CHAPTER 14

Files and Streams

In this chapter you will:

- Learn about computer files and the `File` and `Directory` classes
- Understand file data organization
- Understand streams
- Write to and read from a sequential access text file
- Search a sequential access text file
- Understand serialization and deserialization

In the early chapters of this book, you learned that using variables to store values in computer memory provides programs with flexibility; a program that uses variables to replace constants can manipulate different values each time the program executes. However, when data values in a program are stored in memory, they are lost when the program ends. To retain data values for future use, you must store them in files. In this chapter, you will learn to create and manage files in C#.

Files and the `File` and `Directory` Classes

When data items are stored in a computer system, they can be stored for varying periods of time—temporarily or permanently.

Temporary storage is usually called computer memory, or **random access memory** (RAM). When you write a C# program that stores a value in a variable, you are using temporary storage; the value you store is lost when the program ends or the computer loses power. This type of storage is **volatile**.

Permanent storage, on the other hand, is not lost when a computer loses power; it is **nonvolatile**. When you write a program and save it to a disk, you are using permanent storage. Likewise, you use permanent storage when you write a program that saves data to a file.

When discussing computer storage, *temporary* and *permanent* refer to volatility, not length of time. For example, a *temporary* variable might exist for several hours in a large program or one that the user forgets to end, but a *permanent* piece of data might be saved and then deleted within a few seconds.

A **computer file** is a collection of data stored on a nonvolatile device in a computer system. Files exist on **permanent storage devices**, such as hard disks, USB drives, reels of magnetic tape, and optical discs, which include CDs and DVDs.

You can categorize files by the way they store data:

- **Text files** contain data that can be read in a text editor because the data has been encoded using a scheme such as ASCII or Unicode. Text files might include facts and figures used by business programs; when they do, they are also called **data files**. The C# source programs you have written (with .cs extensions) are stored in text files.
- **Binary files** contain data that has not been encoded as text. Their contents are in binary format, which means that you cannot understand them by viewing them in a text editor. Examples include images, music, videos, and the compiled program files with an .exe extension that you have created using this book.

Although their contents vary, text and binary files have many common characteristics, as follows:

- Each has a name. The name often includes a dot and a file extension that describes the type of the file. For example, MyNotes.txt is a plain text file, and MyPicture.jpg is an image file in Joint Pictures Expert Group format.
- Each file has a specific time of creation and a time it was last modified.
- Each file occupies space on a section of a storage device; that is, each file has a size. Sizes are measured in bytes. A **byte** is a small unit of storage; for example, in a simple text file, a byte holds only one character. Because a byte is so small, file sizes usually are expressed in **kilobytes** (thousands of bytes), **megabytes** (millions of bytes), or **gigabytes** (billions of bytes).

When you use data, you never directly use the copy that is stored in a file. Instead, you use a copy that has been loaded into memory. Especially when data items are stored on a hard disk, their locations might not be clear to you—data just seems to be "in the computer." However, when you work with stored data, you must transfer copies from the storage device into memory. When you copy data from a file on a storage device into RAM, you **read from the file**. When you store data in a computer file on a persistent storage device, you **write to the file**. This means you copy data from RAM to the file.

Because you can erase data from files, some programmers prefer the term ***persistent*** *storage* to permanent storage. In other words, you can remove data from a file stored on a device such as a disk drive, so it is not technically permanent. However, the data remains in the file even when the computer loses power, so, unlike RAM, the data persists, or perseveres.

Computer files are the electronic equivalent of paper documents stored in file cabinets. In a physical file cabinet, the easiest way to store a document is to toss it into a drawer without a folder. When storing computer files, this is the equivalent of placing a file in the main or **root directory** of your storage device. However, for better organization, most office clerks place documents in folders; most computer users also organize their files into **folders** or **directories**. Users also can place folders within folders to form a hierarchy. The combination of the disk drive plus the complete hierarchy of directories in which a file resides is its **path**. For example, in the Windows operating system, the following line would be the complete path for a file named Data.txt on the C drive in a folder named Chapter.14 within the C# folder:

```
C:\C#\Chapter.14\Data.txt
```

The terms *directory* and *folder* are used synonymously to mean an entity that is used to organize files. *Directory* is the more general term; the term *folder* came into use in graphical systems. For example, Microsoft began calling directories *folders* with the introduction of Windows 95.

C# provides built-in classes named `File` and `Directory` that contain methods to help you manipulate files and their directories, respectively.

Using the `File` and `Directory` Classes

The `File` class contains methods that allow you to access information about files. Some of the methods are listed in Table 14-1.

Method	Description
`Create()`	Creates a file
`CreateText()`	Creates a text file
`Delete()`	Deletes a file
`Exists()`	Returns `true` if the specified file exists
`GetCreationTime()`	Returns a `DateTime` object specifying when a file was created
`GetLastAccessTime()`	Returns a `DateTime` object specifying when a file was last accessed
`GetLastWriteTime()`	Returns a `DateTime` object specifying when a file was last modified
`Move()`	Moves a file to the specified location

Table 14-1 Selected `File` class methods

`DateTime` is a structure that contains data about a date and time. In the chapter "Using Controls," you used the data from `DateTime` structures with `MonthCalendar` and `DateTimePicker` GUI objects. `DateTime` values can be expressed using Coordinated Universal Time (UTC), which is the internationally recognized name for Greenwich Mean Time (GMT). By default, `DateTime` values are expressed using the local time set on your computer. The property `DateTime.Now` returns the current local time. The property `DateTime.UtcNow` returns the current UTC time.

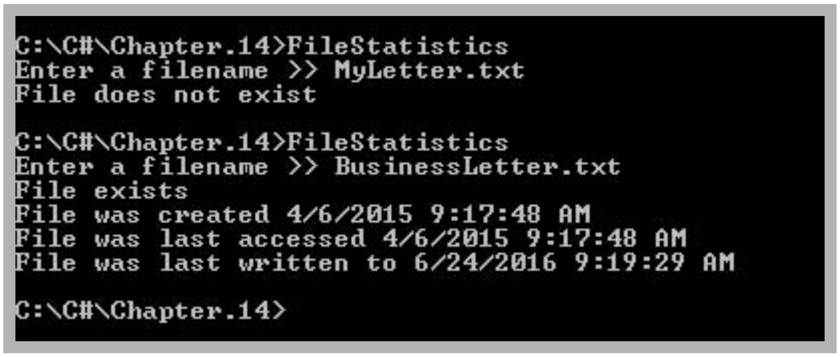
The `File` class is contained in the `System.IO` namespace. So, to use the `File` class, you can use its fully qualified name, `System.IO.File`, or you can just use `File` if you include the statement `using System.IO;` at the top of your program file as shown in the first shaded statement in Figure 14-1. Figure 14-1 shows a program which demonstrates several `File` class methods. The program prompts the user for a filename and then tests the file's existence in the second shaded statement. If the file exists, the creation time, last access time, and last write time are displayed. If the file does not exist, a message is displayed. Figure 14-2 shows two executions of the program. In the first execution, the user enters a filename that is not found. In the second execution, the file is found, and the three significant dates and times are displayed.

The `System.IO.FileInfo` class also allows you to access information about a file. See the Microsoft documentation at *http://msdn.microsoft.com* for more information.

```
using static System.Console;
using System.IO;
class FileStatistics
{
   static void Main()
   {
      string fileName;
      Write("Enter a filename >> ");
      fileName = ReadLine();
      if(File.Exists(fileName))
      {
         WriteLine("File exists");
         WriteLine("File was created " +
            File.GetCreationTime(fileName));
         WriteLine("File was last accessed " +
            File.GetLastAccessTime(fileName));
         WriteLine("File was last written to " +
            File.GetLastWriteTime(fileName));
      }
      else
      {
         WriteLine("File does not exist");
      }
   }
}
```

Figure 14-1 The FileStatistics program

In the FileStatistics program in Figure 14-1, the file must be in the same directory as the program that is running.



```
C:\C#\Chapter.14>FileStatistics
Enter a filename >> MyLetter.txt
File does not exist

C:\C#\Chapter.14>FileStatistics
Enter a filename >> BusinessLetter.txt
File exists
File was created 4/6/2015 9:17:48 AM
File was last accessed 4/6/2015 9:17:48 AM
File was last written to 6/24/2016 9:19:29 AM

C:\C#\Chapter.14>
```

Figure 14-2 Two typical executions of the FileStatistics program

The `Directory` class provides you with information about directories or folders. Table 14-2 lists some available methods in the `Directory` class.

Method	Description
`CreateDirectory()`	Creates a directory
`Delete()`	Deletes a directory
`Exists()`	Returns `true` if the specified directory exists
`GetCreationTime()`	Returns a `DateTime` object specifying when a directory was created
`GetDirectories()`	Returns a `string` array that contains the names of the subdirectories in the specified directory
`GetFiles()`	Returns a `string` array that contains the names of the files in the specified directory
`GetLastAccessTime()`	Returns a `DateTime` object specifying when a directory was last accessed
`GetLastWriteTime()`	Returns a `DateTime` object specifying when a directory was last modified
`Move()`	Moves a directory to the specified location

Table 14-2 Selected `Directory` class methods

Figure 14-3 contains a program that declares an array of string where filenames can be stored. (See the first shaded statement.) The program prompts a user for a directory and then fills the string array with a list of filename using the `GetFiles()` method. (See second shaded statement.) Then a `for` loop is used to display the list of files. Figure 14-4 shows two typical executions of the program.

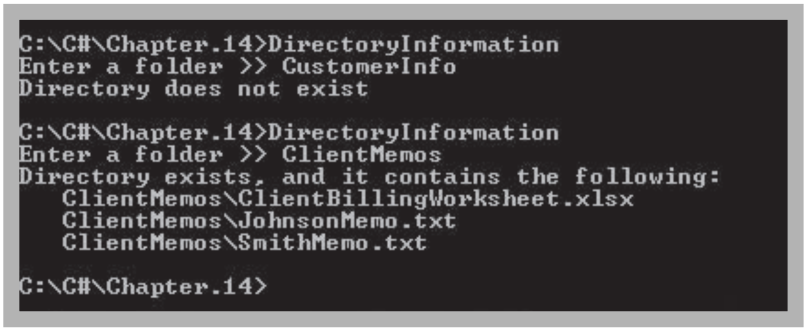
```
using static System.Console;
using System.IO;
class DirectoryInformation
{
   static void Main()
   {
      string directoryName;
      string[] listOfFiles;
      Write("Enter a folder >> ");
      directoryName = ReadLine();
      if(Directory.Exists(directoryName))
      {
         WriteLine("Directory exists, " +
            "and it contains the following: ");
```

Figure 14-3 The `DirectoryInformation` program *(continues)*

(continued)

```
            listOfFiles = Directory.GetFiles(directoryName);
            for(int x = 0; x < listOfFiles.Length; ++x)
               WriteLine("   {0}", listOfFiles[x]);
         }
         else
         {
            WriteLine("Directory does not exist");
         }
      }
   }
```

Figure 14-3 The DirectoryInformation program



```
C:\C#\Chapter.14>DirectoryInformation
Enter a folder >> CustomerInfo
Directory does not exist

C:\C#\Chapter.14>DirectoryInformation
Enter a folder >> ClientMemos
Directory exists, and it contains the following:
   ClientMemos\ClientBillingWorksheet.xlsx
   ClientMemos\JohnsonMemo.txt
   ClientMemos\SmithMemo.txt

C:\C#\Chapter.14>
```

Figure 14-4 Two typical executions of the DirectoryInformation program

Watch the video *File Handling*.

TWO TRUTHS & A LIE

Files and the File and Directory Classes

1. Temporary storage is nonvolatile; permanent storage is volatile.
2. When you write to a file, you copy data from RAM to a permanent storage device.
3. Most computer users organize their files into directories; the complete hierarchy of directories in which a file resides is its path.

The false statement is #1. Temporary storage is volatile; permanent storage is nonvolatile.

Understanding File Data Organization

Businesses store data in a relationship known as the **data hierarchy**, as described in Figure 14-5. For most users, the smallest useful piece of data is the character. A **character** is any one of the letters, numbers, or other special symbols (such as punctuation marks and arithmetic symbols) that constitute data. Characters are made up of bits (the zeros and ones that represent computer circuitry), but people who use data do not care whether the internal representation for an *A* is 01000001 or 10111110; rather, they are concerned with the meaning of *A*. For example, it might represent a grade in a course, a person's initial, or a company code.

C# uses Unicode to represent its characters. You first learned about Unicode in Chapter 1. The set of all the characters used to represent data on a particular computer is that computer's **character set**.

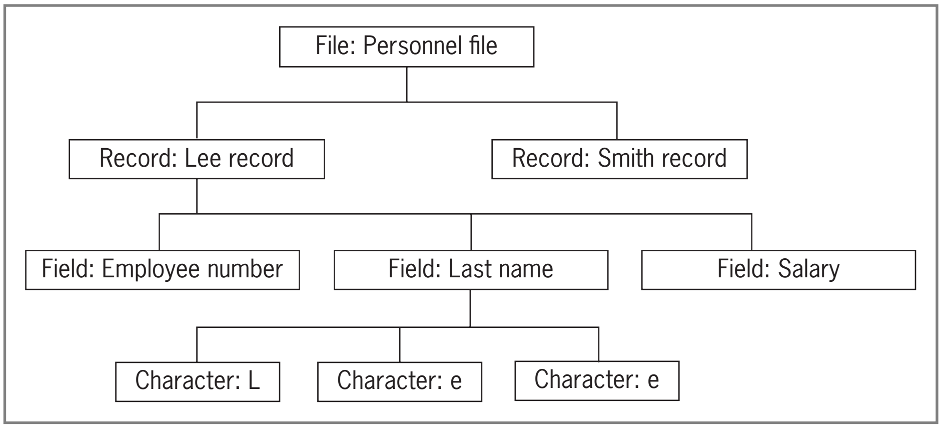


Figure 14-5 Hierarchical relationships of data components

You can think of a character as a unit of information instead of data with a particular appearance. For example, the mathematical character *pi* (π) and the Greek letter *pi* look the same, but have two different Unicode values.

In computer terminology, a character can be any group of bits, and it does not necessarily represent a letter or number. Some characters do not correspond to those in natural language; for example, some "characters" produce a sound or control your display. You also have used the `'\n'` character to start a new line.

When businesses use data, they group characters into fields. A **field** is a character or group of characters that has some meaning. For example, the characters *T*, *o*, and *m* might represent your first name. Other data fields might represent items such as last name, Social Security number, zip code, and salary.

Fields are grouped together to form records. A **record** is a collection of fields that contain data about an entity. For example, the fields that hold a person's first and last names, Social Security number, zip code, and salary collectively are that person's record. When programming in C#, you have created many classes, such as an `Employee` class or a `Student` class. You can think of the data typically stored in each object that is an instance of these classes as a record. In other words, classes contain individual variables that represent data fields. A business's data records usually represent a person, item, sales transaction, or some other concrete object or event.

Records are grouped to create files. Data files consist of related records. For example, a company's personnel file contains many related records—one record for each company employee. Some files have only a few records; perhaps your professor maintains a file for your class with 25 records—one record for each student. Other files contain thousands or even millions of records. For example, a large insurance company maintains a file of policyholders, and a mail-order catalog company maintains a file of available items.

A data file is a **sequential access file** when each record is read in order from first to last in the file. Usually, the records are stored in order based on the value in some field; for example, employees might be stored in Social Security number order, or inventory items might be stored in item number order. The field used to uniquely identify each record in a sequential file is the **key field**. Frequently, records are sorted based on the key field. When records are not used in sequence, the file is used as a **random access file**, in which records can be accessed in any order.

Before an application can use a data file, it must open the file. A C# application **opens a file** by creating an object and associating a stream of bytes with that object. When you finish using a file, the program should **close the file**—that is, make the file no longer available to your application. Failing to close an input file (a file from which you are reading data) usually does not result in serious consequences; the data still exists in the file. However, if you fail to close an output file (a file to which you are writing data), the data might become inaccessible. You should always close every file you open, and you should close the file as soon as you no longer need it. Leaving a file open for no reason uses computer resources, and your computer's performance will suffer. Also, within a network, another program might be waiting to use the file. For example, if your program leaves the company's inventory file open after adding a new item, the program that fills orders for customers might fail to work correctly.

TWO TRUTHS & A LIE

Understanding File Data Organization

1. A field is a character or group of characters that has some meaning.
2. A record is a collection of data files that contain information about an entity.
3. A sequential access data file frequently contains records stored in order based on the value in some field.

The false statement is #2. A record is a collection of fields that contain data about an entity. Data files consist of related records.

Understanding Streams

Whereas people view files as a series of records, with each record containing data fields, C# views files as just a series of bytes. When you perform an input operation in an application, you can picture bytes flowing into your program from an input device through a **stream**, which functions as a pipeline or channel. When you perform output, some bytes flow out of your application through another stream to an output device, as shown in Figure 14-6. A stream is an object, and like all objects, streams have data and methods. The methods allow you to perform actions such as opening, closing, and flushing (clearing) the stream.

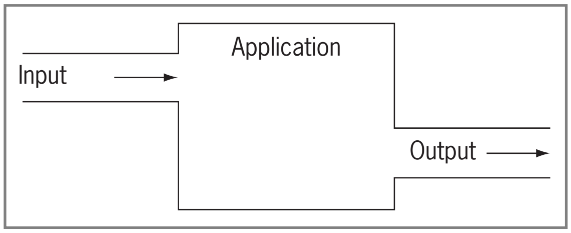


Figure 14-6 File streams

When you produce screen output and accept keyboard input, you use the `Console` class, which provides access to several standard streams:

- `Console.In` refers to the standard input stream object, which accepts data from the keyboard.
- `Console.Out` refers to the standard output stream object, which allows a program to produce output on the screen.
- `Console.Error` refers to the standard error stream object, which allows a program to write error messages to the screen.

You have been using `Console.Out` and its `WriteLine()` and `Write()` methods throughout this book. However, you may not have realized it because you do not need to refer explicitly to `Out`, so you have been writing the instruction as `WriteLine()`. Likewise, you have used `Console.In` with the `ReadLine()` and `Read()` methods.

Most streams flow in only one direction; each stream is either an input or output stream. You might open several streams at once within an application. For example, an application that reads a data disk and separates valid records from invalid ones might require three streams. The data arrives via an input stream, and as the program checks the data for invalid values, one output stream writes some records to a file of valid records, and another output stream writes other records to a file of invalid records.

When you read from or write to a file, you use a file-processing class instead of `Console`. Many file-processing classes are available in C#, including:

- `StreamReader`, for text input from a file
- `StreamWriter`, for text output to a file
- `FileStream` (which is used alone for bytes and with either `StreamReader` and `StreamWriter` for text), for either input from and output to a file

`StreamReader` and `StreamWriter` inherit from `TextReader` and `TextWriter`, respectively. `Console.In` and `Console.Out` are properties of `TextReader` and `TextWriter`, respectively.

When you write a program that stores data in a file, you create a `FileStream` object that defines a file's characteristics and abilities. Programmers say `FileStream` **exposes** a stream around a file. Table 14-3 lists some `FileStream` properties.

Property	Description
`CanRead`	Gets a value indicating whether current `FileStream` supports reading
`CanSeek`	Gets a value indicating whether current `FileStream` supports seeking
`CanWrite`	Gets a value indicating whether current `FileStream` supports writing
`Length`	Gets the length of the `FileStream` in bytes
`Name`	Gets the name of the `FileStream`
`Position`	Gets or sets the current position of the `FileStream`

Table 14-3 Selected `FileStream` properties

The `FileStream` class has 15 overloaded constructors. One that is used frequently includes the filename (which might include the complete path), mode, and type of access. For example, you might construct a `FileStream` object using the following statement:

```
FileStream outFile = new FileStream("SomeText.txt",
   FileMode.Create, FileAccess.Write);
```

In this example, the filename is *SomeText.txt*. Because no path is indicated, the file is assumed to be in the current directory. The mode is `Create`, which means a new file will be created even if one with the same name already exists. The access is `Write`, which means you can write data to the file but not read from it.

Another of `FileStream`'s overloaded constructors requires only a filename and mode. If you use this version and the mode is set to `Append`, then the default access is `Write`; otherwise, the access is set to `ReadWrite`.

Table 14-4 describes the available file modes, and Table 14-5 describes the access types.

Member	Description
`Append`	Opens the file if it exists and seeks the end of the file to append new data
`Create`	Creates a new file; if the file already exists, it is overwritten
`CreateNew`	Creates a new file; if the file already exists, an `IOException` is thrown
`Open`	Opens an existing file; if the file does not exist, a `System.IO.FileNotFoundException` is thrown
`OpenOrCreate`	Opens an existing file; if the file does not exist, it is created
`Truncate`	Opens an existing file; once opended, the file is truncated so its size is zero bytes

Table 14-4 The `FileMode` enumeration

Member	Description
`Read`	Data can be read from the file.
`ReadWrite`	Data can be read from and written to the file.
`Write`	Data can be written to the file.

Table 14-5 The `FileAccess` enumeration

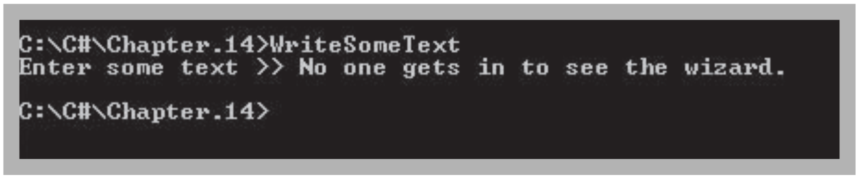
You can use a `FileStream` object as an argument to the `StreamWriter` constructor. Then you use `WriteLine()` or `Write()` with the `StreamWriter` object in much the same way you use it with `Console.Out`. The `FileStream` object must be created first, followed by the `StreamWriter`. At the end of the application, they must be closed in reverse order.

For example, Figure 14-7 shows an application in which a `FileStream` object named `outFile` is created, then associated with a `StreamWriter` named `writer` in the first shaded line. The `writer` object then uses `WriteLine()` to send a `string` to the `FileStream` file instead of sending it to the `Console`. Figure 14-8 shows a typical execution of the program, and Figure 14-9 shows the file as it appears in Notepad.

```
using static System.Console;
using System.IO;
class WriteSomeText
{
   static void Main()
   {
      FileStream outFile = new
         FileStream("SomeText.txt", FileMode.Create,
            FileAccess.Write);
      StreamWriter writer = new StreamWriter(outFile);
      Write("Enter some text >> ");
      string text = ReadLine();
      writer.WriteLine(text);
      // Error occurs if the next two statements are reversed
      writer.Close();
      outFile.Close();
   }
}
```

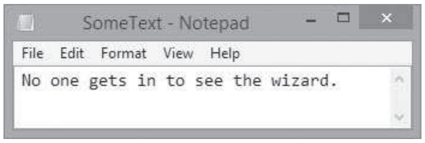
Figure 14-7 The `WriteSomeText` program

Although the WriteSomeText application uses `ReadLine()` to accept user input, you could also create a GUI `Form` to accept input. You will create an application that writes to and reads from files using a GUI environment in a "You Do It" exercise at the end of this chapter.



```
C:\C#\Chapter.14>WriteSomeText
Enter some text >> No one gets in to see the wizard.

C:\C#\Chapter.14>
```

Figure 14-8 Typical execution of the `WriteSomeText` program



```
No one gets in to see the wizard.
```

Figure 14-9 File created by the `WriteSomeText` program

In most applications that use files, you will want to place all the statements that open, write to, read from, and close files in a `try` block and then catch any `IOExceptions` that are thrown. Exception handling is eliminated from many examples in this chapter so that you can concentrate on the details of handling files without extra statements. In the "You Do It" section at the end of this chapter, you will add exception handling to an application.

The classes `BinaryReader` and `BinaryWriter` exist for working with binary files, which can store any of the 256 combinations of bits in any byte instead of just those combinations that form readable text. For example, photographs and music are stored in binary files.

The classes `XmlTextReader` and `XmlTextWriter` exist for working with XML files. **XML** is an abbreviation of eXtensible Markup Language, which is a standard for exchanging data over the Internet.

TWO TRUTHS & A LIE

Understanding Streams

1. When a file is opened in C#, an object is created, and a stream is associated with that object.
2. Most streams flow in only one direction; each stream is either an input or output stream.
3. You can open only one stream at a time within a C# application.

The false statement is #3. You might open several streams at once within an application.

Writing and Reading a Sequential Access File

Although people think of data files as consisting of records that contain fields, C# uses files only as streams of bytes. Therefore, when you write a program to store a data file or write a program to retrieve data from an already-created file, you must dictate the form in which the program will handle the file. Additionally, whether you are writing data to a file or reading data from one, you create a `FileStream` object.

Writing Data to a Sequential Access Text File

For example, suppose that you want to store `Employee` data in a file. Assume that an `Employee` contains an ID number, a name, and a salary that are respectively an `int`, a `string`, and a `double`. You could write stand-alone data for each of the three data types to a file, or you could create an `Employee` class that is similar to many you have seen throughout this book. Figure 14-10 shows a typical `Employee` class that contains three fields and properties for each.

```
class Employee
{
   public int EmpNum {get; set;}
   public string Name {get; set;}
   public double Salary {get; set;}
}
```

Figure 14-10 An `Employee` class

To store `Employee` data to a persistent storage device, you declare a `FileStream` object to define the characteristics of the stored file. For example:

```
FileStream outFile = new FileStream(FILENAME,
   FileMode.Create, FileAccess.Write);
```

The `FileStream` object is then passed to the constructor of a `StreamWriter` object so that text can be written. For example:

```
StreamWriter writer = new StreamWriter(outFile);
```

You can then use the `writer` object's `WriteLine()` method to write `Employee` data to the output stream. You can compose strings to write using text fields and commas. A block of text within a string that represents an entity or field is a **token**. Each comma that separates tokens is a **delimiter**, which is a character used to specify the boundary between data items in text files. Without a delimiter, the process of separating and interpreting tokens is more difficult. For example, suppose you define a delimiter as a comma using the following statement:

```
const string DELIM = ",";
```

Then, when you write data to a file, you can separate the fields with the comma delimiter using a statement such as the following:

```
writer.WriteLine(emp.EmpNum + DELIM + emp.Name + DELIM + emp.Salary);
```

A delimiter can be any character that is not needed as part of the data in a file, but a comma is commonly used. A file that contains comma-separated values is often called a **CSV file**. When a comma cannot be used as a delimiter because commas are needed as characters within the data, sometimes either the Tab character, the pipe character (|), or a comma within quotation marks is used as a delimiter.

The `WriteLine()` method ends output with a carriage return. When you use this method to create file output, the automatically appended carriage return becomes the *record delimiter*.

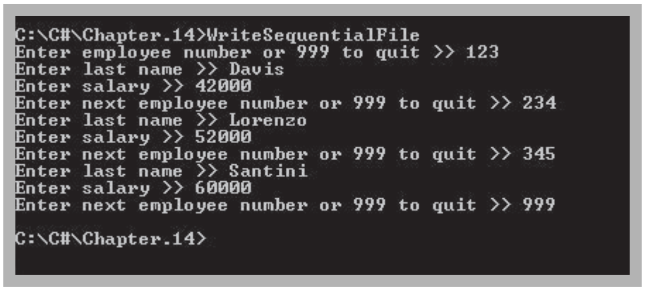
Figure 14-11 contains a complete program that opens a file and continuously prompts the user for `Employee` data. The first shaded statement in the figure is a **priming read—**an input statement that gets a first data item or record. In this case, the first item is an employee number. If the first input is the `END` value, the `while` loop is never entered. If a valid employee number value is entered, the rest of the record's data is retrieved within the loop, and the next employee number is entered in the second shaded statement at the bottom of the loop. When all three fields have been entered for an employee, the fields are written to the file, separated by commas. When the user enters the sentinel value 999 for an `Employee` ID number, the data entry loop ends, and the file is closed. Figure 14-12 shows a typical execution, and Figure 14-13 shows the contents of the sequential data file that is created.

```
using System;
using static System.Console;
using System.IO;
class WriteSequentialFile
{
   static void Main()
   {
      const int END = 999;
      const string DELIM = ",";
      const string FILENAME = "EmployeeData.txt";
      Employee emp = new Employee();
      FileStream outFile = new FileStream(FILENAME,
         FileMode.Create, FileAccess.Write);
      StreamWriter writer = new StreamWriter(outFile);
      Write("Enter employee number or " + END +
         " to quit >> ");
      emp.EmpNum = Convert.ToInt32(ReadLine());
      while(emp.EmpNum != END)
      {
         Write("Enter last name >> ");
         emp.Name = ReadLine();
         Write("Enter salary >> ");
         emp.Salary = Convert.ToDouble(ReadLine());
         writer.WriteLine(emp.EmpNum + DELIM + emp.Name +
            DELIM + emp.Salary);
         Write("Enter next employee number or " +
            END + " to quit >> ");
         emp.EmpNum = Convert.ToInt32(ReadLine());
      }
      writer.Close();
      outFile.Close();
   }
}
```

Figure 14-11 The `WriteSequentialFile` class

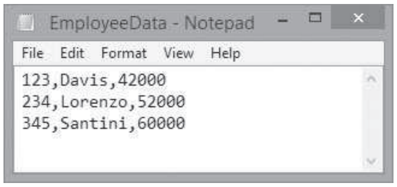
In the `WriteSequentialFile` class, the delimiter is defined to be a `string` instead of a `char` to force the composed argument to `WriteLine()` to be a `string`. If the first data field sent to `WriteLine()` was a `string`, then `DELIM` could have been declared as a `char`.

In the `WriteSequentialFile` program in Figure 14-11, the constant `END` is defined to be 999 so it can be used to check for the sentinel value. You first learned to use named constants in Chapter 2. Defining a named constant eliminates using a magic number in a program. The term **magic number** refers to the bad programming practice of hard-coding numbers (unnamed, literal constants) in code without explanation. In most cases, this makes programs harder to read, understand, and maintain.



```
C:\C#\Chapter.14>WriteSequentialFile
Enter employee number or 999 to quit >> 123
Enter last name >> Davis
Enter salary >> 42000
Enter next employee number or 999 to quit >> 234
Enter last name >> Lorenzo
Enter salary >> 52000
Enter next employee number or 999 to quit >> 345
Enter last name >> Santini
Enter salary >> 60000
Enter next employee number or 999 to quit >> 999

C:\C#\Chapter.14>
```

Figure 14-12 Typical execution of the `WriteSequentialFile` program



```
123,Davis,42000
234,Lorenzo,52000
345,Santini,60000
```

Figure 14-13 Contents of file created by the `WriteSequentialFile` program

Reading from a Sequential Access Text File

A program that reads from a sequential access data file contains many similar components to one that writes to a file. For example, a `FileStream` object is created, as in a program that writes a file. However, the access must be `FileAccess.Read` (or `ReadWrite`), as in the following statement:

```
FileStream inFile = new FileStream(FILENAME,
   FileMode.Open, FileAccess.Read);
```

Then, as data is being written, the `FileStream` object is passed to a `StreamReader` object's constructor, as in the following statement:

```
StreamReader reader = new StreamReader(inFile);
```

After the `StreamReader` has been defined, the `ReadLine()` method can be used to retrieve one line at a time from the data file. For example, the following statement gets one line of data from the file and stores it in a string named `recordIn`:

```
string recordIn = reader.ReadLine();
```

Using `ReadLine()` assumes a carriage return is the record delimiter, which is true, for example, if the records were created using `WriteLine()`.

If the value of `recordIn` is `null`, then no more data exists in the file. Therefore, a loop that begins `while(recordIn != null)` can be used to control the data entry loop.

After a record (line of data) is read in, the `Split()` method can be used to separate the data fields into an array of strings. The `Split()` method is a member of the `String` class; it takes a character delimiter parameter and separates a string into substrings at each occurrence of the delimiter. For example, the following code splits `recordIn` into the `fields` array at each `DELIM` occurrence. Then the three array elements can be stored as an `int`, `string`, and `double`, respectively.

```
string[] fields;
fields = recordIn.Split(DELIM);
emp.EmpNum = Convert.ToInt32(fields[0]);
emp.Name = fields[1];
emp.Salary = Convert.ToDouble(fields[2]);
```

Figure 14-14 contains a complete ReadSequentialFile application that uses the data file created in Figure 14-12. The records stored in the EmployeeData.txt file are read in one at a time, split into their `Employee` record components, and displayed. The first shaded statement is the priming read, and all subsequent records are input using the second shaded statement. Figure 14-15 shows the output.

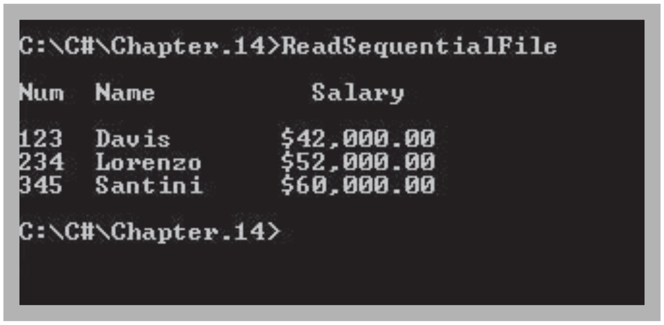
```
using System;
using static System.Console;
using System.IO;
class ReadSequentialFile
{
   static void Main()
   {
      const char DELIM = ',';
      const string FILENAME = "EmployeeData.txt";
      Employee emp = new Employee();
```

Figure 14-14 The `ReadSequentialFile` program *(continues)*

(continued)

```
      FileStream inFile = new FileStream(FILENAME,
         FileMode.Open, FileAccess.Read);
      StreamReader reader = new StreamReader(inFile);
      string recordIn;
      string[] fields;
      WriteLine("\n{0,-5}{1,-12}{2,8}\n",
         "Num", "Name", "Salary");
      recordIn = reader.ReadLine();
      while(recordIn != null)
      {
         fields = recordIn.Split(DELIM);
         emp.EmpNum = Convert.ToInt32(fields[0]);
         emp.Name = fields[1];
         emp.Salary = Convert.ToDouble(fields[2]);
         WriteLine("{0,-5}{1,-12}{2,8}",
            emp.EmpNum, emp.Name, emp.Salary.ToString("C"));
         recordIn = reader.ReadLine();
      }
      reader.Close();
      inFile.Close();
   }
}
```

Figure 14-14 The `ReadSequentialFile` program



```
C:\C#\Chapter.14>ReadSequentialFile

Num  Name           Salary

123  Davis        $42,000.00
234  Lorenzo      $52,000.00
345  Santini      $60,000.00

C:\C#\Chapter.14>
```

Figure 14-15 Output of the `ReadSequentialFile` program

Watch the video *Sequential Access Files*.

TWO TRUTHS & A LIE

Writing and Reading a Sequential Access File

1. Although people think of data files as consisting of records that contain fields, C# uses files only as streams of bytes.
2. A comma is the default C# delimiter.
3. The `Split()` method can be used to separate data fields into an array of strings based on the placement of the designated delimiter.

The false statement is #2. A delimiter is any character used to specify the boundary between characters in text files. Although a comma is commonly used for this purpose, there is no default C# delimiter, and any character could be used.

You Do It

Creating a File

In the next steps, you create a file that contains a list of names.

1. Open a new file named **CreateNameFile**, and write the first lines needed for a program that creates a file of names.

```
using System;
using System.IO;
class CreateNameFile
{
```

2. Start a `Main()` method that declares a `FileStream` you can use to create a file named Names.txt that is open for writing. Also create a `StreamWriter` to which you associate the file.

```
static void Main()
{
   FileStream file = new FileStream("Names.txt",
      FileMode.Create, FileAccess.Write);
   StreamWriter writer = new StreamWriter(file);
```

(continues)

(continued)

3. Add an array of names as follows. Each name is padded with spaces to make it 10 characters long so that you can easily demonstrate the `Seek()` method in a later exercise.

```
string[] names = {"Anthony   ",
                  "Belle      ",
                  "Carolyn    ",
                  "David      ",
                  "Edwin      ",
                  "Frannie    ",
                  "Gina      ",
                  "Hannah    ",
                  "Inez      ",
                  "Juan      "};
```

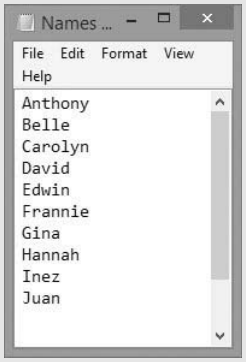
4. Declare a variable to use as an array subscript, then write each name to the output file.

```
int x;
for(x = 0; x < names.Length; ++x)
   writer.WriteLine(names[x]);
```

5. Close the `StreamWriter` and the `FileStream`. Also add two closing curly braces—one for the `Main()` method and one for the class.

```
      writer.Close();
      file.Close();
   }
}
```

6. Save the file, and then compile and execute it. Open the newly created **Names.txt** file in a text editor. The file contents appear in Figure 14-16.



```
Anthony
Belle
Carolyn
David
Edwin
Frannie
Gina
Hannah
Inez
Juan
```

Figure 14-16 File created by the `CreateNameFile` program

(continues)

(continued)

Reading from a File

In the next steps, you read the text from the file created by the CreateNameFile program.

1. Start a new file named **ReadNameFile** as follows:

```
using static System.Console;
using System.IO;
class ReadNameFile
{
```

2. Start a `Main()` method that declares a `FileStream` that uses the same filename as the one created by the `CreateNameFile` program. Declare the file mode to be `Open` and the access to be `Read`. Declare a `StreamReader` with which to associate the file. Also declare an integer that counts the names read and a `string` that holds the names.

```
static void Main()
{
   FileStream file = new FileStream("Names.txt",
      FileMode.Open, FileAccess.Read);
   StreamReader reader = new StreamReader(file);
   int count = 1;
   string name;
```

3. Display a heading, and read the first line from the file. While a name is not `null`, display a count and a name, and increment the count.

```
WriteLine("Displaying all names");
name = reader.ReadLine();
while(name != null)
{
   WriteLine("" + count + " " + name);
   name = reader.ReadLine();
   ++count;
}
```

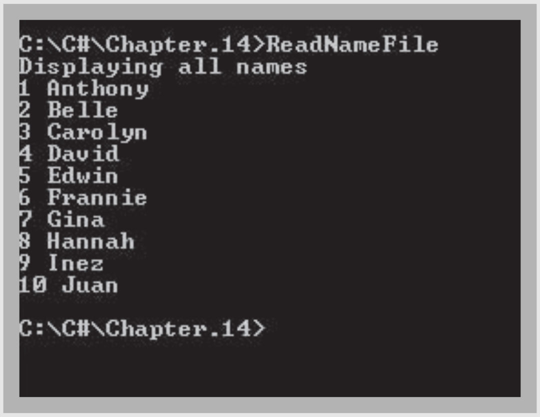
4. Close the `StreamReader` and the `File`, and add closing curly braces for the method and the class.

```
      reader.Close();
      file.Close();
   }
}
```

(continues)

(continued)

5. Save the file, and then compile and execute it. The output appears in Figure 14-17.



```
C:\C#\Chapter.14>ReadNameFile
Displaying all names
1 Anthony
2 Belle
3 Carolyn
4 David
5 Edwin
6 Frannie
7 Gina
8 Hannah
9 Inez
10 Juan

C:\C#\Chapter.14>
```

Figure 14-17 Output produced by the `ReadNameFile` program

Searching a Sequential Text File

When you read data from a sequential text file, as in the `ReadSequentialFile` program in Figure 14-14, the program starts at the beginning of the file and reads each record in turn until all the records have been read. Subsequent records are read in order because a file's **file position pointer** holds the byte number of the next byte to be read. For example, if each record in a file is 32 bytes long, then the file position pointer holds 0, 32, 64, and so on in sequence during the execution of the program.

Sometimes it is necessary to process a file multiple times from the beginning during a program's execution. For example, suppose you want to continue to prompt a user for a minimum salary and then search through a file for `Employee`s who make at least that salary. You can compare the user's entered minimum with each salary in the data file and list those employees who meet the requirement. However, after one list is produced, the file pointer is at the end of the file, and no more records can be read. To reread the file, you could close it and reopen it, but that requires unnecessary overhead. Instead, you can just reposition the file pointer using the `Seek()` method and the `SeekOrigin` enumeration. For example, the following statement repositions the pointer of a file named `inFile` to 0 bytes away from the `Begin` position of the file:

```
inFile.Seek(0, SeekOrigin.Begin);
```

Table 14-6 lists the values in the `SeekOrigin` enumeration that you can use.

Member	Description
`Begin`	Specifies the beginning of a stream
`Current`	Specifies the current position of a stream
`End`	Specifies the end of a stream

Table 14-6 The `SeekOrigin` enumeration

Figure 14-18 contains a program that repeatedly searches a file to produce lists of employees who meet a minimum salary requirement. The shaded portions of the program represent differences from the ReadSequentialFile application in Figure 14-14. In this program, each time the user enters a minimum salary that does not equal 999, the file position pointer is set to the beginning of the file, and then each record is read and compared to the minimum. Figure 14-19 shows a typical execution of the program.

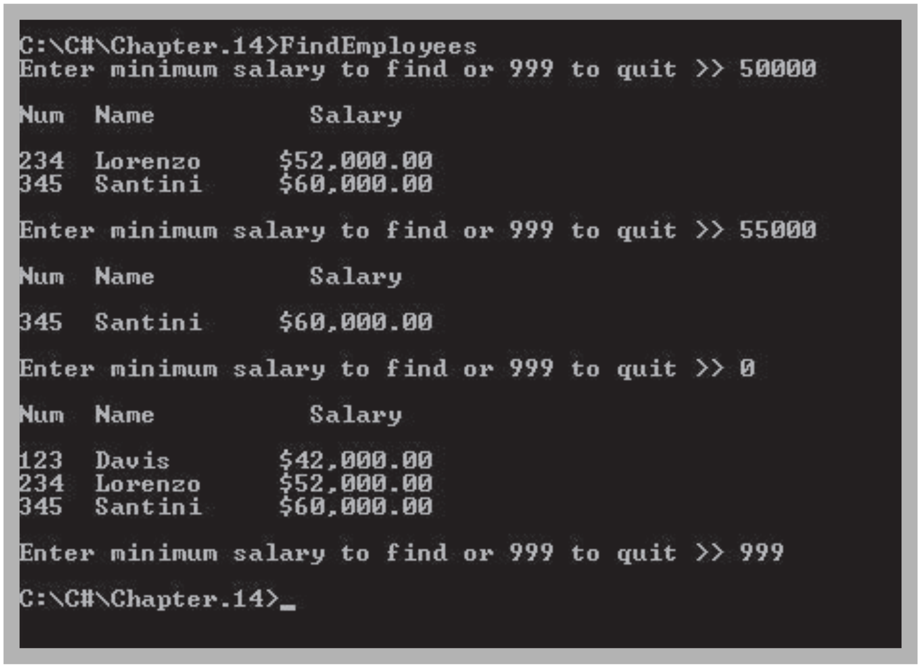
```
using System;
using static System.Console;
using System.IO;
class FindEmployees
{
   static void Main()
   {
      const char DELIM = ',';
      const int END = 999;
      const string FILENAME = "EmployeeData.txt";
      Employee emp = new Employee();
      FileStream inFile = new FileStream(FILENAME,
         FileMode.Open, FileAccess.Read);
      StreamReader reader = new StreamReader(inFile);
      string recordIn;
      string[] fields;
      double minSalary;
      Write("Enter minimum salary to find or " +
         END + " to quit >> ");
      minSalary = Convert.ToDouble(ReadLine());
      while(minSalary != END)
      {
         WriteLine("\n{0,-5}{1,-12}{2,8}\n",
            "Num", "Name", "Salary");
         inFile.Seek(0, SeekOrigin.Begin);
         recordIn = reader.ReadLine();
```

Figure 14-18 The `FindEmployees` program *(continues)*

(continued)

```
         while(recordIn != null)
         {
            fields = recordIn.Split(DELIM);
            emp.EmpNum = Convert.ToInt32(fields[0]);
            emp.Name = fields[1];
            emp.Salary = Convert.ToDouble(fields[2]);
            if(emp.Salary >= minSalary)
               WriteLine("{0,-5}{1,-12}{2,8}", emp.EmpNum,
                  emp.Name, emp.Salary.ToString("C"));
            recordIn = reader.ReadLine();
         }
         Write("\nEnter minimum salary to find or " +
            END + " to quit >> ");
         minSalary = Convert.ToDouble(ReadLine());
      }
      reader.Close(); // Error occurs if
      inFile.Close(); // these two statements are reversed
   }
}
```

Figure 14-18 The `FindEmployees` program



```
C:\C#\Chapter.14>FindEmployees
Enter minimum salary to find or 999 to quit >> 50000

Num  Name          Salary

234  Lorenzo     $52,000.00
345  Santini     $60,000.00

Enter minimum salary to find or 999 to quit >> 55000

Num  Name          Salary

345  Santini     $60,000.00

Enter minimum salary to find or 999 to quit >> 0

Num  Name          Salary

123  Davis       $42,000.00
234  Lorenzo     $52,000.00
345  Santini     $60,000.00

Enter minimum salary to find or 999 to quit >> 999

C:\C#\Chapter.14>_
```

Figure 14-19 Typical execution of the `FindEmployees` program

The program in Figure 14-18 is intended to demonstrate using the `Seek()` method. In a business setting, you might prefer to not leave a file open in one application if other users might be waiting for it. As an alternative, you could load all the records into an array and then search the array for desired records. Problems also exist with this approach because you would have to overestimate the number of records in the file to create an array large enough to hold them.

When you seek beyond the length of the file, you do not cause an error. Instead, the file size grows. In Microsoft Windows NT and later, any data added to the end of a file is set to zero. In Microsoft Windows 98 or earlier, any data added to the end of the file is not set to zero. This means that previously deleted data might become visible to the stream.

TWO TRUTHS & A LIE

Searching a Sequential Text File

1. When you read data from a sequential file, the program starts at the beginning of the file and reads each record in turn until all the records have been read.
2. When you read from a sequential file, its file position pointer holds the number of the record to be read.
3. To reread a file, you can close it and reopen it, or you can reposition the file pointer to the beginning of the file.

The false statement is #2. When you read from a sequential file, its file position pointer holds the byte number of the next byte to be read.

You Do It

Using the `Seek()` Method

In the next steps, you use the `Seek()` method to reposition a file pointer so you can access a file from any location. The user will be prompted to enter a number representing a starting point to list the names in the Names.txt file. Names from that point forward will be listed, and then the user will be prompted for another selection.

1. Start a new program named **AccessSomeNames** that demonstrates how to access requested names from the Names.txt file you created in the CreateNameFile application.

(continues)

(continued)

```
using System;
using static System.Console;
using System.IO;
class AccessSomeNames
{
   static void Main()
   {
      FileStream file = new FileStream("Names.txt",
         FileMode.Open, FileAccess.Read);
      StreamReader reader = new StreamReader(file);
```

2. Declare a constant named `END` that represents an input value that allows the user to exit the program. Then declare other variables that the program will use.

```
const int END = 999;
int count = 0;
int num;
int size;
string name;
```

3. Read a line from the input file. While names are available, continue to read and count them. Then compute the size of each name by dividing the file length by the number of strings stored in it.

```
name = reader.ReadLine();
while(name != null)
{
   ++count;
   name = reader.ReadLine();
}
size = (int)file.Length / count;
```

4. Prompt the user for the number of the first record to read, and read the value from the `Console`.

```
Write("\nWith which number do you want to start? >> ");
num = Convert.ToInt32(ReadLine());
```

5. Start a loop that continues as long as the user does not enter the sentinel `END` value. Within the loop, display the user's number, and then use the `Seek()` method to position the file pointer at the correct file location. Because users enter numbers starting with 1 and file positions start with 0, you calculate the file position by first subtracting 1 from the user's entry. The calculated record number is then multiplied by the size of

(continues)

(continued)

each name in the file. For example, if each name is 10 bytes long, then the calculated starting position should be 0, 10, 20, 30, or some other multiple of the name size.

```
while(num != END)
{
   WriteLine("Starting with name " + num +": ");
   file.Seek((num - 1) * size, SeekOrigin.Begin);
```

6. Read and write the name at the calculated location. Then, in a nested loop, read and write all the remaining names until the end of the file.

```
name = reader.ReadLine();
WriteLine(" " + name);
while(name != null)
{
   name = reader.ReadLine();
   WriteLine(" " + name);
}
```

7. Finally, prompt the user for the next starting value for a new list, inform the user how to quit the application, and accept the next input value. Add a closing brace for the outer `while` loop.

```
   WriteLine("\nWith which number do you " +
      "want to start?");
   Write("    (Enter " + END + " to quit) >> ");
   num = Convert.ToInt32(ReadLine());
}
```

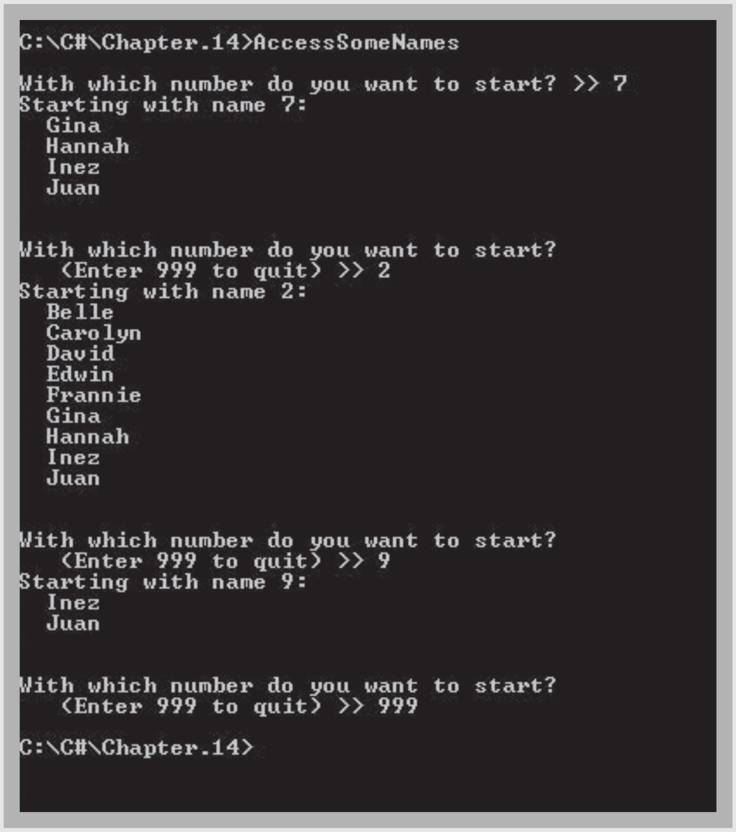
8. Close the `StreamReader` and `File` objects, and add closing braces for the method and the class.

```
      reader.Close();
      file.Close();
   }
}
```

9. Save the file, and then compile and execute it. Figure 14-20 shows a typical execution during which the user displays three sets of names starting at a different point each time.

(continues)

(continued)



```
C:\C#\Chapter.14>AccessSomeNames

With which number do you want to start? >> 7
Starting with name 7:
  Gina
  Hannah
  Inez
  Juan


With which number do you want to start?
   (Enter 999 to quit) >> 2
Starting with name 2:
  Belle
  Carolyn
  David
  Edwin
  Frannie
  Gina
  Hannah
  Inez
  Juan


With which number do you want to start?
   (Enter 999 to quit) >> 9
Starting with name 9:
  Inez
  Juan


With which number do you want to start?
   (Enter 999 to quit) >> 999

C:\C#\Chapter.14>
```

Figure 14-20 Typical execution of the `AccessSomeNames` program

Understanding Serialization and Deserialization

Writing to a text file allows you to store data for later use. However, writing to a text file does present two disadvantages:

- Data in a text file is easily readable in a text editor such as Notepad. Although this feature is useful to developers when they test programs, it is not a very secure way to store data.
- When a record in a data file contains many fields, it is cumbersome to convert each field to text and combine the fields with delimiters before storing the record on a disk. Similarly, when you read a text file, it is somewhat unwieldy to eliminate the delimiters, split the text into tokens, and convert each token to the proper data type. Writing an entire object to a file at once would be more convenient.

C# provides a technique called serialization that can be used for writing objects to and reading objects from data files. **Serialization** is the process of converting objects into streams of bytes. **Deserialization** is the reverse process; it converts streams of bytes back into objects.

To create a class that can be serialized, you mark it with the `[Serializable]` attribute, as shown in the shaded statement in Figure 14-21. The `Employee` class in the figure is identical to the one in Figure 14-10 except for the `[Serializable]` attribute.

```
[Serializable]
class Employee
{
   public int EmpNum {get; set;}
   public string Name {get; set;}
   public double Salary {get; set;}
}
```

Figure 14-21 The serializable `Employee` class

Attributes provide a method of associating information with C# code. They are always contained in square brackets. Search for *C# attributes* at *http://msdn.microsoft.com* for more details.

In a class marked with the `[Serializable]` attribute, every instance variable must also be serializable. By default, all C# simple data types are serializable, including `string`s. However, if your class contains fields that are more complex data types, you must check the declaration of those classes to ensure they are serializable. By default, array objects are serializable. However, if the array contains references to other objects, such as `Date`s or `Student`s, those objects must be serializable.

If you create a class and want to be able to write its objects to a file, you can implement the `ISerializable` interface instead of marking a class with the `[Serializable]` attribute. When you use this approach, you must write a method named `GetObjectData()`. Marking the class with the attribute is simpler.

Two namespaces are included in programs that employ serialization:

- `System.Runtime.Serialization.Formatters.Binary;`
- `System.Runtime.Serialization;`

When you create a program that writes objects to files, you declare an instance of the `BinaryFormatter` class with a statement such as the following:

```
BinaryFormatter bFormatter = new BinaryFormatter();
```

Then, after you fill a class object with data, you can write it to the output file stream named `outFile` with a statement such as the following:

```
bFormatter.Serialize(outFile, objectFilledWithData);
```

The `Serialize()` method takes two arguments—a reference to the output file and a reference to a serializable object that might contain any number of data fields. The entire object is written to the data file with this single statement.

Similarly, when you read an object from a data file, you use a statement like the following:

```
objectInstance = (TypeOfObject)bFormatter.Deserialize(inFile);
```

This statement uses the `Deserialize()` method with a `BinaryFormatter` object to read in one object from the file. The object is cast to the appropriate type and can be assigned to an instance of the object. Then you can access individual fields. An entire object is read with this single statement, no matter how many data fields it contains.

Figure 14-22 shows a program that writes `Employee` class objects to a file and later reads them from the file into the program. After the `FileStream` is declared for an output file, a `BinaryFormatter` is declared in the first shaded statement. The user enters an ID number, name, and salary for an `Employee`, and the completed object is written to a file in the second shaded statement. When the user enters 999, the output file is closed.

```
using System;
using static System.Console;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization;
class SerializableDemonstration
{
   static void Main()
   {
      const int END = 999;
      const string FILENAME = "Data.ser";
      Employee emp = new Employee();
      FileStream outFile = new FileStream(FILENAME,
         FileMode.Create, FileAccess.Write);
      BinaryFormatter bFormatter = new BinaryFormatter();
      Write("Enter employee number or " + END +
         " to quit >> ");
      emp.EmpNum = Convert.ToInt32(ReadLine());
      while(emp.EmpNum != END)
      {
         Write("Enter last name >> ");
         emp.Name = ReadLine();
         Write("Enter salary >> ");
         emp.Salary = Convert.ToDouble(ReadLine());
```

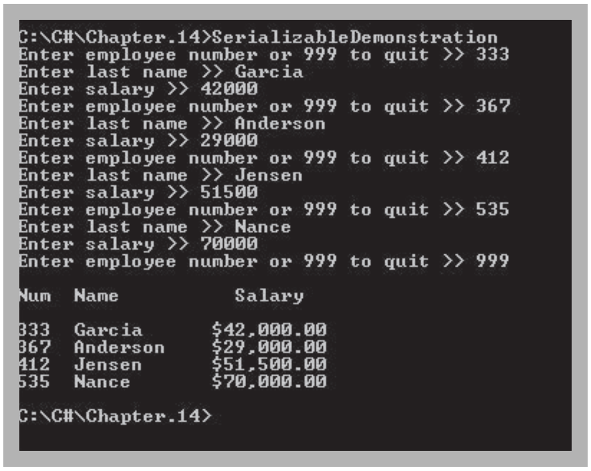
Figure 14-22 The `SerializableDemonstration` program *(continues)*

(continued)

```
            bFormatter.Serialize(outFile, emp);
            Write("Enter employee number or " + END +
               " to quit >> ");
            emp.EmpNum = Convert.ToInt32(ReadLine());
         }
         outFile.Close();
         FileStream inFile = new FileStream(FILENAME,
            FileMode.Open, FileAccess.Read);
         WriteLine("\n{0,-5}{1,-12}{2,8}\n",
            "Num", "Name", "Salary");
         while(inFile.Position < inFile.Length)
         {
            emp = (Employee)bFormatter.Deserialize(inFile);
            WriteLine("{0,-5}{1,-12}{2,8}",
               emp.EmpNum, emp.Name, emp.Salary.ToString("C"));
         }
         inFile.Close();
      }
   }
```

Figure 14-22 The `SerializableDemonstration` program

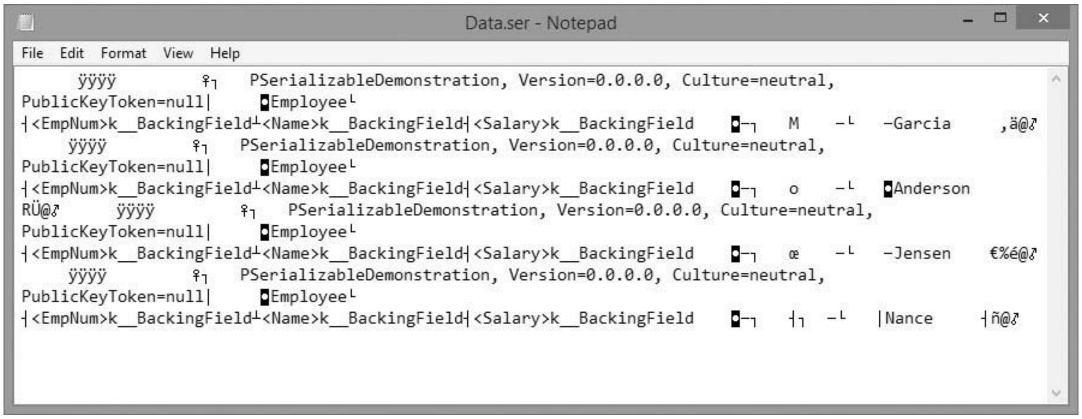
After the output file closes in the SerializableDemonstration program in Figure 14-22, it is reopened for reading. A loop is executed while the `Position` property of the input file is less than its `Length` property. In other words, the loop executes while there is more data in the file. The last shaded statement in the figure deserializes data from the file and casts it to an `Employee` object, where the individual fields can be accessed. Figure 14-23 shows a typical execution of the program.



```
C:\C#\Chapter.14>SerializableDemonstration
Enter employee number or 999 to quit >> 333
Enter last name >> Garcia
Enter salary >> 42000
Enter employee number or 999 to quit >> 367
Enter last name >> Anderson
Enter salary >> 29000
Enter employee number or 999 to quit >> 412
Enter last name >> Jensen
Enter salary >> 51500
Enter employee number or 999 to quit >> 535
Enter last name >> Nance
Enter salary >> 70000
Enter employee number or 999 to quit >> 999

Num  Name          Salary

333  Garcia      $42,000.00
367  Anderson    $29,000.00
412  Jensen      $51,500.00
535  Nance       $70,000.00

C:\C#\Chapter.14>
```

Figure 14-23 Typical execution of the `SerializableDemonstration` program

The file created by the SerializableDemonstration program is not as easy to read as the text file created by the WriteSequentialFile program discussed earlier in the chapter (in Figure 14-11). Figure 14-24 shows the file contents displayed in Notepad (with some newline characters inserted so the output can fit on this page). If you examine the file carefully, you can discern the string names and some `Employee` class information, but the rest of the file is not easy to read.



```
     ÿÿÿÿ          ♀┐   PSerializableDemonstration, Version=0.0.0.0, Culture=neutral,
PublicKeyToken=null|     ◘Employee└
┤<EmpNum>k__BackingField┴<Name>k__BackingField┤<Salary>k__BackingField    ◘─┐   M    ─└   ─Garcia     ,ä@♪
    ÿÿÿÿ          ♀┐   PSerializableDemonstration, Version=0.0.0.0, Culture=neutral,
PublicKeyToken=null|     ◘Employee└
┤<EmpNum>k__BackingField┴<Name>k__BackingField┤<Salary>k__BackingField    ◘─┐   o    ─└   ◘Anderson
RÜ@♪      ÿÿÿÿ          ♀┐   PSerializableDemonstration, Version=0.0.0.0, Culture=neutral,
PublicKeyToken=null|     ◘Employee└
┤<EmpNum>k__BackingField┴<Name>k__BackingField┤<Salary>k__BackingField    ◘─┐   œ    ─└   ─Jensen     €%é@♪
    ÿÿÿÿ          ♀┐   PSerializableDemonstration, Version=0.0.0.0, Culture=neutral,
PublicKeyToken=null|     ◘Employee└
┤<EmpNum>k__BackingField┴<Name>k__BackingField┤<Salary>k__BackingField    ◘─┐   ┤┐  ─└   |Nance     ┤ñ@♪
```

Figure 14-24 Data file created using the `SerializableDemonstration` program

Watch the video *Understanding Serialization and Deserialization*.

TWO TRUTHS & A LIE

Understanding Serialization and Deserialization

1. An advantage of writing data to a text file is that the data is easily readable in a text editor such as Notepad.
2. Serialization is the process of converting objects into streams of bytes. Deserialization is the reverse process; it converts streams of bytes back into objects.
3. By default, all C# classes are serializable.

The false statement is #3. By default, all C# simple data types are serializable, including `strings`. However, if your class contains fields that are more complex data types, you must check the declaration of those classes to ensure they are serializable.

You Do It

Creating a Text File in a GUI Environment

The file writing and reading examples in this chapter have used console applications so that you could concentrate on the features of files in the simplest environment. However, you can write and read files in GUI environments as well. In the next steps, you create two applications. The first allows a user to enter invoice records using a `Form` and to store them in a file. The second application allows a user to view stored records using a `Form`.

1. Open the Visual Studio IDE, and start a new Windows Forms Application project named **EnterInvoices**.
2. Create a `Form` like the one shown in Figure 14-25 by making the following changes:
   - Change the `Text` property of the `Form` to **Invoice Data.**
   - Drag a `Label` onto the `Form`, and change its `Text` property to **Enter invoice data**. Increase the `Label`'s `Font` to 12.
   - Drag three more `Labels` onto the `Form`, and change their `Text` properties to **Invoice number**, **Last name**, and **Amount**, respectively.
   - Drag three `TextBoxes` onto the `Form` next to the three descriptive `Labels`. Change the `Name` properties of the three `TextBoxes` to **`invoiceBox`**, **`nameBox`**, and **`amountBox`**, respectively.
   - Drag a `Button` onto the `Form`, change its `Name` to **`enterButton`**, and change its `Text` to **Enter record**. If necessary, resize **`enterButton`** so all of its text is visible.

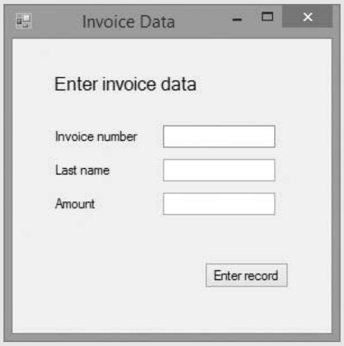


Figure 14-25 Designing the `EnterInvoices` `Form`

3. View the code for the `Form`. At the start of the class, before the `Form1()` constructor, add the shaded code shown in Figure 14-26. The new code contains statements that perform the following:
   - Declare a delimiter that will be used to separate records in the output file.
   - Declare a path and filename. You can change the path if you want to store the file in a different location on your system.

(continues)

(continued)

- Declare variables for the number, name, and amount of each invoice.
- Open the file, and associate it with a `StreamWriter`.

```
namespace EnterInvoices
{
   public partial class Form1 : Form
   {
      const string DELIM = ",";
      const string FILENAME =
         @"C:\C#\Chapter.14\Invoices.txt";
      int num;
      string name;
      double amount;
      static FileStream outFile = new
         FileStream(FILENAME, FileMode.Create,
         FileAccess.Write);
      StreamWriter writer = new StreamWriter(outFile);
      public Form1()
      {
         InitializeComponent();
      }
```

Figure 14-26 Partial code for the `EnterInvoices` program with typed statements shaded

Remember that placing an at sign (@) in front of a string indicates that its characters should be interpreted literally. The @ sign is used with the filename so that the backslashes in the path will not be interpreted as escape sequence characters.

4. At the top of the file, with the other `using` statements, add the following so that the `FileStream` can be declared:

   ```
   using System.IO;
   ```

5. Click **Save All** (and continue to do so periodically as you work). Return to Design view, and double-click the **Enter record** button. As shown in the shaded portions of Figure 14-27, add statements within the method to accept data from each of the three `TextBoxes`, and convert each field to the appropriate type. Then write each field to a text file, separated by delimiting commas. Finally, clear the `TextBox` fields to be ready for the user to enter a new set of data.

(continues)

(continued)

```
private void enterButton_Click(object sender, EventArgs e)
{
   num = Convert.ToInt32(invoiceBox.Text);
   name = nameBox.Text;
   amount = Convert.ToDouble(amountBox.Text);
   writer.WriteLine(num + DELIM + name + DELIM + amount);
   invoiceBox.Clear();
   nameBox.Clear();
   amountBox.Clear();
}
```

Figure 14-27 Code for the `enterButton_Click()` method of the `EnterInvoices` program

6. Locate the `Dispose()` method, which executes when the user clicks the Close button to dismiss the `Form`. A quick way to locate the method in the Visual Studio IDE is to select **Edit** from the main menu, click **Find and Replace**, click **Quick Find**, and type **Dispose** in the dialog box. (The Look in: setting can be either Entire Solution or Current Project.) The method appears on the screen. Add two statements to close `writer` and `outFile`, as shown in the shaded statements in Figure 14-28.

```
protected override void Dispose(bool disposing)
{
   writer.Close();
   outFile.Close();
   if (disposing && (components != null))
   {
      components.Dispose();
   }
   base.Dispose(disposing);
}
```

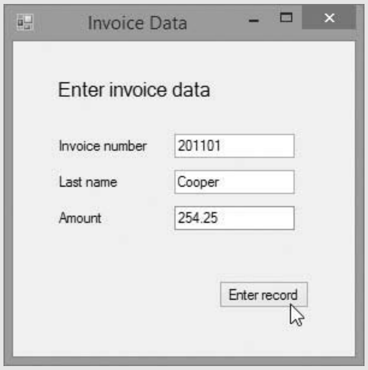
Figure 14-28 The `Dispose()` method in the EnterInvoices program

7. Click **Save All**. Execute the program. When the `Form` appears, enter data in each `TextBox`, and then click the **Enter record** button when you finish.

(continues)

(continued)

The TextBoxes clear in preparation for you to enter another record. Enter at least three records before dismissing the Form. Figure 14-29 shows data entry in progress.



Enter invoice data

Invoice number 201101

Last name Cooper

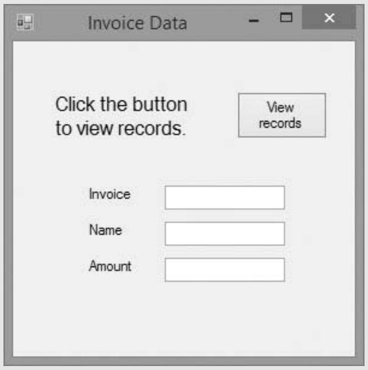
Amount 254.25

Figure 14-29 Entering data in the EnterInvoices application

Reading Data from a Text File into a Form

In the next steps, you create a Form that you can use to read records from a File.

1. Open a new Windows project in Visual Studio, and name it **ViewInvoices**.
2. Create a Form like the one shown in Figure 14-30 by making the following changes:
   - Change the Text of Form1 to **Invoice Data**.
   - Add four Labels with the text, font size, and approximate locations shown in Figure 14-30. (You can click the down arrow next to the Text property of a component to get a box into which you can type multiline text.)
   - Add a Button with the Text **View records**. Resize the Button if necessary. Name the Button **viewButton**.
   - Add three TextBoxes. Name them **invoiceBox**, **nameBox**, and **amountBox**, respectively.
3. In the IDE, double-click the **View records** Button to view the code.



Click the button to view records.

Figure 14-30 The ViewInvoices Form

(continues)

(continued)

4. Add the shaded statements shown in Figure 14-31. They include:

- A `using System.IO` statement
- Constants for the file delimiter character and the filename (change the path for your file if necessary)
- A `string` into which whole records can be read before they are split into fields
- An array of `string`s used to hold the separate, split fields of the entered string
- A `FileStream` and `StreamReader` to handle the input file
- Within the `viewButton_Click()` method, statements to read in a line from the file and split it into three components

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.IO;
namespace ViewInvoices
{
   public partial class Form1 : Form
   {
      const char DELIM = ',';
      const string FILENAME = @"C:\C#\Chapter.14\Invoices.txt";
      string recordIn;
      string[] fields;
      static FileStream file = new FileStream(FILENAME,
         FileMode.Open, FileAccess.Read);
      StreamReader reader = new StreamReader(file);
      public Form1()
      {
         InitializeComponent();
      }
```

Figure 14-31 Partial code for the `ViewInvoices` application *(continues)*

(continues)

(continued)

```
      private void viewButton_Click(object sender, EventArgs e)
       {
          recordIn = reader.ReadLine();
          fields = recordIn.Split(DELIM);
          invoiceBox.Text = fields[0];
          nameBox.Text = fields[1];
          amountBox.Text = fields[2];
       }
    }
}
```

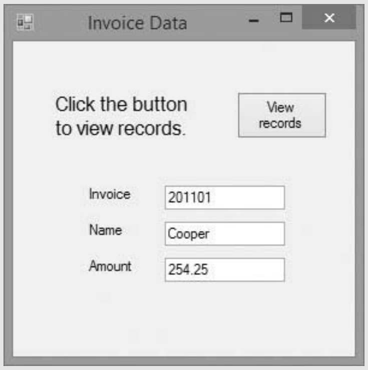
Figure 14-31 Partial code for the `ViewInvoices` application

5. Add two `Close()` statements to the `Dispose()` method in the Form1Designer.cs file, as shown in Figure 14-32.

```
protected override void Dispose(bool disposing)
{
   reader.Close();
   file.Close();
   if (disposing && (components != null))
   {
      components.Dispose();
   }
   base.Dispose(disposing);
}
```

Figure 14-32 The `Dispose()` method for the `ViewInvoices` program

6. Save the project, and then execute it. When the `Form` appears, click the `Button` to view records. You see the data for the first record you entered when you ran the `EnterInvoices` application; your `Form` should look like the one in Figure 14-33. Click the `Button` again to display the next record.



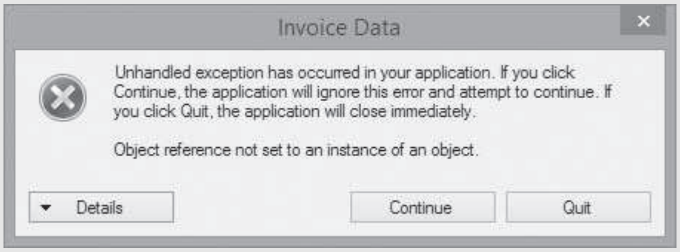
Invoice: 201101
Name: Cooper
Amount: 254.25

Figure 14-33 Typical execution of the `ViewInvoices` program

(continues)

(continued)

7. Continue to click the `Button` to view each record. After you view the last record you entered, click the `Button` again. An unhandled exception is generated, as shown in Figure 14-34, because you attempted to read data past the end of the input file.

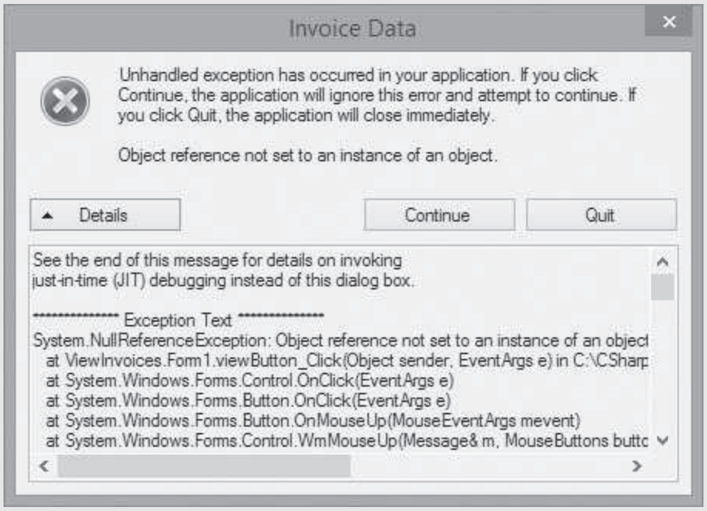


```
Unhandled exception has occurred in your application. If you click
Continue, the application will ignore this error and attempt to continue. If
you click Quit, the application will close immediately.

Object reference not set to an instance of an object.
```

Figure 14-34 Error message window displayed after user attempts to read past the end of the file

Your error message might look different than the one in Figure 14-34 depending on the version of Visual Studio you are using.

8. Click the **Details** button in the `UnhandledException` window to view details of the error. Figure 14-35 shows that a `System.NullReferenceException` was thrown and not handled.



```
Unhandled exception has occurred in your application. If you click
Continue, the application will ignore this error and attempt to continue. If
you click Quit, the application will close immediately.

Object reference not set to an instance of an object.
```

```
See the end of this message for details on invoking
just-in-time (JIT) debugging instead of this dialog box.

************** Exception Text **************
System.NullReferenceException: Object reference not set to an instance of an object
   at ViewInvoices.Form1.viewButton_Click(Object sender, EventArgs e) in C:\CSharp
   at System.Windows.Forms.Control.OnClick(EventArgs e)
   at System.Windows.Forms.Button.OnClick(EventArgs e)
   at System.Windows.Forms.Button.OnMouseUp(MouseEventArgs mevent)
   at System.Windows.Forms.Control.WmMouseUp(Message& m, MouseButtons buttc
```

Figure 14-35 Details displayed by the unhandled exception window

(continues)

(continued)

9. Click **Quit** to close the unhandled exception window.

10. To remedy the unhandled `NullReferenceException` problem, you could take any number of actions. Depending on the application, you might want to do one or more of the following:

    - Display a message.
    - Disallow any more button clicks.
    - End the program.
    - Reposition the file pointer to the file's beginning so the user can view the records again.

    For this example, you take the first two actions: display a message and disallow further button clicks. Return to Visual Studio, and locate the code for the `viewButton_Click()` method. Add a `try…catch` block, as shown in Figure 14-36. Place all the record-handling code in a `try` block, and if a `NullReferenceException` is thrown, change the `Text` in `label1` and disable the View records `Button`.

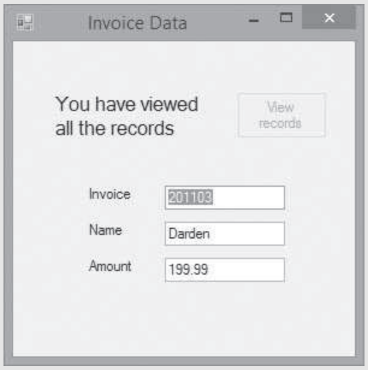
```
private void viewButton_Click(object sender, EventArgs e)
{
   try
   {
      recordIn = reader.ReadLine();
      fields = recordIn.Split(DELIM);
      invoiceBox.Text = fields[0];
      nameBox.Text = fields[1];
      amountBox.Text = fields[2];
   }
   catch (NullReferenceException)
   {
      label1.Text = "You have viewed\nall the records";
      viewButton.Enabled = false;
   }
}
```

Figure 14-36 The `viewButton_Click()` method modified to handle an exception

(continues)

(continued)

11. Save the project, and then execute it. This time, after you have viewed all the available records, the last record remains on the `Form`, an appropriate message is displayed, and the button is disabled, as shown in Figure 14-37.



You have viewed all the records

Invoice 201103

Name Darden

Amount 199.99

Figure 14-37 The `ViewInvoices` `Form` after user has viewed the last record

12. Dismiss the `Form`. Close Visual Studio.

Chapter Summary

- A computer file is a collection of information stored on a nonvolatile, permanent storage device in a computer system. Computer users organize their files into folders or directories. The `File` class contains methods that allow you to access information about files. The `Directory` class provides you with information about directories or folders.
- Data items are stored in a hierarchy of character, field, record, and file. A data file is a sequential access file when each record is read in order from first to last. Usually, the records are stored in order based on the value in a key field. Before an application can use a data file, it must open the file by creating an object and associating a stream of bytes with that object. When you close a file, it is no longer available to your application.
- Bytes flow into and out of applications through streams. When you use the `Console` class, you have access to several standard streams: `Console.In`, `Console.Out`, and

`Console.Error`. When you read from or write to a file, you use a file-processing class instead of `Console`. Many file-processing classes are available in C#, including `StreamReader` for text input from a file, `StreamWriter` for text output to a file, and `FileStream` for both input and output.

- You can use a `StreamWriter` object to write objects to a file using the `WriteLine()` method. Fields should be separated by a delimiter. Data can be read from a file using a `StreamReader` object and the `ReadLine()` method. If the value of the returned string from `ReadLine()` is null, then no more data exists in the file. After a record (line of data) is read in, the `String` class `Split()` method can be used to separate the data fields into an array of strings.
- When you read data from a sequential file, subsequent records are read in order because a file's position pointer holds the byte number of the next byte to be read. To reread a file, you could close it and reopen it, or you can just reposition the file pointer using the `Seek()` method and the `SeekOrigin` enumeration.
- Serialization is the process of converting objects into streams of bytes. Deserialization is the reverse process; it converts streams of bytes back into objects. To create a class that can be serialized, you mark it with the `[Serializable]` attribute. A serializable object can be written to or read from a data file with a single statement.

Key Terms

Random access memory (RAM) is temporary storage in a computer.

Volatile describes storage in which data is lost when power is interrupted.

Nonvolatile storage is permanent storage; it is not lost when a computer loses power.

A **computer file** is a collection of information stored on a nonvolatile device in a computer system.

Permanent storage devices, such as hard disks, USB drives, reels of magnetic tape, and optical discs, are used to store files.

Text files contain data that can be read in a text editor because the data has been encoded using a scheme such as ASCII or Unicode.

Data files contain facts and figures.

Binary files contain data that has been encoded in binary format.

A **byte** is a small unit of storage; in a simple text file, a byte holds only one character.

A **kilobyte** is approximately one thousand bytes.

A **megabyte** is approximately one million bytes.

A **gigabyte** is approximately one billion bytes.

To **read from a file** is to copy data from a file on a storage device into RAM.

To **write to a file** is to store data in a computer file on a permanent storage device.

Persistent storage is nonvolatile storage.

The **root directory** is the main directory of a storage device.

Folders or **directories** are structures used to organize files on a storage device.

A **path** is composed of the disk drive in which a file resides plus the complete hierarchy of directories.

The **data hierarchy** is the relationship of characters, fields, records, and files.

A **character** is any one of the letters, numbers, or other special symbols (such as punctuation marks) that constitute data.

A **character set** is the group of all the characters used to represent data on a particular computer.

A **field** is a character or group of characters that has some meaning.

A **record** is a collection of fields that contain data about an entity.

A **sequential access file** is a data file in which each record is read in order based on its position in the file; usually the records are stored in order based on the value in some field.

The **key field** is the field used to control the order of records in a sequential file.

A **random access file** is one for which records can be accessed in any order.

Opening a file involves creating an object and associating a stream of bytes with it.

Closing a file means it is no longer available to an application.

A **stream** is a pipeline or channel through which bytes are input from and output to a file.

Programmers say `FileStream` **exposes** a stream around a file.

XML is an abbreviation of eXtensible Markup Language, which is a standard for exchanging data over the Internet.

A **token** is a block of text within a string that represents an entity or field.

A **delimiter** is a character used to specify the boundary between characters in text files.

A **CSV file** is one that contains comma-separated values.

A **priming read** is an input statement that gets a first data item or record.

The term **magic number** refers to the bad programming practice of hard-coding numbers in code without explanation.

A file's **file position pointer** holds the byte number of the next byte to be read.

Serialization is the process of converting objects into streams of bytes.

Deserialization is the process of converting streams of bytes back into objects.

Review Questions

1. Random access memory is ______________________.

   a. persistent
   b. volatile
   c. permanent
   d. sequential

2. A collection of data stored on a nonvolatile device in a computer system is a(n) ______________________.

   a. application
   b. computer file
   c. operating system
   d. memory map

3. Which of the following is not permanent storage?

   a. RAM
   b. a hard disk
   c. a USB drive
   d. all of these

4. When you store data in a computer file on a persistent storage device, you are ______________________.

   a. reading
   b. directing
   c. writing
   d. rooting

5. Which of the following is not a `File` class method?

   a. `Create()`
   b. `Delete()`
   c. `Exists()`
   d. `End()`

6. In the data hierarchy, a group of characters that has some meaning, such as a last name or ID number, is a ______________________.

   a. byte
   b. field
   c. file
   d. record

7. When each record in a file is stored in order based on the value in some field, the file is a(n) ______________________file.

   a. random access
   b. application
   c. formatted
   d. sequential

8. A channel through which data flows between a program and storage is a ______________________.

   a. path
   b. folder
   c. stream
   d. directory

9. Which of the following is not part of a `FileStream` constructor?

   a. the file size
   b. the file mode
   c. the filename
   d. the type of access

10. When a file's mode is `Create`, a new file will be created ______________________.

    a. even if one with the same name already exists
    b. only if one with the same name does not already exist
    c. only if one with the same name already exists
    d. only if the access is `Read`

11. Which of the following is not a `FileStream` property?

    a. `CanRead`
    b. `CanExist`
    c. `CanSeek`
    d. `CanWrite`

12. Which of the following is not a file `Access` enumeration?

    a. `Read`
    c. `Write`
    c. `WriteRead`
    d. `ReadWrite`

13. A character used to specify the boundary between data items in text files is a ______________________.

    a. sentinel
    b. stopgap
    c. delimiter
    d. margin

14. Which character can be used to specify a boundary between characters in text files?

    a. a comma
    b. a semicolon
    c. either of these
    d. neither of these

15. After a `StreamReader` has been created, the `ReadLine()` method can be used to ______________________.
    a. retrieve one line at a time from the file
    b. retrieve one character at a time from the file
    c. store one line at a time in a file
    d. split a `string` into tokens

16. The argument to the `String` class `Split()` method is ______________________.
    a. `void`
    b. the number of fields into which to split a record
    c. the character that identifies a new field in a `string`
    d. a `string` that can be split into tokens

17. The `String` class `Split()` method stores its results in ______________________.
    a. a `string`
    b. an array of `strings`
    c. an appropriate data type for each token
    d. an array of bytes

18. A file's ______________________ holds the byte number of the next byte to be read.
    a. index indicator
    b. position pointer
    c. header file
    d. key field

19. The process of converting objects into streams of bytes is ______________________.
    a. extrication
    b. splitting
    c. mapping
    d. serialization

20. Which of the following is serializable?
    a. an `int`
    b. an array of `int`s
    c. a `string`
    d. all of the above

Exercises

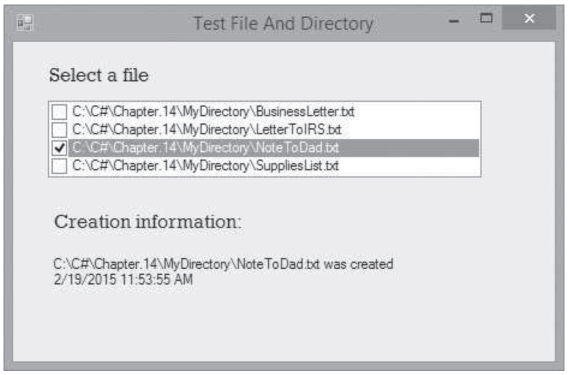
Programming Exercises

1. Create a program named **TestFileAndDirectory** that allows a user to continually enter directory names until the user types *end*. If the directory name exists, display a list of the files in it; otherwise, display a message indicating the directory does not exist. If the directory exists and files are listed, prompt the user to enter one of the filenames. If the file exists, display its creation date and time; otherwise, display a message indicating the file does not exist. Create as many test directories and files as necessary to test your program.

2. Create a program named **FileComparison** that compares two files. First, use a text editor such as Notepad to save your favorite movie quote. Next, copy the file contents, and paste them into a word-processing program such as Word. Then, write the file-comparison application that displays the sizes of the two files as well as the ratio of their sizes to each other. To discover a file's size, you can create a `System.IO.FileInfo` object using statements such as the following, where `FILE_NAME` is a string that contains the name of the file, and `size` has been declared as an integer:

```
FileInfo fileInfo = new FileInfo(FILE_NAME);
size = fileInfo.Length;
```

3. Using Visual Studio, create a `Form` like the one shown in Figure 14-38. Specify a directory on your system, and when the `Form` loads, list the files the directory contains in a `CheckedListBox`. (You first saw an example of a `CheckedListBox` in Chapter 12.) Allow the user to click a file's corresponding check box, and display the file's creation date and time. (Each time the user checks a new filename, display its creation date in place of the original selection.) Save the project as **TestFileAndDirectory2**. Create as many files as necessary to test your program.



Test File And Directory

Select a file

- ☐ C:\C#\Chapter.14\MyDirectory\BusinessLetter.txt
- ☐ C:\C#\Chapter.14\MyDirectory\LetterToIRS.txt
- ✓ C:\C#\Chapter.14\MyDirectory\NoteToDad.txt
- ☐ C:\C#\Chapter.14\MyDirectory\SuppliesList.txt

Creation information:

C:\C#\Chapter.14\MyDirectory\NoteToDad.txt was created
2/19/2015 11:53:55 AM

Figure 14-38 Typical execution of the `TestFileAndDirectory2` program

4. a. Create a program named **WriteCustomerRecords** that allows you to enter data for your business's customers and saves the data to a file. Create a `Customer` class that contains fields for ID number, name, and current balance owed.

   b. Create a program named **ReadCustomerRecords** that reads the file created in Exercise 4a and displays each customer's data on the screen.

   c. Create a program named **FindCustomerRecords** that prompts the user for a customer number, reads the file created in Exercise 4a, and displays data for the specified record.

   d. Create a program named **FindCustomerRecords2** that prompts the user for a minimum balance due, reads the file created in Exercise 4a, and displays all the records containing a balance greater than or equal to the entered value.

5. Create a program named **CustomizeAForm** that includes a `Form` for which a user can select options for the background color, size, and title. Change each feature of the `Form` as the user makes selections. After the user clicks a button to save the form settings, save the color, size, and title as strings to a file, and disable the button.

6. Design a program named **RetrieveCustomizedForm** that includes a `Form` like the one created in Exercise 5, except that instead of saving settings, the `Form`'s `Button` should retrieve the saved settings. When the user clicks the `Button`, read the saved settings from the file created in the CustomizeAForm project, and set the new `Form`'s color, size, and title appropriately.

7. Create a program named **HighScore** containing a `Form` that hosts a game in which the computer randomly selects one of three letters (A, B, or C) and the user tries to guess which letter was selected. At the start of the game, read in the previous high score from a data file. (Create this file to hold *0* the first time the game is played.) Display the previous high score on the `Form` to show the player the score to try to beat. Play continues for 10 rounds. As the player makes each guess, show the player's guess and the computer's choice, and award a point if the player correctly guesses the computer's choice. Keep a running count of the number of correct guesses. After 10 rounds, disable the game controls, and create a file that holds the new high score, which might be the same as before the game or a new higher number. When the player begins a new game, the high score will be displayed on the `Form` as the new score to beat.

Debugging Exercises

1. Each of the following files in the Chapter.14 folder of your downloadable student files has syntax and/or logical errors. In each case, determine the problem, and fix the program. After you correct the errors, save each file using the same filename preceded with *Fixed*. For example, save DebugFourteen1.cs as **FixedDebugFourteen.cs**.
   a. DebugFourteen1.cs
   b. DebugFourteen2.cs
   c. DebugFourteen3.cs
   d. DebugFourteen4.cs

Case Problems

1. In Chapter 11, you created the most recent version of the **GreenvilleRevenue** program, which prompts the user for contestant data for this year's Greenville Idol competition. Now, save all the entered data to a file that is closed when data entry is complete and then reopened and read in, allowing the user to view lists of contestants with requested talent types.
2. In Chapter 11, you created the most recent version of the **MarshallsRevenue** program, which prompts the user for customer data for scheduled mural painting. Now, save all the entered data to a file that is closed when data entry is complete and then reopened and read in, allowing the user to view lists of customer orders for mural types.