CHAPTER 9

METHODS – LOGICAL PERFORMING UNITS IN PROGRAMS

The word "method" has already been mentioned many times in this book. A method is a piece of program code that performs a well-defined task. In every program we have had a method named **`Main()`**. Method **`Main()`** is always executed first when the execution of a program begins. We have also encountered methods **`Console.Write()`** and **`Console.ReadLine()`** which are standard methods to write data to the screen and read data from the keyboard. In Chapter 8 we studied many string methods (**`IndexOf()`**, **`Substring()`**, **`Compare()`**, etc.) which are standard methods to manipulate strings. Now we are going to take a closer look at methods and their use. You will learn how to write your own methods. The term "calling" will be an important concept associated with methods.

With C# we can write static methods and non-static instance methods. The methods that we are going to study in this chapter are static methods. The instance methods, which are in some ways different from the static methods, will be studied in the following chapter.

9.1 Simple static methods and the concept of calling

All programs that we have studied so far are of the form

```
// SomeName.cs

using System ;

class SomeName
{
   static void Main()
   {
      Statements that declare data (variables, objects, and arrays).

      Functional action statements.
   }
}
```

The source program statements that we have seen so far have been statements of the method named **`Main()`**. Method **`Main()`** has always the reserved words **`static`** and **`void`** preceding its name, and a pair of empty parentheses () is written after the method name **`main`**. The C# statements that dictate what method **`Main()`** does are inside a pair of braces { }.

From now on, we will start studying programs that may contain several methods. The simplest form of a method is similar to method **`Main()`** above. In a simple method, the method is static, the type of the method is **`void`**, and the parentheses after the method name are empty.

Program **Messages.cs** is an example where a simple method is called inside the method **`Main()`**. Although the structure of **Messages.cs** is such that the source code of the method **`print_message()`** is written first and method **`Main()`** is at the end of program, the program execution starts from method **`Main()`**. Method **`Main()`** is the "main program" in the file. The operating system of the computer where the program is executed always begins the program by executing the method that is named **`Main()`**.

Method **`print_message()`** in program **Messages.cs** can be considered a subroutine because its execution is completely controlled by method **`Main()`**. The source code of the subroutine starts

```
static void print_message()
{
   ...
```

Note that there is no semicolon (;) here.

and it is called inside method **`Main()`** simply by writing the method name in the following way

```
print_message() ;
```

What happens in a method call is that the calling method stops running, and the statements of the method that was called are executed. When all statements of the method that was called are executed, the program execution continues in the calling method from the statement that follows the method call.

In **Messages.cs**, method **`print_message()`** is called twice. By studying the output you can find out that **`print_message()`** always prints the same text lines, while method **`Main()`** prints something else in between the message from **`print_message()`**. Program **Messages.cs** could, of course, be written without method **`print_message()`**. If the statements inside method **`print_message()`** were copied to those two places where **`print_message()`** is called in method **`Main()`**, the program would behave in the same way as it is doing now, but it would not need any method calls.

The statements that form the body of method **print_message()** are inside these braces. The structure of this method is similar to the structure of method **Main()**. Generally, the name of a method can be invented by the programmer, but method **Main()** must have that name.

All methods must be written inside some class declaration. In this program, class **Messages** contains two separate methods.

```
//  Messages.cs  (c) Kari Laitinen

using System ;

class Messages
{
   static void print_message()
   {
      Console.Write( "\n   This is method named \"print_message()\"." ) ;
      Console.Write( "\n   Methods usually contain many statements. " ) ;
      Console.Write( "\n   Let us now return to the calling method." ) ;
   }

   static void Main()
   {
      Console.Write( "\n THE FIRST STATEMENT IN METHOD \"Main()\"." ) ;

      print_message() ;

      Console.Write( "\n THIS IS BETWEEN TWO METHOD CALLS." ) ;

      print_message() ;

      Console.Write( "\n END OF METHOD \"Main()\".\n" ) ;
   }
}
```

Method **print_message()** is called twice inside method **Main()**. A simple static method belonging to the same class as the calling method can be called by writing its name, a pair of empty parentheses, and a semicolon. Method calls are statements in C#. What happens in a method call is that the statements inside the called method are executed, and program execution continues from the statement that follows the method call in the calling method.

Messages.cs - 1. Method Main() calling a simple method named print_message().

```
D:\csfiles2>Messages

 THE FIRST STATEMENT IN METHOD "Main()".
   This is method named "print_message()".
   Methods usually contain many statements.
   Let us now return to the calling method.
 THIS IS BETWEEN TWO METHOD CALLS.
   This is method named "print_message()".
   Methods usually contain many statements.
   Let us now return to the calling method.
 END OF METHOD "Main()".
```

Those lines containing mostly lowercase letters are printed by method **print_message()**.

Messages.cs - X. Method print_message() prints always the same message.

In programming terminology, the method that calls another method is the caller, and the method that is called is the callee. In **Messages.cs**, method **`Main()`** is the caller and method **`print_message()`** is the callee. A caller calls a callee like an employer employs an employee. A callee is always subordinate to its caller. The caller decides when a callee is executed. The caller continues by executing the statements that follow the method call when the statements of a callee have been executed.

Methods are executed, statement by statement, from the first statement to the last statement. Although computers can execute statements extremely fast, only one statement is being executed at a time. To better understand what is happening when a program is being executed, we can think that there exists such a thing as "program control". The program control is at that statement which is currently being executed. When the current statement has been completely executed, the program control is passed to the following statement. The program control is at the first executable statement of method **`Main()`** when the execution of a program begins. When the last statement of method **`Main()`** is executed, the program control is passed back to the operating system of the computer.

A method call is a statement that passes the program control to the called method, the callee. Just after the execution of a method call, the program control is at the first executable statement in callee. The program control goes through every statement in callee. After the last statement in the callee has been executed, the program control is passed to the statement that follows the method call in caller.

In large computer programs there are methods that call other methods that call other methods that call other methods ... In well-designed programs there is, of course, always a last method that is called but which does not call any other methods. In large programs, methods are useful because they allow programs to be divided into manageable pieces of source code. Program **Letters.cs** is an example where a called method calls two other methods. Method **`print_letters()`** is a callee in relation to method **`Main()`**, but it is a caller in relation to the two other methods.

Although **Letters.cs** does not do anything that could be considered as creative computing (i.e. the program is a simple textbook program), the program is an example of how a programming task can be divided into smaller programming tasks with the help of methods. What program **Letters.cs** does is that it prints all letters of the English alphabet. First it prints all uppercase letters and then it prints all lowercase letters. We can imagine that **Letters.cs** is the result of a software development project. A boss in a software company could have started a software project to produce a program that first prints all uppercase letters and then all lowercase letters. The software developers working on the project could have divided the programming work into the subtasks

- print uppercase letters
- print lowercase letters

which would have been implemented (i.e. programmed) as two separate methods by different people.

A method is a piece of source program that performs a certain activity. When a caller calls a method, the call is like a command to perform the activity that is programmed inside the method. Because method calls are like commands, it is usual that method names are in a commanding, imperative form. For example, the method names

```
print_uppercase_letters
print_message
```

are in the form of a command, since an imperative verb is the first word in the name. Technically, programmers are free to name methods according to the general naming rules of C#, but it is useful to name methods so that they are commands. This way method names can be easily distinguished from variable names. Inventing accurate and descriptive names for the methods you write helps you to understand your programming task better.

```
//  Letters.cs  (c) 2003 Kari Laitinen

using System ;

class Letters
{
   static void print_uppercase_letters()
   {
      Console.Write( "\n Uppercase English letters are: \n\n" ) ;

      for ( char letter_to_print  =  'A' ;
                 letter_to_print  <= 'Z' ;
                 letter_to_print  ++  )
      {
         Console.Write(  " "  +  letter_to_print ) ;
      }
   }

   static void print_lowercase_letters()
   {
      Console.Write( "\n\n Lowercase English letters are: \n\n" ) ;

      for ( char letter_to_print  =  'a' ;
                 letter_to_print  <= 'z' ;
                 letter_to_print  ++ )
      {
         Console.Write(  " "  +  letter_to_print ) ;
      }
   }

   static void print_letters()
   {
      print_uppercase_letters() ;
      print_lowercase_letters() ;
   }

   static void Main()
   {
      print_letters() ;
   }
}
```

These two methods are called by the method **print_letters()**.

The method called by **Main()** contains two other method calls.

Method **Main()** has only one statement which is a method call. You should note that these methods are in such an order that a callee is always written before the caller. In this book programs are generally written so that the method that will be called later in the program is placed before the calling method in the source program file.

Letters.cs - 1. Method Main() calling a method that calls two other methods.

```
D:\csfiles2>Letters

 Uppercase English letters are:

 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

 Lowercase English letters are:

 a b c d e f g h i j k l m n o p q r s t u v w x y z
```

Letters.cs - X. All text is printed here by the two topmost methods.