Files

This chapter explains:

- what a text file is;
- how to read and write data to files;
- how to manipulate folder paths and names.

Introduction

You have already made use of files, in the sense of having used the IDE to create files containing C# programs, and having used Windows to view a hierarchical structure of folders (directories). In this chapter, we will look at the nature of the information you may choose to store in files, and how you can write programs to manipulate files.

Initially, let us clarify the difference between RAM (random-access memory) and file storage devices (e.g. disks, hard disk drives and CD-ROMs). Storage capacity is measured in bytes in all these cases. There are several major differences between RAM and file storage:

- RAM is used to store programs as they execute. They are copied from a file storage device immediately before execution.
- The time to access an item in RAM is much faster.
- The cost of RAM is higher (megabyte for megabyte).
- Data in RAM is temporary: it is erased when power is switched off. File storage devices can hold data permanently.

The capacity of file storage devices is higher. CD-ROMs have a capacity of around 650 Mbyte (megabytes) and DVDs (Digital Versatile Disks) have a capacity of 4.7 Gbyte (gigabytes, 1024 Mbyte). Both CD-ROM and DVD have writable versions. Typical

hard drives can hold around 80 Gbyte, and the lowly floppy holds around 1.4 Mbyte. However, technology is evolving rapidly – the race is to create cheap, fast, small storage devices that modern computer software requires, especially in the area of high-quality moving images and sound.

The essentials of streams

Streams let us access a file as a sequence of items. The term 'stream' is used in the sense of a stream of data flowing in or out of the program. Let us introduce the jargon, which is similar in most programming languages. If we wish to process the data in an existing file, we must:

1. Open the file.
2. Read (input) the data item-by-item into variables.
3. Close the file when we have finished with it.

To transfer some data from variables into a file, we must:

1. Open the file.
2. Output (write) our items in the required sequence.
3. Close the file when we have finished with it.

Note that, when reading from a file, all we can do is read the next item in the file. If, for example, we only needed to examine the last item in a file, we would have to code a loop to read each preceding item in turn, until the required item is reached. For many tasks, it is convenient to visualize a text file as a series of lines, each made up of a number of characters. Each line is terminated by an end-of-line marker, consisting of either the *newline* character, or the *return* character, or both of these. Your response to this might be to say 'I just hit Enter at the end of a line!' Behind the scenes, Windows software will put the newline and return character at the end of each line. Most of this intricacy is hidden by C#.

As well as files containing lines of text, C# can also manipulate files containing binary data, such as images. However, such data is usually arranged in a more complicated format within files.

We shall make use of the C# classes that allow us to access a file line-by-line, as text strings. This is particularly useful, because many applications (such as word-processors, text editors and spreadsheets) can read and write such files.

The `StreamReader` and `StreamWriter` classes

To read and write lines of text, we will use:

- The `ReadLine` method of `StreamReader`. This reads a whole line of text into a string, excluding the end-of-line marker. (If we need to split the line into separate parts, we can use the `Split` method of the `String` class, described in Chapter 16.)

- The `StreamWriter` class. This has two main methods: `Write` and `WriteLine`. They both write a string to a file, but `WriteLine` adds the end-of-line marker after the string. We can also use `WriteLine` with no string argument, where an end-of-line marker is written to the file. As an alternative to `WriteLine`, we can use `Write` with `\r\n` within quotes. This is shown later in the chapter.
- The `OpenText` and `CreateText` methods of the `File` class. These are static methods, and provide us with a new instance of a text stream. A selection of other methods and properties of the `File` class is covered later in this chapter.

The file classes are in the `System.IO` namespace. This is not automatically imported, so we must put:

```
using System.IO;
```

at the very top of all our file-processing programs.

● File output

The File Output program opens a file and writes three lines to it. The user interface only consists of a single button, so is not shown. Here is the code:

```
private void button1_Click(object sender, System.EventArgs e)
{
    // write some lines of text to the file
    StreamWriter outputStream = File.CreateText(@"c:\myfile.txt");
    outputStream.WriteLine("This file will");
    outputStream.WriteLine("contain 3");
    outputStream.WriteLine("lines of text.");
    outputStream.Close();
}
```

Firstly we create and open the file:

```
StreamWriter outputStream = File.CreateText(@"c:\myfile.txt");
```

Here, we make use of the static method `CreateText` from the `File` class. This creates a new `StreamReader` object for us, and opens the file. Note that there are two items which refer to the file:

- There is a string which specifies the file name that the operating system uses when it displays folders: `@"c:\myfile.txt"`. Alter this path if you wish to create the file in a different place. If you omit the folder name and just use `myfile.txt`, then the file will be created in a folder named `debug`, which is in the `bin` folder of your current C# project. Note the need to use `@`, which makes the `\` have its normal interpretation, rather than its use with end-of-line characters.
- There is a variable which we chose to name as `outputStream`. It is an instance of the class `StreamWriter`, which provides us with the `WriteLine` method. The use of `CreateText` associates `outputStream` with the file `"c:\myfile.txt"`.

To actually write a line of text to the file, we use `WriteLine`, as in:

```
outputStream.WriteLine("This file will");
```

If the data we wish to place in the file is typed in by the user, perhaps in a text box, we would put:

```
outputStream.WriteLine(textBox1.Text)
```

If the file existed already, its original contents will be erased, and replaced by the three lines.

Finally, we close the file:

```
outputStream.Close();
```

This ensures that any data in transit is actually placed in the file, and also allows the file to be re-opened for reading or writing.

The writing process could also have been accomplished using `Write` with the end-of-line characters, as in:

```
outputStream.Write("Two lines\r\nof text.\r\n");
```

This is sometimes convenient when we manipulate a string which contains several lines within it.

In summary, our file output process was:

- to open the file `"C:\myfile.txt"`;
- to output (write) some strings to the file;
- to close the file.

SELF-TEST QUESTION

18.1 Explain what the following code does:

```
string fileName =@"c:\pattern.txt";
StreamWriter streamOut = File.CreateText(fileName);
for (int lines = 1;lines<=10; lines++)
{
    for (int stars = 1;stars<=lines; stars++)
    {
        streamOut.Write("*");
    }
    streamOut.WriteLine();
}
streamOut.Close();
```

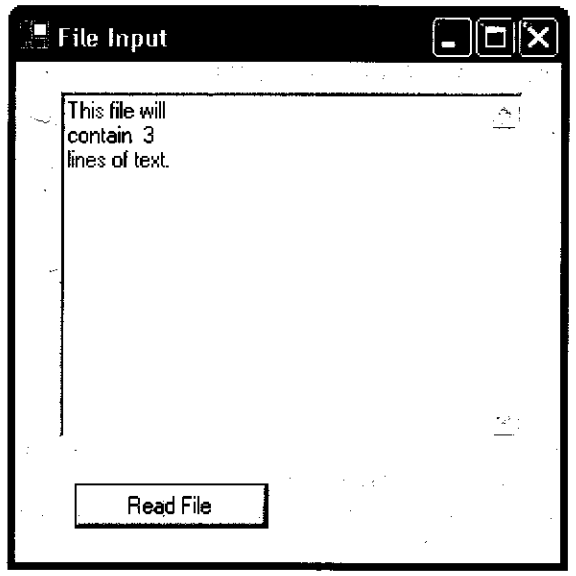

Figure 18.1 Screenshot of File Input program.

File input

Here we examine the program named File Input, which opens a file, inputs its contents, and displays it in a text box. The screenshot (taken after the button was clicked) is given in Figure 18.1. Here is the code:

```
private void button1_Click(object sender, System.EventArgs e)
{
    //read the file line-by-line
    StreamReader inputStream = File.OpenText(@"c:\myfile.txt");
    string line;
    line = inputStream.ReadLine();
    while (line != null)
    {
        textBox1.AppendText(line +"\r\n");
        line = inputStream.ReadLine();
    }
    inputStream.Close();
}
```

The code is contained in the method `button1_Click`. The file we choose to input is the one that was created by the previous File Output program, containing three lines of text.

First, we create a stream to access the file:

```
StreamReader inputStream = File.OpenText(@"c:\myfile.txt");
```

(If the file we specify cannot be found, an exception will be produced. For now, we ignore this possibility, but discuss it later in this chapter.)

Then we use `ReadLine` to input the series of lines in the file, appending each one to our text box. There is one crucial point here: we don't know how many lines are in the file, so we set up a loop which terminates when there is nothing more to read:

```
line = inputStream.ReadLine();
while (line != null)
{
    textBox1.AppendText(line +"\r\n");
    line = inputStream.ReadLine();
}
```

When `ReadLine` runs out of lines to read, it returns an 'empty' object. We can detect this by comparing the line to `null`. This is a keyword in C#, indicating that the object does not exist.

Note that we have used `ReadLine` twice. The first `ReadLine` prior to the loop is needed so that the first time `while` tests `line`, it has a value.

Each line is placed at the end of any existing text already in the text box, by using `Append`. Because `ReadLine` removes end-of-line characters as it reads, we need to put them back for the text box with `\r\n`. If we omitted this, the text box would contain one long line, with no breaks.

Reading a file line-by-line allows us to process each line individually (as we do in the File Search program below). Alternatively, it might be more appropriate to read the whole file into one long string, complete with end-of-line markers. In this case, C# provides us with a method named `ReadToEnd`, which we use for the text editor program shown later.

In summary, the program:

1. opens a file;
2. inputs lines from the file and appends them to the text box, as long as the end of the file is not reached;
3. closes the file.

SELF-TEST QUESTIONS

18.2 The following code is meant to display the length of each line in a file. Explain any problems.

```
string fileName = @"c:\temp.txt";
StreamReader stream = File.OpenText(fileName);
string line;
```

```
line = stream.ReadLine();
while (line != null)
{
    line = stream.ReadLine();
    textBox1.AppendText("length: " + line.Length + ",");
}
stream.Close();
```

18.3 Explain any problems in this code:

```
string fileName = @"c:\temp.txt";
StreamReader stream = File.OpenText(fileName);
string line;
line = stream.ReadLine();
while (line != null)
{
    textBox1.AppendText(line);
}
stream.Close();
```

File searching

Searching a file for an item that meets some specified criteria is a classic task. Here we will construct a program which searches a file of exam marks, which takes the form:

```
J.Doe, 43 , 67
D.Bell ,87,     99
M.Parr, 54, 32
J.Hendrix, 67,43
etc...
```

We might create this file by writing and running a C# program, or with a text editor. Each line is split into three areas, separated by commas. However, we will allow for extra spaces. In data processing, such areas are known as fields. The program will allow us to enter a file name, and to enter a student name, which we assume is unique. If the names are not unique, we would have to introduce an extra field to hold a unique identification number for each person. The program will search the file, and display the marks for our chosen student. The code we need to add to our previous file input example is a `while` which terminates when the end of the file is encountered or when the required name is found. We will use the `Split` method to separate the three fields.

Because there are two ways that the loop can terminate, we introduce an additional variable, `found`, to indicate whether the item was found or not. Note that we expect

that the item will sometimes not be found – it is not regarded as an error, and we code this with `if` rather than using exceptions.

The informal English structure of the search is:

```
found = false
while (more lines to read) And found == false
{    read line
    split line into three fields
    if first field matches name
    {
        found = true
        display rest of fields in labels
    }
}
if not found
{
    display a warning
}
```

We have provided a search button, which causes the file to be opened and searched. The user can select any file for searching. Figure 18.2 shows the screenshot, and here is the code:

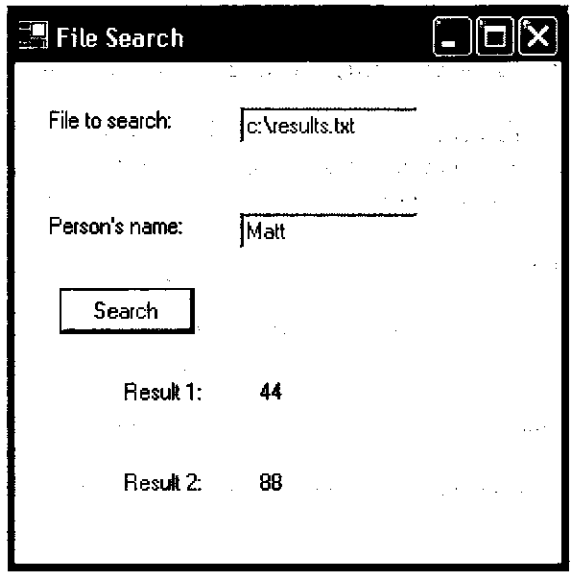

Figure 18.2 Screenshot of File Search program.

```
private void button1_Click(object sender, System.EventArgs e)
{
    string line;
    string[] words=new string[3];
    char []separator = {','};
    bool found = false;
    StreamReader inputStream;

    //clear any previous results
    result1Label.Text = "";
    result2Label.Text = "";

    if (fileNameBox.Text == "")
    {
        MessageBox.Show("Error: missing file name!");
    }
    else if (studentNameBox.Text == "" )
    {
        MessageBox.Show("Error: missing student name!");
    }
    else
    {
        inputStream = File.OpenText(fileNameBox.Text);
        line = inputStream.ReadLine();
        while (line != null && found == false)
        {
            words = line.Split(separator);
            if( words[0].Trim() == studentNameBox.Text)
            {
                result1Label.Text = words[1].Trim();
                result2Label.Text = words[2].Trim();
                found = true;
            }
            else
            {
                line = inputStream.ReadLine();
            }
        }
        if (found==false)
        {
            MessageBox.Show(studentNameBox.Text + " not found");
        }
        inputStream.Close();
    }
}
```


SELF-TEST QUESTIONS

18.4 Amend the search program so that the searching is done in a case-insensitive way (i.e. so that `john` matches `john`, `John` and `JOHN`).

18.5 Explain the problem that would arise if we re-coded our search loop with `||` (or) as in:

```
while (line != null || found == false)
```

Files and exceptions

File input–output is a major source of exceptions. For example, the file name supplied by the user might be incorrect, the disk might be full, or the user might remove a CD-ROM while reading is in progress. We can minimize the effects of incorrectly entered file names by using file dialogs (covered later), which let the user browse folders and click on file names, rather than typing the names into text boxes. But exceptions will still occur. In Chapter 17 we covered exceptions, and here we examine exceptions as they relate to files.

A number of exceptions can be thrown when we access files. Here is a selection:

- the `File.OpenText` method can throw a number of exceptions – we shall single out `FileNotFoundException` in particular;
- the `ReadLine` method of class `StreamReader` and the `WriteLine` method of class `StreamWriter` can throw an `IOException`, along with other classes of exceptions.

Here is another version of the search program, with exception-handling:

```
private void button2_Click(object sender, System.EventArgs e)
{
    string line;
    string[] words=new string[3];
    char []separator = {','};
    bool found = false;
    StreamReader inputStream;

    //clear any previous results
    result1Label.Text = "";
    result2Label.Text = "";

    if (fileNameBox.Text == "" )
    {
        MessageBox.Show("Error: missing file name!");
    }
```

```
        else if (studentNameBox.Text == "" )
        {
            MessageBox.Show("Error: missing student name!");
        }
        else
            try
            {
                inputStream = File.OpenText(fileNameBox.Text);

                line = inputStream.ReadLine();
                while ((line !=null) && found == false)
                {
                    words = line.Split(separator);
                    if (words[0].Trim()== studentNameBox.Text )
                    {
                        result1Label.Text = words[1].Trim();
                        result2Label.Text = words[2].Trim();
                        found = true;
                    }
                    else
                    {
                        line = inputStream.ReadLine();
                    }
                }
                if (found==false)
                {
                    MessageBox.Show(studentNameBox.Text
                        + " not found");
                }

                inputStream.Close();
            }
            catch(FileNotFoundException problem)
            {
                MessageBox.Show(problem.Message
                    + ". Re-enter name.");
            }

            catch(Exception problem)
            {
                MessageBox.Show("Error concerning file: "
                    + fileNameBox.Text
                    + ". " + problem.Message);
            }
    }
```


We have singled out `FileNotFoundException` as the most likely exception, and have produced a specific error message. Other exceptions might occur, but they cannot be neatly classified – so we choose to catch them all in one place, by referring to the class `Exception`.

Message boxes and dialogs

Sometimes we need to bring the user's attention to a vital decision or piece of information. Merely displaying some text in a label is not enough. C# provides a range of overloaded message box methods to provide configured dialogs. In addition, there are specific dialogs to request file names from the user, which we review later. Here are the message box methods:

```
MessageBox.Show(message);
MessageBox.Show(message, title);
MessageBox.Show(message, title, buttons);
MessageBox.Show(message, title, buttons, icon);
```

The arguments are as follows:

- The message argument will be positioned in the centre of the message box.
- The title argument will be positioned at the top of the message box.
- The buttons argument is a constant specifying any buttons we need. For example, we might require yes/no buttons. The `MessageBoxButtons` class provides `AbortRetryIgnore`, `OK`, `OKCancel`, `RetryCancel`, `YesNo`, `YesNoCancel`.
- The icon argument specifies a symbol, such as a large exclamation mark or question mark. The `MessageBoxIcon` class provides us with `Error`, `Exclamation`, `Information`, `Question`.

The `Show` method also returns a code which we can examine to find out which button was clicked. The `DialogResult` class provides these constants which we can use for comparison: `Abort`, `Cancel`, `Ignore`, `No`, `None`, `OK`, `Retry`, `Yes`. Figures 18.3 and 18.4 show some examples, which show a warning message, and a question.

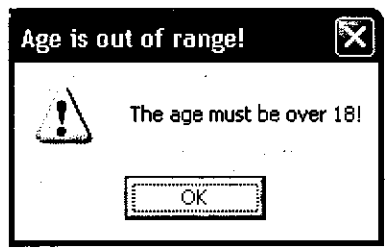

Figure 18.3 Warning with a message box.