234567890);
```

Like James Bond, constructors are not allowed to fail. Whatever happens during the constructor call, it will complete and a new instance will be created.

This poses a problem. It looks as if we can veto stupid values at every point except the one which is most important, i.e. when the object is first created.

Programmer's Point:Managing Failure is Hard Work

This brings us on to a kind of recurring theme in our quest to become good programmers. Writing code to do a job is usually very easy. Writing code which will handle all the possible failure conditions in a useful way is much trickier. It is a fact of programming life that you will (or at least should) spend more time worrying about how things fail than you ever do about how they work correctly.

Constructors and Exceptions

The only way round this at the moment is to have the constructor throw an exception if it is unhappy. This means that the user of the constructor must make sure that they catch exceptions when creating objects, which is not a bad thing. The really clever way to do this is to make the constructor call the set methods for each of the properties that it has been given, and if any of them returns with an error the constructor should throw the exception at that point:

```
public Account (string inName, string inAddress){
  if ( SetName ( inName ) == false ) {
    throw new Exception ( "Bad name " + inName) ;
  }
  if ( SetAddress ( inAddress) == false ) {
    throw new Exception ( "Bad address" + inAddress) ;
  }
}
```

If we try to create an account with a bad name it will throw an exception, which is what we want. The only problem here is that if the address is wrong too, the user of the method will not know this until they have fixed the name and then called the constructor again.

I hate it when I'm using a program and this happens. It is normally when I'm filling in a form on the web. I type in my name wrong and it complains about that. Then I put my name right, and it complains about my address. What I want is a way in which I can get a report of all the invalid parts of the item at once. This can be done, at the expense of a little bit of complication:

```
public Account (string inName, string inAddress){
   string errorMessage = "";
   if ( SetName ( inName ) == false ) {
      errorMessage = errorMessage + "Bad name " + inName;
   }
   if ( SetAddress ( inAddress) == false ) {
      errorMessage = errorMessage + " Bad addr " + inAddress;
   }
   if ( errorMessage != "" ) {
     throw new Exception ( "Bad account" + errorMessage) ;
   }
 }
```

This version of the constructor assembles an error message which describes everything which is wrong with the account. Each new thing which is wrong is added to the message and then the whole thing is put into an exception and thrown back to the caller.

Programmer's Point:Consider the International Issues

The code above assembles a text message and sends it to the user when something bad happens. This is a good thing. However, if you write the program as above this might cause a problem when you install the code in a French branch of the bank. During the specification process you need to establish if the code is ever going to be created in multiple language versions. If it is you will need to manage the storage and selection of appropriate messages. Fortunately there are some C# libraries which are designed to make this easier.

Bank Notes: Constructing an Account

The issues revolving around the constructor of a class are not directly relevant to the bank account specification as such, since they really relate to how the specification is implemented, and not what the system itself actually does.

That said, if the manager says something like "The customer fills in a form, enters their name and address and this is used to create the new account" this gives you a good idea of what parameters should be supplied to the constructor.

From Object to Component

I take the view that as you develop as a software writer you go through a process of "stepping back" from problems and thinking at higher and higher levels. Posh people call this "abstraction". This is the progress that we have made so far:

- representing values by named locations (variables)
- creating actions which work on the variables (statements and blocks)
- putting behaviours into lumps of code which we can give names to. We can reuse these behaviours and also use them in the design process (methods)
- creating things which contain member variables as properties and member methods as actions (objects)

Rather than spend a lot of time at the start of a project worrying just how we are going represent an account and precisely what it should do, we just say "We need an account here" and then move on to other things. Later we will come back and revisit the problem in a greater level of detail, and from the point of view of what the **`Account`** class needs to do.

The next thing to do is consider how we take a further step back and consider expressing a solution using *components* and *interfaces*. In this section you will find out the difference between an object and a component, and how to design systems using them.

Components and Hardware

Before we start on things from a software point of view it is probably worth considering things from a hardware point of view. You should be familiar with the way that, in a typical home computer, some parts are not "hard wired" to the system. For example, the graphics adapter is usually a separate device which is plugged into the main board. This is good; because it means that I can buy a new graphics adapter at any time and fit it into the machine to improve the performance.

For this to work properly the people who make main boards and the people who make graphics adapters have had to agree on an *interface* between two devices. This takes the form of a large document which describes exactly how the two components interact, for example which signals are inputs, which signals are outputs and so on. Any main board which contains a socket built to the standard can accept a graphics card.

So, from the point of view of hardware, components are possible because we have created standard *interfaces* which describe exactly how they fit together.

Software components are **exactly the same**.

Why we Need Software Components?

At the moment you might not see a need for software components. When we are creating a system we work out what each of the parts of it need to do, and then we create those parts. It is not obvious at this stage why components are required.

Well, a system designed without components is exactly like a computer with a graphics adapter which part of the main board. It is not possible for me to improve the graphics adapter because it is "hard wired" into the system.

However, it is unfortunately the case that with our bank system we may have a need to create different forms of bank account class. For example, we might be asked to create a "BabyAccount" class which only lets the account holder draw out up to ten pounds each time. This might happen even after we have installed the system and it is being used.

If everything has been hard wired into place this will be impossible. By describing objects in terms of their interfaces however, we can use anything which behaves like an Account in this position.

Components and Interfaces

One point I should make here is that we are **not** talking about the *user interface* to our program. The user interface is the way a person using our program would make it work for them. These are usually either text based (the user types in commands and gets responses) or graphical (the user clicks on “buttons” on a screen using the mouse).

An *interface* on the other hand just specifies how a software component could be used by another software component. Please don’t be tempted to answer an exam question about the C# interface mechanism with a long description of how windows and buttons work. This will earn you zero marks.

Interfaces and Design

So, instead of starting off by designing classes we should instead be thinking about describing their interfaces, i.e. what it is they have to do. In C# we express this information in a thing called an *interface*. An interface is simply a set of method definitions which are lumped together.

Our first pass at a bank account interface could be as follows:

```
public interface IAccount {
   void PayInFunds ( decimal amount );
   bool WithdrawFunds ( decimal amount );
   decimal GetBalance ();
}
```

This says that the **IAccount** interface is comprised of three methods, one to pay money in; another to withdraw it and a third which returns the balance on the account. From the balance management point of view this is all we need. Note that at the interface level I am not saying how it should be done, I am instead just saying what should be done. An interface is placed in a source file just like a class, and compiled in the same way. It sets out a number of methods which relate to a particular task or role, in this case what a class must do to be considered a bank account.

Implementing an Interface in C#

Interfaces become interesting when we make a class *implement* them. Implementing an interface is a bit like a setting up a contract between the supplier of resources and the consumer. If a class implements an interface it is saying that for every method described in the interface, it has a corresponding implementation.

In the case of the bank account, I am going to create a class which implements the interface, so that it can be thought of as an account component, irrespective of what it really is:

```
public class CustomerAccount : IAccount {

   private decimal balance = 0;

   public bool WithdrawFunds ( decimal amount )
   {
      if ( balance < amount )
      {
         return false ;
      }
      balance = balance - amount ;
      return true;
   }

   public void PayInFunds ( decimal amount )
   {
      balance = balance + amount ;
   }

   public decimal GetBalance ()
   {
      return balance;
   }
}
```

The code above does not look that different from the previous account class. The only difference is the top line:

```
public class CustomerAccount : IAccount {
...
```

The highlighted part of the line above is where the programmer tells the compiler that this class implements the **IAccount** interface. This means that the class contains concrete versions of all the methods described in the interface. If the class does not contain a method that the interface needs you will get a compilation error:

```
error CS0535: 'AccountManagement.CustomerAccount' does not
implement interface member
'AccountManagement.IAccount.PayInFunds(decimal)'
```

In this case I missed out the **PayInFunds** method and the compiler complained accordingly.

References to Interfaces

Once we have made the **CustomerAccount** class compile, we have now got something which can be regarded in two ways:

- as a **CustomerAccount** (because that is what it is)
- as an **IAccount** (because that is what it can do)

People do this all the time. You can think of me in a whole variety of ways, here are two:

- Rob Miles the individual (because that is who I am)
- A university lecturer (because that is what I can do)

If you think of me as a lecturer you would be using the interface that contains methods like **GiveLecture**. And you can use the same methods with any other lecturer (i.e. person who implements that interface). From the point of view of the university, which has to manage a large number of interchangeable lecturers, it is much more useful for it to think of me as a lecturer, rather than Rob Miles the individual.

So, with interfaces we are moving away from considering classes in terms of what they are, and starting to think about them in terms of what they can do. In the case of our bank, this means that we want to deal with objects in terms of **IAccount**,(the set of account abilities) rather than **CustomerAccount** (a particular account class).

In C# terms this means that we need to be able to create reference variables which refer to objects in terms of interfaces they implement, rather than the particular type that they are. It turns out that this is quite easy:

```
IAccount account = new CustomerAccount();
account.PayInFunds(50);
```

The **account** variable is allowed to refer to objects which implement the **IAccount** interface. The compiler will check to make sure that **CustomerAccount** does this, and if it does, the compilation is successful.

Note that there will never be an instance of **IAccount** interface. It is simply a way that we can refer to something which has that ability (i.e. contains the required methods).

This is the same in real life. There is no such physical thing as a "lecturer", merely a large number of people who can be referred to as having that particular ability or role.

Using interfaces

Now that we have our system designed with interfaces it is much easier to extend it. I can create a **BabyAccount** class which implements the **IAccount** interface. This

implements all the required methods, but they behave slightly differently because we want all withdrawals of over ten pounds to fail:

```
public class BabyAccount : IAccount {

   private decimal balance = 0;

   public bool WithdrawFunds ( decimal amount )
   {
      if (amount > 10)
      {
         return false ;
      }
      if (balance < amount)
      {
         return false ;
      }
      balance = balance - amount ;
      return true;
   }

   public void PayInFunds ( decimal amount )
   {
      balance = balance + amount ;
   }

   public decimal GetBalance ()
   {
      return balance;
   }
}
```

The nice thing about this is that as it is a component we don't have to change all the classes which use it. When we create the account objects we just have ask if a standard account or a baby account is required. The rest of the system can then pick up this object and use it without caring exactly what it is. We will of course have to create some tests especially for it, so that we can make sure that withdrawals of more than ten pounds do fail, but using the new kind of account in our existing system is very easy.

Implementing Multiple Interfaces

A component can implement as many interfaces as are required. The **`IAccount`** interface lets me regard a component purely in terms of its ability to behave as a bank account. However, I may want to regard a component in a variety of ways. For example, the bank will want the account to be able to print itself out on paper.

You might think that all I have to do is add a print method to the **`IAccount`** interface. This would be reasonable if all I ever wanted to print was bank accounts. However, there will be lots of things which need to be printed, for example warning letters, special offers and the like. Each of these items will be implemented in terms of a component which provides a particular interface (**`IWarning`**, **`ISpecialOffer`** for example). I don't want to have to provide a print method in each of these, what I really want is a way that I can regard an object in terms of its ability to print.

This is actually very easy. I create the interface:

```
public interface IPrintToPaper {
   void DoPrint ();
}
```

Now anything which implements the **`IPrintToPaper`** interface will contain the **`DoPrint`** method and can be thought of in terms of its ability to print.

A class can implement as many interfaces as it needs. Each interface is a new way in which it can be referred to and accessed.

```
public class BabyAccount : IAccount, IPrintToPaper {
...
```

This means that a **BabyAccount** instance behaves like an account and it also contains a **DoPrint** method which can be used to make it print out.

Designing with Interfaces

If you apply the "abstraction" technique properly you should end up with a system creation process which goes along the lines of:

- gather as much *metadata* as you can about the problem; what is important to the customer, what values need to be represented and manipulated and the range of those values
- identify classes that you will have to create to represent the components in the problem
- identify the actions (methods) and the values (properties) that the components must provide
- put these into interfaces for each of the components
- decide how these values and actions are to be tested
- implement the components and test them as you go

You can/should do much of this on paper, before you write any code at all. There are also graphical tools that you can use to draw formal diagrams to represent this information. The field of Software Engineering is entirely based on this process.

Programmer's Point:Interfaces are just promises

An interface is less of a binding contract, and more a promise. Just because a class has a method called **PayInFunds** does not mean that it will pay money into the account; it just means that a method with that name exists within the class. Nothing in C# allows you to enforce a particular behaviour on a method; that is down to how much you trust the programmer that made the class that you are using, and how good your tests are. In fact, we sometimes use this to good effect when building a program, in that we can create "dummy" components which implement the interface but don't have the behaviour as such.

Bank Notes: Account Interfaces

The interface mechanism gives us a great deal of flexibility when making our components and fitting them together. It means that once we have found out what our bank account class needs to hold for us we can then go on to consider what we are going to ask the accounts to do. This is the **real** detail in the specification. Once we have set out an interface for a component we can then just think in terms of what the component must do, not precisely how it does it.

For example, the manager has told us that each bank account must have an account number. This is a very important value, in that it will be fixed for the life of the account and can never be changed. No two accounts should ever have the same number.

From the point of view of interface design this means that the account number will be set when the account is created and the account class will provide a method to let us get the value (but there will **not** be a method to set the account number).

We don't care what the method **GetAccountNumber** actually does, as long as it always returns the value for a particular. So this requirement ends up being expressed in the *interface* that is implemented by the account class.

```
interface IAccount {
    int GetAccountNumber ();
}
```

This method returns the integer which is the account number for this instance. By placing it in the interface we can say that the account must deliver this value, but we have not actually described how this should be done. The design of the interfaces in a system is just this. They state that we have a need for behaviours, but they do not necessarily state how they are made to work. I have added comments to give more detail about what the method does.

The need for things like account numbers, which really need to be unique in the world, has resulted in the creation of a set of methods in the C# libraries to create things called *Globally Unique Identifiers* or *GUIDs*. These are data items which are created based on the date, time and certain information about the host computer. Each GUID is unique in the world. We could use these in our Account constructor to create a GUID which allows each account to have a unique number.

Inheritance

Inheritance is another way we can implement creative laziness. It is a way that we can pick up behaviours from classes and just modify the bits we need to make new ones. In this respect you can regard it as a mechanism for what is called *code reuse*. It can also be used at the design stage of a program if you have a set of related objects that you wish to create.

Inheritance lets a class pick up behaviours from the class which is its parent. You can regard an interface as a statement by a class that it has a set of behaviours because it implements a given interface. If a class is descended from a particular parent class this means that it has a set of behaviours because it has *inherited* them from its parent. In short:

> Interface: "I can do these things because I have told you I can"
> Inheritance: "I can do these things because my parent can"

Extending a parent class

We can see an example of a use for inheritance in our bank account project. We have already noted that a **`BabyAccount`** must behave just like a **`CustomerAccount`** except in respect of the cash withdrawal method. Customer accounts can draw out as much as they want. Baby accounts are only allowed to draw up to 10 pounds out at a time.

We have solved this problem from a design point of view by using interfaces. By separating the thing that does the job from the description of the job (which is what an interface lets you do) we can get the whole banking system thinking in terms of **`IAccount`** and then plug in accounts with different behaviours as required. We can even create brand new accounts at any time after the system has been deployed. These can be introduced and work alongside the others because they behave correctly (i.e. they implement the interface).

But this does make things a bit tiresome when we write the program. We need to create a **`BabyAccount`** class which contains a lot of code which is duplicated in the **`CustomerAccount`** class. "This is not a problem" you probably think "I can use the editor block copy to move the program text across". But:

Programmer's Point:Block Copy is Evil

I still make mistakes when I write programs. You might think that after such a huge number of years in the job I get everything right every time. Wrong. And a lot of the mistakes that I make are caused by improper use of block copy. I write some code and find that I need something similar, but not exactly the same, in another part of the program. So I use block copy. Then I change most, but not all, of the new code and find that my program doesn't work properly.

Try not to do this. A great programmer writes every piece of code once, and only once. If you need to use it in more than one place, make it a method.

What we really want to do is pick up all the behaviours in the **CustomerAccount** and then just change the one method that needs to behave differently. It turns out that we can do this in C# using inheritance. When I create the **BabyAccount** class I can tell the compiler that it is based on the **CustomerAccount** one:

```
public class BabyAccount : CustomerAccount,IAccount {
}
```

The key thing here is the highlighted part after the class name. I have put the name of the class that **BabyAccount** is *extending*. This means that everything that **CustomerAccount** can do, **BabyAccount** can do.

I can now write code like:

```
BabyAccount b = new BabyAccount();
b.PayInFunds(50);
```

This works because, although **BabyAccount** does not have a **PayInFunds** method, the parent class does. This means that the **PayInFunds** method from the **CustomerAccount** class is used at this point.

So, instances of the **BabyAccount** class have abilities which they pick up from their parent class. In fact, at the moment, the **BabyAccount** class has no behaviours of its own; it gets everything from its parent.

Overriding methods

We now know that we can make a new class based on an existing one. The next thing we need to be able to do is change the behaviour of the one method that we are interested in. We want to replace the **WithdrawFunds** method with a new one. This is called *overriding* a method. In the **BabyAccount** class I can do it like this:

```
public class BabyAccount : CustomerAccount,IAccount {

   public override bool WithdrawFunds (decimal amount)
   {
      if (amount > 10)
      {
         return false ;
      }
      if (balance < amount)
      {
         return false ;
      }
      balance = balance - amount ;
      return true;
   }
}
```

The keyword override means "use this version of the method in preference to the one in the parent". This means that code like:

```
BabyAccount b = new BabyAccount();
b.PayInFunds(50);
b.WithdrawFunds(5);
```

The call of **PayInFunds** will use the method in the parent (since that has not been overridden) but the call of **WithdrawFunds** will use the method in **BabyAccount**.

Virtual Methods

Actually, there is one other thing that we need to do in order for the overriding to work. The C# compiler needs to know if a method is going to be overridden. This is because it must call an overridden method in a slightly different way from a "normal" one. In other words, the above code won't compile properly because the compiler has not been told that **WithDrawFunds** might be overridden in classes which are children of the parent class.

To make the overriding work correctly I have to change my declaration of the method in the **CustomerAccount** class.

```
public class CustomerAccount : IAccount {

   private decimal balance = 0;

   public virtual bool WithdrawFunds ( decimal amount )
   {
      if ( balance < amount )
      {
         return false ;
      }
      balance = balance - amount ;
      return true;
   }

   public void PayInFunds ( decimal amount )
   {
      balance = balance + amount ;
   }

   public decimal GetBalance ()
   {
      return balance;
   }
}
```

The keyword **virtual** means “I might want to make another version of this method in a child class”. You don’t have to override the method, but if you don’t have the word present, you definitely can’t.

This makes **override** and **virtual** a kind of matched pair. You use **virtual** to mark a method as able to be overridden and **override** to actually provide a replacement for the method.

Protection of data in class hierarchies

It turns out that the code above still won’t work. This is because the balance value in the **CustomerAccount** class is **private**. We carefully made it **private** so that methods in other classes can’t get hold of the value and change it directly.

However, this protection is too strict, in that it stops the **BabyAccount** class from being able to change the value. To get around this problem C# provides a slightly less restrictive access level called **protected**. This makes the member visible to classes which extend the parent. In other words, methods in the **BabyAccount** class can see and use a protected member because they are in the same *class hierarchy* as the class containing the member.

A class hierarchy is a bit like a family tree. Every class has a parent and can do all the things that the parent can do. It also has access to all the protected members of the classes above it.

```
public class CustomerAccount : IAccount {

   protected decimal balance = 0;

   .....
}
```

I'm not terribly happy about doing this, the **balance** is very important to me and I'd rather that nobody outside the **CustomerAccount** class could see it. However, for now making this change will make the program work. Later we will see better ways to manage this situation.

Bank Notes: Overriding for Fun and Profit

The ability to override a method is very powerful. It means that we can make more general classes (for example the **CustomerAccount**) and customise it to make them more specific (for example the **BabyAccount**). Of course this should be planned and managed at the design stage. This calls for more *metadata* to be gathered from the customer and used to decide which parts of the behaviour need to be changed during the lift of the project. We would have made the **WithDrawFunds** method **virtual** because the manager would have said "We like to be able to customise the way that some accounts withdraw funds". And we would have written this down in the specification.

Using the base method

Remember that programmers are essentially lazy people who try to write code only once for a given problem. Well, it looks as if we are breaking our own rules here, in that the **WithDrawFunds** method in the **BabyAccount** class contains all the code of the method in the parent class.

We have already noted that we don't like this much, in that it means that the balance value has to be made more exposed that we might like. Fortunately the designers of C# have thought of this and have provided a way that you can call the *base* method from one which overrides it.

The word base in this context means "a reference to the thing which has been overridden". I can use this to make the **WithDrawFunds** method in my **BabyAccount** much simpler:

```
public class BabyAccount : CustomerAccount,IAccount {

   public override bool WithdrawFunds (decimal amount)
   {
      if (amount > 10)
      {
         return false ;
      }
      return base.WithdrawFunds(amount);
   }
}
```

The very last line of the **WithDrawFunds** method makes a call to the original **WithDrawFunds** method in the parent class, i.e. the one that the method overrides.

It is important that you understand what I'm doing here, and why I'm doing it:

- I don't want to have to write the same code twice

- I don't want to make the **balance** value visible outside the **CustomerAccount** class.

The use of the word **base** to call the overridden method solves both of these problems rather beautifully. Because the method call returns a **bool** result I can just send whatever it delivers. By making this change I can put the **balance** back to **private** in the **CustomerAccount** because it is not changed outside it.

Note that there are other useful spin-offs here. If I need to fix a bug in the behaviour of the **WithDrawFunds** method I just fix it once, in the top level class, and then it is fixed for all the classes which call back to it.

Making a Replacement Method

This bit is rather painful, but don't worry too much since it actually does make sense when you think about it. If you play around with C# you will find out that you don't actually seem to need the **virtual** keyword to override a method. If I leave it out (and leave out the **override** too) the program seems to work fine.

This is because in this situation there is no overriding, you have just supplied a new version of the method (in fact the C# compiler will give you a warning which indicates that you should provide the keyword **new** to indicate this):

```
public class BabyAccount : CustomerAccount,IAccount {

   public new bool WithdrawFunds (decimal amount)
   {
      if (amount > 10)
      {
         return false ;
      }
      if (balance < amount)
      {
         return false ;
      }
      balance = balance - amount ;
      return true;
   }
}
```

The problem with this way of working is that you are unable to use **base**. This makes it more difficult to pick up behaviours from parent classes.

Programmer's Point:Don't Replace Methods

I am very against replacing methods rather than overriding them. If you want to have a policy of allowing programmers to make custom versions of classes in this way it is much more sensible to make use of overriding since this allows a well managed way of using the method that you over-rid. In fact, I'm wondering why I mentioned this at all..

Stopping Overriding

Overriding is very powerful. It means that a programmer can just change one tiny part of a class and make a new one with all the behaviours of the parent. This goes well with a design process which means that as you move down the "family tree" of classes you get more and more specific.

However, overriding/replacing is not always desirable. Consider the **GetBalance** method. This is never going to need a replacement. And yet a naughty programmer could write their own and override or replace the one in the parent:

```
public new decimal GetBalance ()
{
    return 1000000;
}
```

This is the banking equivalent of the bottle of beer that is never empty. No matter how much cash is drawn out, it always returns a balance value of a million pounds!

A naughty programmer could insert this into a class and give himself a nice spending spree. What this means is that we need a way to mark some methods as not being able to be overridden. C# does this by giving us a **sealed** keyword which means "You can't override this method any more".

Unfortunately this is rather hard to use. The rules are that you can only seal an overriding method (which means that we can't seal the **GetBalance** virtual method in the **CustomerAccount** class) and you can still replace a sealed method.

Another use for **sealed**, which has a bit more potential, is that you can mark a class as sealed. This means that the class cannot be extended, i.e. it cannot be used as the basis for another class.

```
public sealed class BabyAccount : CustomerAccount,IAccount {

    .....

}
```

The compiler will now stop the **BabyAccount** from being used as the basis of another account.

Bank Notes: Protect Your Code

As far as the bank application is concerned, the customer will not have particularly strong opinions on how you use things like **sealed** in your programs. But they will want to have confidence in the code that you make. One of the unfortunate things about this business is that you will have to allow for the fact that people who use your components might not all be nice and trustworthy. This means that you should take steps when you design the program to decide whether or not methods should be flagged as virtual and also make sure that you seal things when you can do so.

For a programming course at this level it is probably a bit heavy handed of me to labour this point just right now, and if it didn't all make sense there is no particular need to worry, just remember that when you create a program this is another risk that you will have to consider.

Constructors and Hierarchies

A constructor is a method which gets control during the process of object creation. It is used by a programmer to allow initial values to be set into an object:

```
robsAccount = new CustomerAccount("Rob Miles","Hull");
```

The code above will only work if the **CustomerAccount** class has a constructor which accepts two strings, the name and the address of the new customer.

You might think that I could solve this by writing a constructor a bit like this:

```
public CustomerAccount (string inName,decimal inBalance)
{
    name = inName;
    balance = inBalance;
}
```

But this class is an extension of the **Account** class. In other words, to make a **CustomerAccount** I have to make an **Account**. And the account is the class which will have a constructor which sets the name and the initial balance. In this situation the constructor in the child class will have to call a particular constructor in the parent to set that up before it is created. The keyword base is used to make a call to the parent constructor. In other words, the proper version of the customer account constructor is as follows:

```
public CustomerAccount (string inName,decimal inBalance) :
base ( inName, inBalance)
{
}
```

The base keyword is used in the same way as this is used to call another constructor in the same class. The constructor above assumes that the Account class which CustomerAccount is a child of has a constructor which accepts two parameters, the first a string and the second a decimal value.

Constructor Chaining

When considering constructors and class hierarchies you must therefore remember that to create an instance of a child class an instance of the parent must first be created. This means that a constructor in the parent must run before the constructor in the child. In other words, to create a **CustomerAccount** you must first create an **Account**. The result of this is that programmers must take care of the issue of *constructor chaining*. They must make sure that at each level in the creation process a constructor is called to set up the class at that level.

Programmer's Point:Design your class construction process

The means by which your class instances are created is something you should design into the system that you build. It is part of the overall architecture of the system that you are building. I think of these things as a bit like the girders that you erect to hold the floors and roof of a large building. They tell programmers who are going to build the components which are going to implement the solution how to create those components. It is of course very important that you have these designs written down and readily available to the development team.

Abstract methods and classes

At the moment we are using overriding to modify the behaviour of an existing parent method. However, it is also possible to use overriding in a slightly different context. I can use it to force a set of behaviours on items in a class hierarchy. If there are some things that an account must do then we can make these *abstract* and then get the child classes to actually provide the implementation.

For example, in the context of the back application we might want to provide a method which creates a warning letter to the customer that their account is overdrawn. This will have to be different for each type of account (we don't want to use the same language to a baby account holder as we do for an older one). This means that at the time we create the bank account system we know that we need this method, but we don't know precisely what it does in every situation.

We could just provide a "standard" method in the **CustomerAccount** class and then rely on the programmers overriding this with a more specific message but we then have no way of making sure that they really do provide the method.

C# provides a way of flagging a method as **abstract**. This means that the method body is not provided in this class, but will be provided in a child class:

```
public abstract class Account
{
    public abstract string RudeLetterString();
}
```

The fact that my new **Account** class contains an **abstract** method means that the class itself is **abstract** (and must be marked as such). It is not possible to make an instance of an abstract class. If you think about it this is sensible. An instance of **Account** would not know what to do it the **RudeLetterString** method was ever called.

An abstract class can be thought of as a kind of template. If you want to make an instance of a class based on an abstract parent you must provide implementations of all the abstract methods given in the parent.

Abstract classes and interfaces

You might decide that an abstract class looks a lot like an interface. This is true, in that an interface also provides a "shopping list" of methods which must be provided by a class. However, abstract classes are different in that they can contain fully implemented methods alongside the abstract ones. This can be useful because it means you don't have to repeatedly implement the same methods in each of the components that implement a particular interface.

The problem is that you can only inherit from one parent, so you can only pick up the behaviours of one class. If you want to implement interfaces as well, you may have to repeat methods as well.

Perhaps at this point a more fully worked example might help.

If we consider our bank account problem we can identify two types of behaviour:

- those which every type of bank account must provide (for example **PayInFunds** and **GetBalance**)
- those which each type of bank account must provide in a way specific to that particular account type (for example **WithdrawFunds** and **RudeLetterString**)

The trick is to take all the methods in the first category and put them inside the parent class. The methods in the second category must be made abstract. This leads us to a class design a bit like this:

```
public interface IAccount
{
   void PayInFunds ( decimal amount );
   bool WithdrawFunds ( decimal amount );
   decimal GetBalance ();
   string RudeLetterString();
}
public abstract class Account : IAccount
{
   private decimal balance = 0;

   public abstract string RudeLetterString();

   public virtual bool WithdrawFunds ( decimal amount )
   {
      if ( balance < amount )
      {
         return false ;
      }
      balance = balance - amount ;
      return true;
   }

   public virtual decimal GetBalance ()
   {
      return balance;
   }

   public void PayInFunds ( decimal amount )
   {
      balance = balance + amount ;
   }
}

public class CustomerAccount : Account
{
   public override string RudeLetterString()
   {
      return "You are overdrawn" ;
   }
}

public class BabyAccount : Account
{
   public override bool WithdrawFunds ( decimal amount )
   {
      if (amount > 10)
      {
         return false ;
      }
      return base.WithdrawFunds(amount);
   }
   public override string RudeLetterString()
   {
      return "Tell daddy you are overdrawn";
   }
}
```

This code repays careful study. Note how I have moved all the things that all accounts must do into the parent **`Account`** class. Then I have added customised methods into the child classes where appropriate. Note also though that I have left the interface in place. That is because; even though I now have this abstract structure I still want to

think of the account objects in terms of their "accountness" rather than any particular specific type.

References to abstract classes

References to abstract classes work just like references to interfaces. A reference to an **`Account`** class can refer to any class which extends from that parent. This might seem useful, as we can consider something as an "account" rather than a **`BabyAccount`**.

However, I much prefer it if you mange references to abstract things (like accounts) in terms of their interface instead.

Bank Notes: Designing with interface and abstract

For the purpose of this part of the course you now have broad knowledge of all the tools that can be used to design large software systems. If you have an understanding of what an interface and abstract classes are intended to achieve this will stand you in very good stead for your programming career. Broadly:

Interface: lets you identify a set of behaviours (i.e. methods) which a component can be made to implement. Any component which implements the interface can be thought of in terms of a reference of that interface type. A concrete example of this would be something like **`IPrintHardCopy`**. Lots of items in my bank system will need to do this and so we could put the behaviour details into the interface for them to implement in their own way. Then our printer can just regard each of the instances that implement this interface purely in this way. *Interfaces let me describe a set of behaviours which a component can implement. Once a component can implement an interface it can be regarded purely in terms of a component with this ability.* Objects can implement more than one interface, allowing them to present different faces to the systems that use them.

Abstract: lets you create a parent class which holds template information for all the classes which extend it. If you want to create a related set of items, for example bank account, receipt, invoice, person then the best way to do this is to set up a parent class which contains abstract and non-abstract methods. The child classes can make use of the methods from the parent and override the ones that need to be provided differently for that particular class.

One important consideration though, is that even if you make use of an abstract parent class I reckon that you should still make use of interfaces to reference the accounts. And the reason why goes like this:

If our bank takes over another bank and wants to share account information we might need a way to use their accounts. If their accounts are software components too (and they should be) then all we have to do is implement the required interfaces at each end and then our systems understand each other. In other words the other bank must create the methods in the **`IAccount`** interface, get their account objects (whatever they are called) to implement the interface and, hey presto, I can now use their accounts.

This would be much more difficult if my entire system thought in terms of a parent Account class – since their classes would not fit into this hierarchy at all.

Don't Panic

This is all deep stuff. If you don't get it now, don't worry. These features of C# are tied up with the process of software design which is a very complex business. The important point to bear in mind is that the features are all provided so that you can solve one problem: