CHAPTER: 05

Methods

Objectives:

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

- *explain the meaning of the term* ***method****;*
- *declare and define methods;*
- ***call*** *a method;*
- *explain the meaning of the terms* ***actual parameters*** *and* ***formal parameters****;*
- *devise simple* ***algorithms*** *with the help of pseudocode;*
- *identify the* ***scope*** *of a particular variable;*
- *explain the meaning of the term* ***polymorphism****;*
- *declare and use* ***overloaded*** *methods.*

5.1 Introduction

As early as chapter 1 we were using the term **method**. There you found out that a method is a part of a class, and contains a particular set of instructions. So far, all the classes you have written have contained just one method, the `main` method. In this chapter you will see how a class can contain not just a `main` method, but many other methods as well.

Normally a method will perform a single well-defined task. Examples of the many sorts of task that a method could perform are calculating the area of a circle, displaying a particular message on the screen, converting a temperature from Fahrenheit to Celsius, and many many more. In this chapter you will see how we can collect the instructions for performing these sorts of tasks together in a method.

You will also see how, once we have written a method, we can get it to perform its task within a program. When we do this we say that we are **calling** the method. When we call a method, what we are actually doing is telling the program to jump to a new place (where the method instructions are stored), carry out the set of instructions that it finds there, and, when it has finished (that is, when the method has terminated), return and carry on where it left off.

So in this chapter you will learn how to write a method within a program, how to call a method from another part of the program and how to send information into a method and get information back.

5.2 Declaring and defining methods

Let's illustrate the idea of a method by thinking about a simple little program. The program prompts the user to enter his or her year of birth, month of birth and day of birth; each time the prompt is displayed, it is followed by a message, consisting of a couple of lines, explaining that the information entered is confidential. This is shown below in program 5.1 – the program would obviously then go on to do other things with the information that has been entered, but we are not interested in that, so we have just replaced all the rest of the program with a comment.

Program 5.1

```
import java.util.*;

public class DataEntry
{
   public static void main(String[] args)
   {
      Scanner keyboard = new Scanner(System.in);

      int year, month, day;

      // prompt for year of birth
      System.out.println("Please enter the year of your birth");

      // display confidentiality message
      System.out.println("Please note that all information supplied is confidential");
      System.out.println("No personal details will be shared with any third party");

      // get year from user
      year = keyboard.nextInt();

      // prompt for month of birth
      System.out.println("Please enter the month of your birth as a number from 1 to 12");

      // display confidentiality message
      System.out.println("Please note that all information supplied is confidential");
      System.out.println("No personal details will be shared with any third party");

      // get month from user
      month = keyboard.nextInt();

      // prompt for day of birth
      System.out.println("Please enter the day of your birth as a number from 1 to 31");

      // display confidentiality message
      System.out.println("Please note that all information supplied is confidential");
      System.out.println("No personal details will be shared with any third party");

      // get day from user
      day = keyboard.nextInt();

      // more code here
   }
}
```

You can see from the above program that we have had to type out the two lines that display the confidentiality message three times. It would be far less time-consuming if we could do this just once, then send the program off to wherever these instructions are stored, and then come back and carry on with what it was doing. You will probably have realized by now that we can indeed do this – by writing a *method*. The job of this particular method will be simply to display the confidentiality message on the screen – we need to give our method a name so that we can refer to it when required, so let's call it `displayMessage`. Here is how it is going to look:

```
static void displayMessage()
{
  System.out.println("Please note that all information supplied is confidential");
  System.out.println("No personal details will be shared with any third party");
}
```

The body of this method, which is contained between the two curly brackets, contains the instructions that we want this method to perform, namely to display two lines of text on the screen. The first line, which declares the method, is called the method **header**, and consists of three words – let's look into each of these a bit more closely:

`static`

You have seen this word in front of the `main` method many times now. However, we won't be explaining its meaning to you properly until chapter 8. For now, all you need to know is that methods that have been declared as **`static`** (such as `main`) can only call other methods in the class if they too are **`static`**. So, if we did not declare `displayMessage` as **`static`** and tried to call it from the `main` method then our program would not compile.

`void`

In the next section you will see that it is possible for a method to *send back* or *return* some information once it terminates. This particular method simply displays a message on the screen, so we don't require it to send back any information when it terminates. The word **`void`** indicates that the method does not return any information.

`displayMessage()`

This is the name that we have chosen to give our method. You can see that the name is followed by a pair of empty brackets. Very soon you will learn that it is possible to send some information *into* a method – for example, some values that the method needs in order to perform a calculation. When we need to do that we list, in these brackets, the types of data that we are going to send in; here, however, as the method is doing nothing more that displaying a message on the screen we do not have to send in any data, and the brackets are left empty.

5.3 Calling a method

Now that we have declared and defined our method, we can make use of it. The idea is that we get the method to perform its instructions as and when we need it to do so – you have seen that this process is referred to as *calling* the method. To call a method in Java, we simply use its name, along with the following brackets, which in this case are empty. So in this case our method call, which will be placed at the point in the program where we need it, looks like this:

```
displayMessage();
```

Now we can rewrite program 5.1, replacing the appropriate lines of code with the simple method call. The whole program is shown below in program 5.2:

Program 5.2

```
import java.util.*;

public class DataEntry2
{
   public static void main(String[] args)
   {
      Scanner keyboard = new Scanner(System.in);

      int year, month, day;

      System.out.println("Please enter the year of your birth");
      displayMessage(); // call displayMessage method
      year = keyboard.nextInt();

      System.out.println("Please enter the month of your birth as a number from 1 to 12");
      displayMessage(); // call displayMessage method
      month = keyboard.nextInt();

      System.out.println("Please enter the day of your birth as a number from 1 to 31");
      displayMessage(); // call displayMessage method
      day = keyboard.nextInt();

      // more code here
   }

   // the code for displayMessage method
   static void displayMessage()
   {
      System.out.println("Please note that all information supplied is confidential");
      System.out.println("No personal details will be shared with any third party");
   }
}
```

You can see that the method itself is defined separately after the `main` method – although it could have come before it, since the order in which methods are presented doesn't matter to the compiler. When the program is run, however, it always starts with `main`. The overall result is of course that program 5.2 runs exactly the same as program 5.1.

We should emphasize again here that when one method calls another method, the first method effectively pauses at that point, and the program then carries out the instructions in the called method; when it has finished doing this, it returns to the original method, which then resumes. In most of the programs in this chapter it will be the `main` method that calls the other method. This doesn't have to be the case, however, and it is perfectly possible for any method to call another method – indeed, the called method could in turn call yet another method. This would result in a number of methods being "chained". When each method terminates, the control of the program would return to the method that called it.

You can see an example of a method being called by a method other than `main` in section 5.6.

5.4 Method input and output

We have already told you that it is possible to send some data into a method, and that a method can send data back to the method that called it. Now we will look into this in more detail.

In order to do this we will use as an example a program that we wrote in chapter 2 – program 2.3. Here is a reminder of that program:

A reminder of program 2.3

```
import java.util.*;

/* a program to input the initial price of a product and then calculate and display its cost after tax
   has been added */

public class FindCost3
{
   public static void main(String[] args)
   {
      Scanner keyboard = new Scanner(System.in);
      double price, tax;
      System.out.println("*** Product Price Check ***");
      System.out.print("Enter initial price: ");
      price = keyboard.nextDouble();
      System.out.print("Enter tax rate: ");
      tax = keyboard.nextDouble();
      price = price * (1 + tax/100);
      System.out.println("Cost after tax = " + price);
   }
}
```

The line that calculates the new price, with the sales tax added, is this one:

```
price = price * (1 + tax/100);
```

Let's create a method that performs this calculation – in a real application this would be very useful, because we might need to call this method at various points within the program, and, as you will see, each time we do so we could get it to carry out the calculation for different values of the price and the tax. We will need a way to send in these values to the method. But on top of that, we need to arrange for the method to tell us the result of adding the new tax – if it didn't do that, it wouldn't be much use!

The method is going to look like this:

```
static double addTax(double priceIn, double taxIn)
{
  return priceIn * (1 + taxIn/100);
}
```

First, take a careful look at the header. You are familiar with the first word, **`static`**, but look at the next one; this time, where we previously saw the word **`void`**, we now have the word **`double`**. As we have said, this method must send back – or **return** – a result, the new price of the item. So the type of data that the method is to return in this case is a **`double`.** In fact what we are doing here is declaring a method of *type* **`double`**. Thus, the *type* of a method refers to its *return* type. It is possible to declare methods of any type – **`int`**, **`boolean`**, **`char`** and so on.

After the type declaration, we have the name of the method, in this case `addTax` – and this time the brackets aren't empty. You can see that within these brackets we are declaring two variables, both of type

`double`. The variables declared in this way are known as the **formal parameters** of the method. Formal parameters are variables that are created exclusively to hold values sent in from the calling method. They are going to hold, respectively, the values of the price and the tax that are going to be sent in from the calling method (you will see how this is done in a moment). Of course, these variables could be given any name we choose, but we have called them `priceIn` and `taxIn` respectively. We will use this convention of adding the suffix `In` to variable names in the formal parameter list throughout this book.

Now we can turn our attention to the body of the method, which as you can see, in this case, consists of a single line:

```
return priceIn * (1 + taxIn/100);
```

The word **`return`** in a method serves two very important functions. First it ends the method – as soon as the program encounters this word, the method terminates, and control of the program jumps back to the calling method. The second function is that it sends back a value. In this case it sends back the result of the calculation:

```
priceIn * (1 + taxIn/100)
```

You should note that if the method is of type **`void`**, then there is no need to include a **`return`** instruction – the method simply terminates once the last instruction is executed.

Now we can discuss how we actually call this method and use its return value. The whole program appears below as program 5.3:

Program 5.3

```
import java.util.*;

/* we have adapted program 2.3 so that the new price is determined by calling a method that adds the
   sales tax */

public class FindCost4
{
  public static void main(String[] args )
  {
      Scanner keyboard = new Scanner(System.in);

      double price, tax;

      System.out.println("*** Product Price Check ***");

      System.out.print("Enter initial price: ");
      price = keyboard.nextDouble();

      System.out.print("Enter tax rate: ");
      tax = keyboard.nextDouble();

      price = addTax(price, tax); // call the addTax method

      System.out.println("Cost after tax = " + price);
  }

  static double addTax(double priceIn, double taxIn)
  {
    return priceIn * (1 + taxIn/100);
   }
}
```

The line in `main` that calls the method is this one:

```
price = addTax(price, tax);
```

First, we will consider the items in brackets after the method name. As you might have expected, there are two items in the brackets – these are the *actual* values that we are sending into our method. They are therefore referred to as the **actual parameters** of the method. Their values are copied onto the formal parameters in the called method. This process, which is referred to as **passing** parameters, is illustrated in figure 5.1.

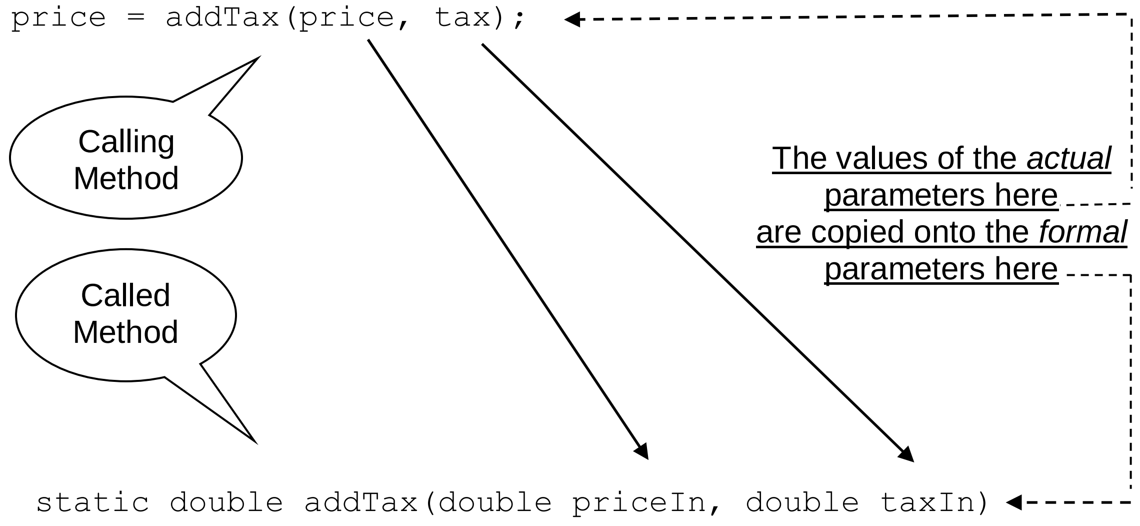
price = addTax(price, tax);
Calling Method
Called Method
The values of the actual parameters here are copied onto the formal parameters here
static double addTax(double priceIn, double taxIn)

Fig 5.1. Passing parameters from one method to another method

You might have been wondering how the program knows which values in the actual parameter list are copied onto which variables in the formal parameter list. The answer to this is that it is the *order* that is important – you can see this from figure 5.1 – the value of `price` is copied onto `priceIn`; the value of `tax` is copied onto `taxIn`. Although the variable names have been conveniently chosen, the names themselves have nothing to do with which value is copied to which variable.

You might also be wondering what would happen if you tried to call the method with the wrong number of variables. For example:

```
price = addTax(price);
```

The answer is that you would get a compiler error, because there is no method called `addTax` that requires just one single variable to be passed into it.

You can see then that the actual parameter list must match the formal parameter list exactly. Now this is important not just in terms of the number of variables, but also in terms of the *types*. For example, using actual values this time instead of variable names, this method call would be perfectly acceptable:

```
price = addTax(187.65, 17.5);
```

However, this would cause a compiler error:

```
price = addTax(187.65, 'c');
```

The reason, of course, is that `addTax` requires two **`double`**s, not a **`double`** and a **`char`**.

We can now move on to looking at how we make use of the return value of a method.

The `addTax` method returns the result that we are interested in, namely the new price of the item. What we need to do is to assign this value to the variable `price`. As you have already seen we have done this in the same line in which we called the method:

```
price = addTax(price, tax);
```

A method that returns a value can in fact be used just as if it were a variable of the same type as the return value! Here we have used it in an assignment statement – but in fact we could have simply dropped it into the `println` statement, just as we would have done with a simple variable of type **`double`**:

```
System.out.println("Cost after tax = " + addTax(price, tax));
```

5.5 More examples of methods

Just to make sure you have got the idea, let's define a few more methods. To start with, we will create a very simple method, one that calculates the square of a number. When we are going to write a method, there are four things to consider:

- the name that we will give to the method;
- the inputs to the method (the formal parameters);
- the output of the method (the return value);
- the body of the method (the instructions that do the job required).

In the case of the method in question, the name `square` would seem like a sensible choice. We will define the method so that it will calculate the square of any number, so it should accept a single value of type **`double`**. Similarly, it will return a **`double`**.

The instructions will be very simple – just return the result of multiplying the number by itself. So here is our method:

```
static double square(double numberIn)
{
   return numberIn * numberIn;
}
```

Remember that we can choose any names we want for the input parameters; here we have stuck with our convention of using the suffix `In` for formal parameters.

To use this method in another part of the program, such as the `main` method, is now very easy. Say, for example, we had declared and initialized two variables as follows:

```
double a = 2.5;
double b = 9.0;
```

Let's say we wanted to assign the square of `a` to a **`double`** variable `x` and the square of `b` to a **`double`** variable `y`. We could do this as follows:

```
x = square(a);
y = square(b);
```

After these instructions, `x` would hold the value 6.25 and `y` would hold the value 81.0.

For our next illustration we will choose a slightly more complicated example. We will define a function that we will call `max`; it will accept two integer values, and will return the bigger value of the two (of course, if they are equal, it can return the value of either one). It should be pretty clear that we will require two integer parameters, and that the method will return an integer. As far as the instructions are concerned, it should be clear that an **`if`**…**`else`** statement should do the job – if the first number is greater than the second, return the first number, if not return the second. Here is our method:

```
static int max(int firstIn, int secondIn)
{
  if(firstIn > secondIn)
  {
    return firstIn;
  }
  else
  {
    return secondIn;
  }
}
```

You should note that in this example we have two **`return`** statements, each potentially returning a different value – the value that is actually returned is decided at run-time by the values of the variables `firstIn` and `secondIn`. remember, as soon as a **`return`** statement is reached the method terminates.

Working out how to write the instructions for this method was not too hard a job. In fact it was so simple, we didn't bother to design it with pseudocode. However, there will be many occasions in the future when the method has to carry out a much more complex task, and you will need to think through how to perform this task. A set of instructions for performing a job is known as an **algorithm** – common examples of algorithms in everyday life are recipes and DIY (Do-It-Yourself) instructions. Much of a programmer's time is spent devising algorithms for particular tasks, and, as you saw in chapter 2, we can use pseudocode to help us design our algorithms. We will look at further examples as we progress through this chapter.

Let's develop one more method. There are many instances in our programming lives where we might need to test whether a number is even or odd. Let's provide a method that does this job for us. We will call our method `isEven`, and it will report on whether or not a particular number is an even number. The test will be performed on integers, so we will need a single parameter of type **`int`**. The return value is interesting – the method will tell us whether or not a number is even, so it will need to return a value of **`true`** if the number is even or **`false`** if it is not. So our return type is going to be **`boolean`**. The instructions are quite simple to devise – again, an **`if`**…**`else`** statement should certainly do the job. But how can we test whether a number is even or not? Well, an even number will give a remainder of zero when divided by 2. An odd number will not. So we can use the modulus operator here. Here is our method:

```
static boolean isEven(int numberIn)
{
  if(numberIn % 2 == 0)
  {
    return true;
  }
  else
  {
    return false;
  }
}
```

Actually there is a slightly neater way we could have written this method. The expression:

```
numberIn % 2 == 0
```

will evaluate to either **`true`** or **`false`** – and we could therefore simply have returned the value of this expression and written our method like this:

```
static boolean isEven(int numberIn)
{
  return (numberIn % 2 == 0);
}
```

It is interesting to note that the calling method couldn't care less how the called method is coded – all it needs is for it to do the calculation correctly, and return the desired value. This is something that will become very significant when we look at methods that call methods of other classes later in this semester.

A method that returns a **`boolean`** value can be referred to as a **`boolean`** method. In chapter 3 you came across **`boolean`** *expressions* (expressions that evaluate to **`true`** or **`false`**) such as:

```
temperature > 10
```

or

```
y == 180
```

Because **`boolean`** methods also evaluate to **`true`** or **`false`** (that is, they return a value of **`true`** or **`false`**) they can - as with boolean expressions - be used as the test in a selection or loop.

For example, assuming that the variable `number` had been declared as an **`int`**, we could write something like:

```
if(isEven(number))
{
   // code here
}
```

or

```
while(isEven(number))
{
   // code here
}
```

To test for a **`false`** value we simply negate the expression with the *not* operator (!):

```
if(!isEven(number))
{
   // code here
}
```

Before we leave this section, there is one thing we should make absolutely clear – a method cannot change the *original* value of a variable that was passed to it as a parameter. The reason for this is that all that is being passed to the method is a *copy* of whatever this variable contains. In other words, just a *value.* The method does not have access to the original variable. Whatever value is passed is copied to the parameter in the called method. We will illustrate this with a very simple program indeed – program 5.4 below:

Program 5.4

```
public class ParameterDemo
{
  public static void main(String[] args)
  {
    int x = 10;
    demoMethod(x);
    System.out.println(x);
  }
  static void demoMethod(int xIn)
  {
    xIn = 25;
    System.out.println(xIn);
  }
}
```

You can see that in the `main` method we declare an integer, `x`, which is initialized to 10. We then call a method called `demoMethod`, with `x` as a parameter. The formal parameter of this method – `xIn` – will now of course hold the value 10. But the method then assigns the value of 25 to the parameter – it then displays the value on the screen.

The method ends there, and control returns to the `main` method. The final line of this method displays the value of `x`.

The output from this method is as follows:

```
25

10
```

This shows that the original value of `x` has not in any way been affected by what happened to `xIn` in `demoMethod`.

5.6 Variable scope

Looking back at program 5.3, it is possible that some of you asked yourselves the following questions: Why do we need to bother with all this stuff in the brackets? We've already declared a couple of variables called `price` and `tax` – why can't we just use them in the body of the method? Well, go ahead and try it – you will see that you get a compiler error telling you that these variables are not recognized!

How can this be? They have certainly been declared. The answer lies in the matter of *where* exactly these variables have been declared. In actual fact variables are only "visible" within the pair of curly brackets in which they have been declared – this means that if they are referred to in a part of the program outside these brackets, then you will get a compiler error. Variables that have been declared inside the brackets of a particular method are called **local** variables – so the variables `price` and `tax` are said to be *local* to the `main` method. We say that variables have a **scope** – this means that their visibility is limited to a particular part of the program. If `price` or `tax` were referred to in the `addtax` method, they would be out of scope.

Let's take another, rather simple, example. Look at program 5.5:

Program 5.5

```
public class ScopeTest
{
  public static void main(String[] args)
  {
    int x = 1;      // x is local to main
    int y = 2;     // y is local to main
    method1(x, y); // call method1
  }

  static void method1(int xIn, int yIn)
  {
    int z; // z is local to method1
    z = xIn + yIn;
    System.out.println(z);
  }
}
```

In this program the variables `x` and `y` are local to `main`. The variable `z` is local to `method1`. The variables `xIn` and `yIn` are the formal parameters of `method1`. This program will compile and run without a problem, because none of the variables is referred to in the wrong place.

Imagine, however, that we were to rewrite program 5.5 as program 5.6, below:

Program 5.6

```
// this program will give rise to two compiler errors

public class ScopeTest2
{
  public static void main(String[] args)
  {
    int x = 1; // x is local to main
    int y = 2; // y is local to main
    method1(x, y); // call method1
    System.out.println(z); // this line will cause a compiler error as z is local to method1
  }

  static void method1(int xIn, int yIn)
  {
    int z;             // z is local to method1
    z = x + y;         // this line will cause a compiler error as x and y are local to main
    System.out.println(z);
  }
}
```

As the comments indicate, the lines in bold will give rise to compiler errors, as the variables referred to are out of scope.

It is interesting to note that, since a method is completely unaware of what has been declared inside any other method, you could declare variables with the same name inside different methods. The compiler would regard each variable as being completely different from any other variable in another method which simply had the same name. So, for example, if we had declared a *local* variable called `x` in `method1`, this would be perfectly ok – it would exist completely independently from the variable named `x` in `main`.

To understand why this is so, it helps to know a little about what goes on when the program is running. A part of the computer's memory called the stack is reserved for use by running programs. When a method is called, some space on the stack is used to store the values for that method's formal parameters and its local variables. That is why, whatever names we give them, they are local to their particular method. Once the method terminates, this part of the stack is no longer accessible, and the variables effectively no longer exist. And this might help you to understand even more clearly why the value of a variable passed as a parameter to a method cannot be changed by that method.

Before we move on, it will be helpful if we list the kinds of variables that a method can access:

- a method can access variables that have been declared as formal parameters;
- a method can access variables that have been declared locally – in other words that have been declared within the curly brackets of the method;
- as you will learn in chapter 8, a method has access to variables declared as *attributes of the class* (don't worry – you will understand what this means in good time!).

A method cannot access any other variables.

5.7 Method overloading

You have already encountered the term *overloading* in previous chapters, in connection with operators. You found out, for example, that the division operator (/) can be used for two distinct purposes – for division of integers, and for division of real numbers. The + operator, is not only used for addition, but also for concatenating two strings. So the same operator can behave differently depending on what it is operating on – operators can be overloaded.

Methods too can be overloaded. To illustrate, let's return to the `max` method of section 5.5. Here it is again:

```
static int max(int firstIn, int secondIn)
{
  if(firstIn > secondIn)
  {
    return firstIn;
  }
  else
  {
    return secondIn;
  }
}
```

As you will recall, this method accepts two integers and returns the greater of the two. But what if we wanted to find the greatest of three integers? We would have to write a new method, which we have shown below. We are just showing you the header here – we will think about the actual instructions in a moment:

```
static int max(int firstIn, int secondIn, int thirdIn)
{
    // code goes here
}
```

You can see that we have given this method the same name as before – but this time it has *three* parameters instead of two. And the really clever thing is that we can declare and call both methods within the same class. Both methods have the same name but the parameter list is different – and each one will *behave* differently. In our example, the original method compares two integers and returns the greater of the two; the second one, once we have worked out the algorithm for doing this, will examine three integers and return the value of the one that is the greatest of the three. When two or more methods, distinguished by their parameter lists, have the same name but perform different functions we say that they are **overloaded**. Method overloading is actually one example of what is known as **polymorphism**. Polymorphism literally means *having many forms*, and it is an important feature of object-oriented programming languages. It refers, in general, to the phenomenon of having methods and operators with the same name performing different functions. You will come across other examples of polymorphism in later chapters.

Now, you might be asking yourself how, when we call an overloaded method, the program knows which one we mean. The answer of course depends on the actual parameters that accompany the method call – they are matched with the formal parameter list, and the appropriate method will be called. So, if we made this call somewhere in a program:

```
int x = max(3, 5);
```

then the first version of `max` would be called – the version that returns the bigger of two integers. This, of course, is because the method is being called with two integer parameters, matching this header:

```
static int max(int firstIn, int secondIn)
```

However, if this call, with three integer parameters, were made:

```
int x = max(3, 5, 10);
```

then it would be the second version that was called:

```
static int max(int firstIn, int secondIn, int thirdIn)
```

One very important thing we have still to do is to devise the *algorithm* for this second version. Can you think of a way to do it? Have go at it before reading on.

One way to do it is to declare an integer variable, which we could call `result`, and start off by assigning to it the value of the first number. Then we can consider the next number. Is it greater than the current value of `result`? If it is, then we should assign this value to `result` instead of the original value. Now we can consider the third number – if this is larger than the current value of `result`, we assign its value to `result`. You should be able to see that `result` will end up having the value of the greatest of the three integers. It is helpful to express this as pseudocode:

```
SET result TO first number
IF second number > result
BEGIN
 SET result TO second number
END
IF third number > result
BEGIN
```

```
 SET result TO third number
END
RETURN result
```

Here is the code:

```
static int max(int firstIn, int secondIn, int thirdIn)
{
  int result;
  result = firstIn;
  if(secondIn > result)
  {
    result = secondIn;
  }
  if(thirdIn > result)
  {
    result = thirdIn;
  }
  return result;
}
```

Program 5.7 illustrates how both versions of our `max` method can be used in the same program:

Program 5.7

```
public class OverloadingDemo
{
  public static void main(String[] args)
  {
    int maxOfTwo, maxOfThree;
    maxOfTwo = max(2, 10);      // call the first version of max
    maxOfThree = max(-5, 5, 3); // call the second version of max
    System.out.println(maxOfTwo);
    System.out.println(maxOfThree);
  }

  // this version of max accepts two integers and returns the greater of the two
  static int max(int firstIn, int secondIn)
  {
      if(firstIn > secondIn)
      {
         return firstIn;
      }
      else
      {
         return secondIn;
      }
   }

   // this version of max accepts three integers and returns the greatest of the three
   static int max(int firstIn, int secondIn, int thirdIn)
   {
      int result;
      result = firstIn;
      if(secondIn > result)
      {
         result = secondIn;
```

```
        }
        if(thirdIn > result)
        {
           result = thirdIn;
        }
        return result;
   }
}
```

As the first call to `max` in the `main` method has two parameters, it will call the first version of `max`; the second call, with its three parameters, will call the second version. Not surprisingly then the output from this program looks like this:

```
10

5
```

It might have occurred to you that we could have implemented the second version of `max` (that is the one that takes three parameters) in a different way. We could have started off by finding the maximum of the first two integers (using the first version of `max`), and then doing the same thing again, comparing the result of this with the third number.

This version is presented below – this is an example of how we can call a method not from the `main` method, but from another method.

```
static int max(int firstIn, int secondIn, int thirdIn)
{
   int step1, result;
   step1 = max(firstIn, secondIn); // call the first version of max
   result = max(step1, thirdIn); // call the first version of max again
   return result;
}
```

Some of you might be thinking that if we wanted similar methods to deal with lists of four, five, six, or even more numbers, it would be an awful lot of work to write a separate method for each one – and indeed it would! But don't worry – in the next chapter you will find that there is a much easier way to deal with situations like this.

5.7 Using methods in menu-driven programs

In chapter 4 we developed a program that presented the user with a menu of choices; we pointed out that this was a very common interface for programs before the days of graphics. Until we start working with graphics later in the semester, we will use this approach with some of our more complex programs.

Here is a reminder 4.10.

Program 4.10 - a reminder

```
   import java.util.*;
```

```
public class TimetableWithLoop
{
  public static void main(String[] args)
  {
      char group, response;
      Scanner sc = new Scanner (System.in);
      System.out.println("***Lab Times***");
      do // put code in loop
      {
         // offer menu of options
        System.out.println(); // create a blank line
        System.out.println("[1] TIME FOR GROUP A");
        System.out.println("[2] TIME FOR GROUP B");
        System.out.println("[3] TIME FOR GROUP C");
        System.out.println("[4] QUIT PROGRAM");
        System.out.print("enter choice [1,2,3,4]: ");
        response = sc.next().charAt(0); // get response
        System.out.println(); // create a blank line
        switch(response)  // process response
        {
          case '1': System.out.println("10.00 a.m ");
                    break;
          case '2': System.out.println("1.00 p.m ");
                    break;
          case '3': System.out.println("11.00 a.m ");
                    break;
          case '4': System.out.println("Goodbye ");
                    break;
          default:  System.out.println("Options 1-4 only!");
        }
      } while (response != '4'); // test for Quit option
    }
}
```

In this program, each **`case`** statement consisted of a single instruction (apart from the **`break`**), which simply displayed one line of text. Imagine, though, that we were to develop a more complex program in which each menu choice involved a lot of processing. The **`switch`** statement would start to get very messy, and the program could easily become very unwieldy. In this situation, confining each menu option to a particular method will make our program far more manageable.

Program 5.8 is an example of such a program. The program allows a user enter the radius of a circle and then enables the area and circumference of the circle to be calculated and displayed. Four menu options are offered. The first allows the user to enter the radius. The second displays the area of the circle, and the third the circumference. The final option allows the user to quit the program.

Study it carefully, and then we will point out some of the interesting features.

Program 5.8

```
import java.util.*;

/* This program demonstrates how methods can be used in a menu-driven program */

public class CircleCalculation
{
  public static void main(String[] args)
  {
     /* The variable below is local to the main method; if the value is needed by another method, it
        must be passed in as a parameter */

     Scanner keyboard = new Scanner(System.in);
     double radius = -999; // initialize with a dummy value to show that nothing has been entered
     char choice;
     do
     {
        System.out.println();
        System.out.println("*** CIRCLE CALCULATIONS ***");
        System.out.println();
        System.out.println("1. Enter the radius of the circle");
        System.out.println("2. Display the area of the circle");
        System.out.println("3. Display the circumference of the circle");
        System.out.println("4. Quit");
        System.out.println();
        System.out.println("Enter a number from 1 - 4");
        System.out.println();
        choice = keyboard.next().charAt(0);
        switch(choice)
        {
           case '1' : radius = option1(); // call method option1
                      break;
           case '2' : option2(radius); // call method option2
                      break;
           case '3' : option3(radius); // call method option3
                      break;
           case '4' : break;
           default : System.out.println("Enter only numbers from 1 - 4");
           System.out.println();
        }
     } while(choice != '4');
    }

    // option1 gets the user to enter the radius of the circle
    static double option1()
    {
        double myRadius; // local variable
        Scanner keyboard = new Scanner(System.in);
        System.out.print("Enter the radius of the circle: ");
        myRadius = keyboard.nextDouble();
        return myRadius;
    }

    // option2 calculates and displays the area of the circle
    static void option2(double radiusIn)
    {
        if(radiusIn == -999)
        {
           System.out.println("Radius has not been entered");
        }
        else
        {
           double area; // local variable
           area = 3.1416 * radiusIn * radiusIn; // calculate the area of the circle
           System.out.println("The area of the circle is: " + area);
        }
    }


    // option2 calculates and displays the circumference of the circle
    static void option3(double radiusIn)
    {
        if(radiusIn == -999)
        {
           System.out.println("Radius has not been entered");
        }
        else
        {
           double circumference;; // local variable
           circumference = 2 * 3.1416 * radiusIn; // calculate the circumference of the circle
           System.out.println("The circumference of the circle is: " + circumference);
        }
    }
}
```

There are no new programming techniques in this program; it is the *design* that is interesting. The comments are self-explanatory; so we draw your attention only to a few important points:

- The radius is initialized with a "dummy" value of -999. This allows us to check if the radius has been entered before attempting to perform a calculation.
- Choosing menu option 1 causes the method `option1` to be called – the value of the radius entered by the user is returned.
- Choosing menu option 2 causes the method `option2` to be called. The radius of the circle is sent in as a parameter. After using the dummy value to check that a value for the radius has been entered, the area is then calculated and displayed.
- Choosing menu option 3 causes the method `option3` to be called. This is similar to `option2`, but for the circumference instead of the area.
- Choosing option 4 causes the program to terminate – this happens because the body of the **`while`** loop executes only while `choice` is not equal to 4. If it is equal to 4, the loop is not executed and the program ends. The associated **`case`** statement consists simply of the instruction **`break`**, thus causing the program to jump out of the **`switch`** statement.
- You can see that we have had to declare a new `Scanner` object in each method where it is needed – now that you understand the notion of variable *scope*, you should understand why we have had to do this.

Self-test questions

1 Explain the meaning of the term *method*.

2 Consider the following program:

```
public class MethodsQ2
{
  public static void main(String[] args)
  {
    System.out.println(myMethod(3, 5));
    System.out.println(myMethod(3, 5, 10));
  }

  static int myMethod(int firstIn, int secondIn, int thirdIn)
  {
    return firstIn + secondIn + thirdIn;
  }

  static int myMethod(int firstIn, int secondIn)
  {
    return firstIn - secondIn;
  }
}
```

a) By referring to this program:
   i) Distinguish between the terms *actual parameters* and *formal parameters*.
   ii) Explain the meaning of the terms *polymorphism* and *method overloading*?

b) What would be displayed on the screen when this program was run?

c) Explain, giving reasons, the effect of adding either of the following lines into the `main` method:

   i) `System.out.println(myMethod(3));`
   ii) `System.out.println(myMethod(3, 5.7, 10));`

3 What would be displayed on the screen as a result of running the following program?

```
public class MethodsQ3
{
  public static void main(String[] args)
  {
    int x = 3;
    int y = 4;
    System.out.println(myMethod(x, y));
    System.out.println(y);
  }

  static int myMethod(int firstIn, int secondIn)
  {
    int x = 10;
    int y;
    y = x + firstIn + secondIn;
    return y;
  }
}
```

4 What would be displayed on the screen as a result of running the following program?

```
public class MethodsQ4
{
  public static void main(String[] args)
  {
    int x = 2;
    int y = 7;
    System.out.println(myMethod(x, y));
    System.out.println(y);
  }

  static int myMethod(int a, int x)
  {
    int y = 20;
    return y - a - x;
  }
}
```

Programming exercises

1 Implement the programs from the self-test questions in order to verify your answers

2 In chapter 2, programming exercise 3, you wrote a program that converted pounds to kilograms. Rewrite this program, so that the conversion takes place in a separate method which is called by the `main` method**.**

3 In the exercises at the end of chapter 2 you were asked to write a program that calculated the area and perimeter of a rectangle. Re-write this program so that now the instructions for calculating the area and perimeter of the rectangle are contained in two separate methods.

4 a) Design and implement a program that converts a sum of money to a different currency. The amount of money to be converted, and the exchange rate, are entered by the user. The program should have separate methods for:

- obtaining the sum of money from the user;
- obtaining the exchange rate from the user;
- making the conversion;
- displaying the result.

b) Adapt the above program so that after the result is displayed the user is asked if he or she wishes to convert another sum of money. The program continues in this way until the user chooses to quit.

5 a) Write a menu-driven program that provides three options:

- the first option allows the user to enter a temperature in Celsius and displays the corresponding Fahrenheit temperature;
- the second option allows the user to enter a temperature in Fahrenheit and displays the corresponding Celsius temperature;
- the third option allows the user to quit.

The formulae that you need are as follows, where *C* represents a Celsius temperature and *F* a Fahrenheit temperature:

$$F = \frac{9C}{5} + 32$$

$$C = \frac{5(F - 32)}{9}$$

b) Adapt the above program so that the user is not allowed to enter a temperature below absolute zero; this is –273.15C, or –459.67F.