# Chapter 17

Contax T / Kodak Tri-X

Champs Elysées – Paris

# File I/O

## Learning Objectives

- *Create and manipulate directories and files*
- *Create and manipulate text files*
- *Make a class serializable by using the Serializable attribute*
- *Serialize/deserialize objects to/from disk with the BinaryFormatter class*
- *Serialize/deserialize objects to/from disk with the XMLSerializer class*
- *Append data to existing files*
- *Properly handle file I/O exceptions*
- *List and describe the contents of the System.IO namespace*
- *Create and read log files using classes from the System.IO.Log namespace*
- *Use FileDialogs to graphically locate and open/save files*

## Introduction

All but the most trivial software applications must preserve their data in some form or another. This chapter shows you how to preserve your application data to local files. These files might be located on a hard drive, a floppy disk, a USB drive, or some other type of media connected to your computer. In most cases, the type of media is of no concern to you because the operating system, and the storage device's driver software, handle the machine-specific details. All you need to know to conduct file Input/Output (I/O) operations is a handful of .NET Framework classes. The operating system does the rest.

You're going to learn a lot of cool things in this chapter, like how to manipulate files and directories, how to serialize and deserialize objects to disk, how to read and write text files, how to perform random access file I/O, how to write log files, and finally, how to use an OpenFileDialog to locate and open files. You will be surprised to learn you can do all these things with only a small handful of classes, structures, and enumerations, most of which are found in the System.IO namespace.

When you finish this chapter, you will have reached an important milestone in your C# programming career — you will be able to write applications that save data to disk. You will find this to be a critical skill to have in your programmer's toolbox.

## Manipulating Directories And Files

In most all cases, data generated by an application and stored on an auxiliary storage device such as a hard disk, is saved as an organized, related collection of data in a structure commonly referred to as a *file*. I say "in most cases" because it is entirely possible to write data to an absolute or random position on a device, depending of course on what type of storage medium you're talking about. (*i.e.,* A disk drive works differently than a tape drive.)

It is the operating system's responsibility to manage the organization, reading, and writing of files. When you add a new storage device to your computer, it must first be formatted in a way that allows the operating system to access its data. The file management services provided by the operating system are part of a set of layered services that make it possible to build complex computing systems, as Figure 17-1 partially illustrates.

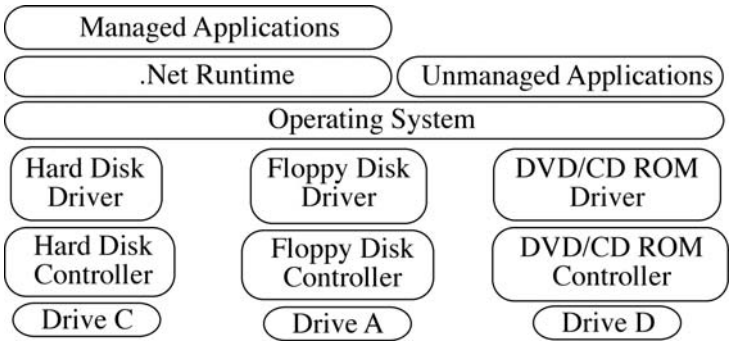


Figure 17-1: Simplified View of Service Layers

Referring to Figure 17-1 — attached storage devices interact with the operating system via an associated software interface referred to as a *driver*. Each device will have its own particular software driver that must be installed and recognized by the operating system before it will work. This applies not only to storage devices but to network cards, display devices, printers, etc. The operating system dictates the rules by which attached storage devices must play, and it is the responsibility of the storage device manufacturer to implement these rules in the device driver.

The operating system makes the services offered by its various device drivers available to running applications. Well-behaved applications target the operating system and do not directly interact with attached storage devices. (**Note:** .NET applications target the .NET runtime environment.)

  

## Files, Directories, And Paths

The Microsoft Windows operating system assigns each attached storage device a letter. On computers with only one hard drive, the letter assigned is 'C' and is referred to as your "C drive". If you have a 3.5 inch floppy drive, its assigned letter is 'A'. The operating system assigns the next available letter to the next available storage device. Thus, if you also have a CD/ROM or DVD drive, its letter will most likely be 'D'. If you plug in a removable USB drive, the operating system will assign to it the letter 'E' for as long as it's attached to the machine.

The file, from the operating system's point of view, is the fundamental storage organizational element. An application's associated data can be stored in one or more files. A file is located in another organizational element called a *directory*. A directory is a special type of file that contains a list of files and directories. A directory contained inside another directory is called a subdirectory. In modern operating systems like Windows or Apple's OS X, the metaphors *folder* and *subfolder* are used to refer to a directory and a subdirectory respectively.

The topmost directory structure on a storage device is referred to as the *root* directory. A particular drive's root directory is indicated by the name of the drive followed by a colon ':', followed by a backward slash character '\'. The root directory of the C drive would be "C:\". Figure 17-2 illustrates these concepts.

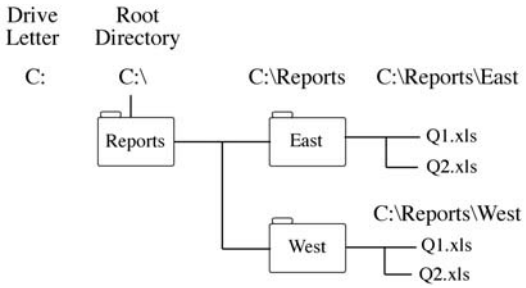


Figure 17-2: Typical Directory Structure

The location of a particular file within a directory structure is indicated by a string of characters called a *path*. The path to the file's location can be *absolute* or *relative*. An absolute path includes the name or letter of the drive and all directory and subdirectory names required to pinpoint the file's location. For example, referring to Figure 17-2 — the absolute path to the Microsoft Excel spreadsheet file named Q2.xls located in the East directory, which is located in the Reports directory, which is located in the root directory of the C drive would be:

"`C:\Reports\East\Q2.xls`".

Figure 17-3 illustrates the concept of an absolute path.

A relative path is the path to a file from some arbitrary starting point, usually a working directory.

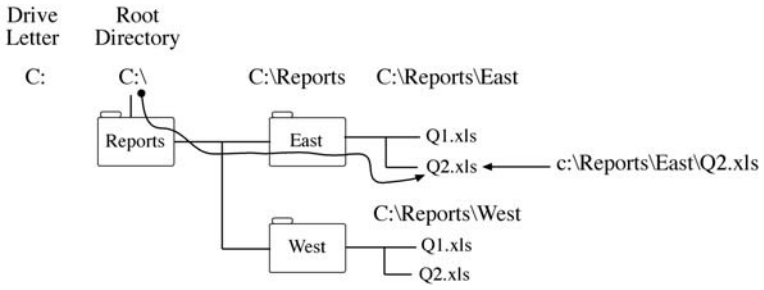


Figure 17-3: The Absolute Path to the Reports\East\Q2.xls File

## Manipulating Directories And Files

You can easily create and manipulate directories and files with the help of several classes provided by the .NET Framework System.IO namespace. These include the *Path*, *File*, *FileInfo*, *Directory*, *DirectoryInfo*, and *DriveInfo* classes. The difference between the Directory/File classes vs. DirectoryInfo/FileInfo classes is that the former are static classes while the latter are non-static, meaning you can create instances of FileInfo and DirectoryInfo. Use the

static class versions when you need to perform one or two operations on a directory or file. If you need to do more robust directory or file processing use the -Info versions.

The use of these classes is fairly straightforward. Example 17.1 offers a short program that prints out information about the current directory, the files it contains, and the drives available on the computer.

*17.1 DirectoryClassDemo.cs*

```
1   using System;
2   using System.IO;
3
4   public class DirectoryClassDemo {
5     public static void Main(){
6       Console.WriteLine("The full path name of the current directory is...");
7       Console.WriteLine("\t" + Directory.GetCurrentDirectory());
8       Console.WriteLine("The current directory has the following files...");
9       String[] files = Directory.GetFiles(Directory.GetCurrentDirectory());
10      foreach(String s in files){
11        FileInfo file = new FileInfo(s);
12        Console.WriteLine("\t" + file.Name);
13      }
14      Console.WriteLine("The computer has the following attached drives...");
15      String[] drives = Directory.GetLogicalDrives();
16      foreach(String s in drives){
17       Console.WriteLine("\t" + s);
18      }
19    }
20  }
```

Referring to Example 17.1 — this example actually demonstrates the use of both the static Directory class and the non-static FileInfo class. On line 7, the Directory.GetCurrentDirectory() method is used to get the absolute path to the current, or working, directory. (*i.e.,* The directory in which the program executes.) On line 9, the Directory.GetFiles() method returns an array of strings representing each of the files in the current working directory. (**Note:** The Directory.GetFileSystemEntries() method would return a string array with the names of all files and directories in the current working directory.)

Given the array of filename strings, the `foreach` statement on line 10 iterates over each entry, creates a new FileInfo object for each filename, and prints its name in the console. You could have simply printed out the array of strings, but that would give you the complete path name of each file. The FileInfo.Name property only returns the name of the file, not its complete path name.

Finally, on line 15, the Directory.GetLogicalDrives() method returns a string array containing the names of all drives connected to the computer. Figure 17-4 shows the results of running this program.

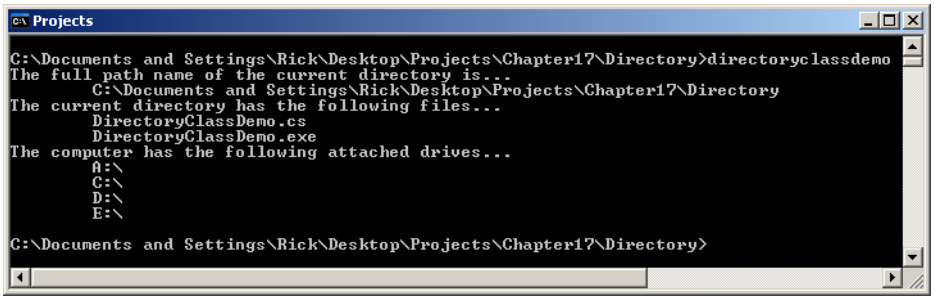


Figure 17-4: Results of Running Example 17.1

## Verbatim String Literals

From now on, you will find it more convenient to use *verbatim string literals* rather than ordinary strings when formulating path names. When using ordinary strings, you must precede special characters with the escape character '\'. For example, a path name formulated as an ordinary string would look like this:

```
String path = "c:\\Reports\\East\\Q1.xls"; //ordinary string
```

Verbatim strings are formulated by preceding the string with the '@' character, which signals the compiler to "...interpret the following string literally, including special characters and line breaks." The path string given above would look like this as a verbatim string:

```
String path = @"c:\Reports\East\Q1.xls"; // verbatim string
```

## Quick Review

In most all cases, data generated by an application and stored on an auxiliary storage device such as a hard disk, is saved as an organized, related collection of data in a structure commonly referred to as a *file*.

It is the operating system's responsibility to manage the organization, reading, and writing of files. When you add a new storage device to your computer, it must first be formatted in a way that allows the operating system to access its data.

The file, from the operating system's point of view, is the fundamental storage organizational element. An application's associated data can be stored in one or more files. A file is located in another organizational element called a *directory*. A directory is a special type of file that contains a list of files and directories. A directory contained inside another directory is called a *subdirectory*.

The topmost directory structure is referred to as the root directory. The root directory of a particular drive is indicated by the name of the drive followed by a colon ':', followed by a backward slash character '\'. The root directory of the C drive would be "C:\".

The location of a particular file within a directory structure is indicated by a string of characters called a *path*. The path to the file's location can be *absolute* or *relative*. An *absolute path* includes the name or letter of the drive and all directory and subdirectory names required to pinpoint the file's location. A *relative path* is the path to a file from some arbitrary starting point, usually a working directory.

You can easily create and manipulate directories and files with the help of several classes provided in the .NET Framework System.IO namespace. These include the *Path*, *File*, *FileInfo*, *Directory*, *DirectoryInfo*, and *DriveInfo* classes.

*Verbatim strings* are formulated by preceding the string with the '@' character which signals the compiler to "...interpret the following string literally, including special characters and line breaks."

# Serializing Objects To Disk

The easiest way to save data to a file is via *serialization*. Serialization is the term used to describe the process of encoding objects in such a way as to facilitate their transmission out of the computer and into or onto some other type of media. Objects can be serialized to disk and then later *deserialized* and reconstituted into objects. The same objects can be serialized for transmission across a network and deserialized at the other end.

While powerful and convenient for you the programmer, serialization is the least flexible way to store data to disk because doing so ties you to the .NET platform. You can't edit the resulting data file. Well, you could edit the file, but because object information is encoded, it's not an ordinary text file, so it's highly likely that you'd screw something up if you did try to edit the file with, say, an ordinary text editor. One way around this is to serialize objects into an XML file.

The nice thing about serialization is that you can serialize single objects, or collections of objects. In this section I will show you how to serialize collections of objects using ordinary serialization with the help of the BinaryFormatter class, and XML serialization with the help of the XMLSerializer class.

## Serializable Attribute

Before any object can be serialized it must be tagged as being serializable. You do this by tagging the class with the Serializable attribute. When dealing with collections of objects, not only must the collection itself be serializable — all the objects contained within the collection must be serializable as well. However, you need not worry about collections, and this includes arrays, as they are already tagged as being serializable. Example 17.2 demonstrates the use of the Serializable attribute to make the Dog class serializable.

*17.2 Dog.cs*

```
1   using System;
2
3   [Serializable]
4   public class Dog {
5
6       private String name = null;
7       private DateTime birthday;
```

```
8
9       public Dog(String name, DateTime birthday){
10           this.name = name;
11           this.birthday = birthday;
12      }
13
14      public Dog():this("Dog Joe", new DateTime(2005,01,01)){ }
15
16      public Dog(String name):this(name, new DateTime(2005,01,01)){ }
17
18
19      public int Age {
20        get {
21        int years = DateTime.Now.Year - birthday.Year;
22          int adjustment = 0;
23        if((DateTime.Now.Month <= birthday.Month) && (DateTime.Now.Day < birthday.Day)){
24          adjustment = 1;
25        }
26        return years - adjustment;
27      }
28      }
29
30      public DateTime Birthday {
31       get {  return birthday; }
32       set {  birthday = value; }
33
34      }
35
36      public String Name {
37       get {  return name; }
38       set {  name = value; }
39      }
40
41
42      public override String ToString(){
43        return (name + "," + Age);
44      }
45
46  } // end class definition
```

Referring to Example 17.2 — the Serializable attribute appears on line 3 just above the start of the class definition in square brackets. That's it! This tells the compiler that instances of the Dog class can be serialized. In the next section I'll show you how to serialize an array of Dog objects with the help of the BinaryFormatter class.

## Serializing Objects With BinaryFormatter

To serialize an object to disk, you'll need to perform the following steps:

Step 1: Create a FileStream object with the name of the file you want to create on disk.

Step 2: Create a BinaryFormatter object and call its Serialize() method, passing in a reference to a FileStream object and a reference to the object you want to serialize.

Deserialization is the opposite of serialization. Deserialization is the process of reconstituting an object that has been previously serialized and turning it back into an object. To deserialize an object from disk, you must perform the following steps:

Step 1: Create a FileStream object that opens the file that contains the object you want to deserialize.

Step 2: Create a BinaryFormatter object and call its Deserialize() method passing in a reference to the FileStream object.

Step 3: The BinaryFormatter.Deserialize() method returns an object. This object must be cast to the appropriate type.

Example 17.3 offers a short program that serializes and deserializes an array of Dog objects. This program depends on the Dog class presented in Example 17.2.

*17.3 MainApp.cs*

```
1   using System;
2   using System.IO;
3   using System.Runtime.Serialization.Formatters.Binary;
4   using System.Runtime.Serialization;
5
6   public class MainApp {
7    public static void Main(String[] args){
8      /*********************************************
9        Create an array of Dogs and populate
```

  

```
10      ***********************************************/
11      Dog[] dog_array = new Dog[3];
12
13      dog_array[0] = new Dog("Rick Miller", new DateTime(1965, 07, 08));
14      dog_array[1] = new Dog("Coralie Powell", new DateTime(1973, 08, 10));
15      dog_array[2] = new Dog("Kyle Miller", new DateTime(1990, 05, 01));
16
17      /*********************************************
18       Iterate over the dog_array and print values
19     *********************************************/
20      Console.WriteLine("-----Original Dog Array Contents----------------------");
21      for(int i = 0; i<dog_array.Length; i++){
22        Console.WriteLine(dog_array[i].Name + ", " + dog_array[i].Age);
23      }
24
25      /*********************************************
26       Serialize the array of dog objects to a file
27     *********************************************/
28      FileStream fs = null;
29      try{
30        fs = new FileStream("DogFile.dat", FileMode.Create);
31        BinaryFormatter bf = new BinaryFormatter();
32        bf.Serialize(fs, dog_array);
33
34      }catch(IOException e){
35        Console.WriteLine(e.Message);
36      }catch(SerializationException se){
37        Console.WriteLine(se.Message);
38      }finally{
39         fs.Close();
40     }
41
42     /*********************************************
43       Deserialize the array of dogs and print values
44     *********************************************/
45         fs = null;                          //start fresh
46         Dog[] another_dog_array = null;     //here too!
47         try{
48           fs = new FileStream("DogFile.dat", FileMode.Open);
49           BinaryFormatter bf = new BinaryFormatter();
50           another_dog_array = (Dog[])bf.Deserialize(fs);
51           Console.WriteLine("-----After Serialization and Deserialization---------");
52           for(int i = 0; i<another_dog_array.Length; i++){
53              Console.WriteLine(another_dog_array[i].Name + ", " + another_dog_array[i].Age);
54           }
55
56         }catch(IOException e){
57
58         Console.WriteLine(e.Message);
59            }catch(SerializationException se){
60               Console.WriteLine(se.Message);
61            }finally{
62               fs.Close();
63       }
64    } // end Main() definition
65  } // end MainApp class definition
```

Referring to Example 17.3 — note the namespaces you must use to serialize objects to disk with a BinaryFormatter. These include System.IO, System.Runtime.Serialization, and System.Runtime.Serialization.Formatters.Binary. The first thing the program does is create an array of Dogs on line 11 and populate it with references to three Dog objects. The `for` loop starting on line 21 iterates over the dog_array and prints each dog's name and age to the console. The serialization process starts on line 28 with the declaration of the FileStream reference named fs. In the body of the `try` block that begins on line 29, the FileStream object is created using the filename "DogFile.dat" and a FileMode of Create. (**Note:** You can name your files anything you like within the rules of the operating system.)

The BinaryFormatter is created on line 31 and on the next line the Serialize() method is called passing in the reference to the FileStream (fs) and the reference to the array of dogs (dog_array). The appropriate exceptions are handled should something go wrong.

The deserialization process begins on line 45 by setting the reference fs to null and creating a completely new array to house the deserialized array of Dog objects. On line 48, a new FileStream object is created given the appropriate file name and a FileMode of Open. A new BinaryFormatter object is created on the following line and its Deserialize() method is called passing in a reference to the FileStream object. Note how the deserialized object is cast to an array of Dogs (*i.e. Dog[]*). The `for` loop on line 52 iterates over another_dog_array and prints each dog's name and age to the console. Figure 17-5 shows the results of running this program.

  

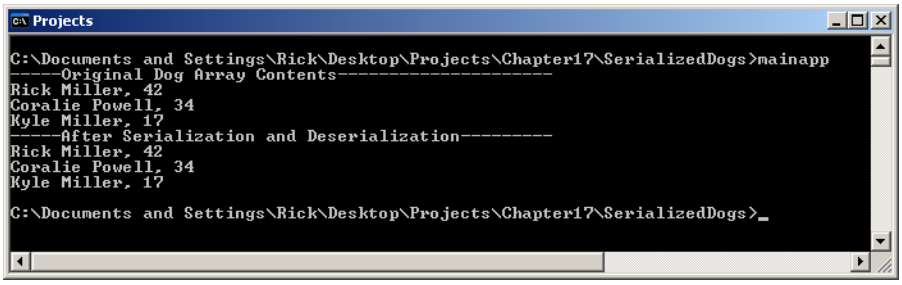


Figure 17-5: Results of Running Example 17.3

## Serializing Objects With XMLSerializer

You can serialize objects to disk in XML format with the help of the XMLSerializer class. The steps required to serialize objects to an XML file are similar to those of ordinary serialization:

Step 1: Create a StreamWriter object passing in the name of the file where you want to save the object.

Step 2: Create an XMLSerializer object and call its Serialize() method passing in a reference to the file and to the object you want to serialize.

To deserialize an XML file you would do the following:

Step 1: Create a FileStream object passing in the name of the file you want to read.

Step 2: Create an XMLSerializer object and call its Deserialize() method.

Step 3: The Deserialize() method returns an object. You must cast this object to the appropriate type.

Example 17.4 gives a modified version of MainApp.cs that serializes an array of Dog objects to disk in an XML file.

*17.4 MainApp.cs (Mod 1)*

```
using System;
using System.IO;
using System.Xml;
using System.Xml.Serialization;

public class MainApp {
 public static void Main(String[] args){
   /*********************************************
     Create an array of Dogs and populate
   *********************************************/
   Dog[] dog_array = new Dog[3];

   dog_array[0] = new Dog("Rick Miller", new DateTime(1965, 07, 08));
   dog_array[1] = new Dog("Coralie Powell", new DateTime(1973, 08, 10));
   dog_array[2] = new Dog("Kyle Miller", new DateTime(1990, 05, 01));

   /*********************************************
    Iterate over the dog_array and print values
   *********************************************/
   Console.WriteLine("-----Original Dog Array Contents----------");
   for(int i = 0; i<dog_array.Length; i++){
     Console.WriteLine(dog_array[i].Name + ", " + dog_array[i].Age);
   }

   /*********************************************
    Serialize the array of dog objects to a file
   *********************************************/
   TextWriter writer = null;
   try{
     writer = new StreamWriter("dogfile.xml");
     XmlSerializer serializer = new XmlSerializer(typeof(Dog[]));
     serializer.Serialize(writer, dog_array);


   }catch(IOException ioe){
     Console.WriteLine(ioe.Message);
   }catch(Exception ex){
     Console.WriteLine(ex.Message);
   }finally{
```

```
40        writer.Close();
41      }
42
43    /*********************************************
44      Deserialize the array of dogs and print values
45    *********************************************/
46        FileStream fs = null;                          //start fresh
47        Dog[] another_dog_array = null;     //here too!
48        try{
49          fs = new FileStream("dogfile.xml", FileMode.Open);
50          XmlSerializer serializer = new XmlSerializer(typeof(Dog[]));
51          another_dog_array = (Dog[])serializer.Deserialize(fs);
52          Console.WriteLine("-----After Serialization and Deserialization---------");
53          for(int i = 0; i<another_dog_array.Length; i++){
54         Console.WriteLine(another_dog_array[i].Name + ", " + another_dog_array[i].Age);
55          }
56
57        }catch(IOException ioe){
58
59        Console.WriteLine(ioe.Message);
60           }catch(Exception ex){
61             Console.WriteLine(ex.Message);
62           }finally{
63              fs.Close();
64      }
65    } // end Main() definition
66  } // end MainApp class definition
```

Referring to Example 17.4 — note now that the namespaces required to serialize objects to an XML file include System.IO, System.XML, and System.XML.Serialization. The serialization process begins on line 28 with the declaration of a TextWriter reference. In the body of the try block, a StreamWriter object is actually created passing in the name of the file that will be used to hold the serialized dog_array. On line 31, an XMLSerializer object is created. Note that what gets passed as an argument to the constructor is the type of object that will be serialized. The Serialize() method is called on the following line passing in the reference to the output file (writer) and the object to be serialized (dog_array).

The deserialization process starts on line 46 with the declaration of the FileStream reference fs. Another dog array is declared named another_dog_array. In the body of the `try` block starting on line 48, the FileStream object is created passing in the name of the input file and a FileMode of Open. Next, an XMLSerializer object is created again passing to its constructor the type of object that will be deserialized. Lastly, the Deserialize() method is called passing in the name of the input file. The resulting object must be cast to the type Array of Dog (Dog[]). The `for` loop then iterates over the contents of another_dog_array and prints the name and age of each dog to the console. Figure 17-6 gives the results of running this program.

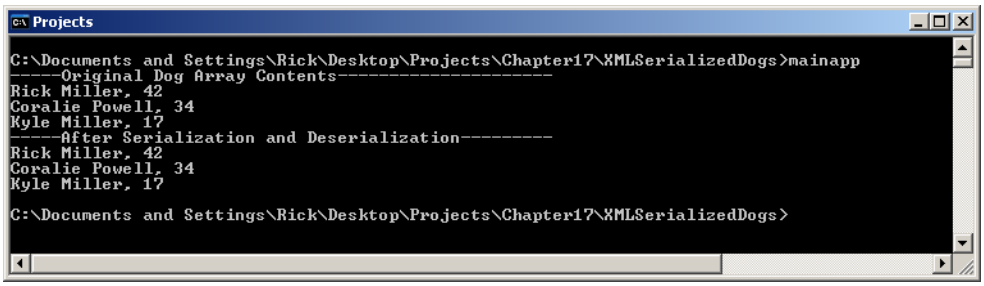


Figure 17-6: Results of Running Example 17.4

At this point you'll find it interesting to explore the contents of both the DogFile.dat and the dogfile.xml files. The DogFile.dat file appears to contain a log of gibberish, while the XML file is a readable text file that contains XML tags corresponding to the object or objects that were serialized. Example 17.5 gives the listing of dogfile.xml.

*17.5 Contents of dogfile.xml*

```
1   <?xml version="1.0" encoding="utf-8"?>
2   <ArrayOfDog xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3                       xmlns:xsd="http://www.w3.org/2001/XMLSchema">
4     <Dog>
5       <Birthday>1965-07-08T00:00:00</Birthday>
6       <Name>Rick Miller</Name>
7     </Dog>
8     <Dog>
9       <Birthday>1973-08-10T00:00:00</Birthday>
10      <Name>Coralie Powell</Name>
```

```
11    </Dog>
12    <Dog>
13      <Birthday>1990-05-01T00:00:00</Birthday>
14      <Name>Kyle Miller</Name>
15    </Dog>
16  </ArrayOfDog>
```

### Quick Review

Object *serialization* provides an easy, convenient way for you to persist application data to disk. Object serialization is also the least flexible way to store application data because you can't edit the resulting file. Use a FileStream object and a BinaryFormatter to serialize objects to disk. Before an object can be serialized it must be tagged as being serializable with the Serializable attribute. Place the Serializable attribute above the class declaration line.

When serializing a collection of objects, remember that all objects contained within the collection must be serializable. You don't have to worry about the collections themselves, including ordinary arrays, as they are already tagged as being serializable.

You can get around the limitation of ordinary serialization by serializing objects to disk in XML format. Use the StreamWriter and XMLSerializer classes to serialize objects to disk in XML format. Use a FileStream and XMLSerializer to deserialize objects from an XML file.

## Working With Text Files

One of the best ways to store data in a way that can be easily shared between different applications or different computer platforms is in a *text file*. The System.IO namespace provides two classes that make it easy to process text files: *StreamReader* and *StreamWriter*. The StreamReader class extends the abstract *TextReader* class; the StreamWriter extends the abstract *TextWriter* class.

### Some Issues You Must Consider

Before you start writing code to process text files, you'll need to spend some time in the design phase working on exactly what format the text within your text file will have. By format I mean how the text is organized within the file. The decisions you make regarding this issue will vary according to your application's data storage needs. For example, a small database application might store records as separate lines of text. These lines may be, and usually are, separated by special characters referred to as *carriage-return/line-feed* (\r\n). Individual fields within each record may by further separated or *delimited* with another type of character. One character that's commonly used to delimit fields is the comma ‘,’.

Another critically important point to consider is, “What data needs to be preserved in the text file?” For example, if you are working with Person objects within your program, and you want to save this data to a file, what data about each Person object must you save to allow the creation of Person objects later when the data is read from the file?

Also, how might the data be treated later in its life? Will it be read by another program? If so, what type of application is it and how will the data's format affect the application's performance.

### Saving Dog Data To A Text File

Example 17.6 offers a short program that saves the data for an array of Dog objects to a text file. After the file is written, the program reads and parses the text file and recreates the array of Dog objects.

*17.6 TextFileDemo.cs*

```
1   using System;
2   using System.IO;
3
4   public class TextFileDemo {
5     public static void Main(){
6      /*******************************************
7          Create an array of Dogs and populate
8      *******************************************/
9      Dog[ ] dog_array = new Dog[ 3];
```

  

```
10
11      dog_array[ 0]  = new Dog("Rick Miller", new DateTime(1965, 07, 08));
12      dog_array[ 1]  = new Dog("Coralie Powell", new DateTime(1973, 08, 10));
13      dog_array[ 2]  = new Dog("Kyle Miller", new DateTime(1990, 05, 01));
14
15      /*******************************************
16       Iterate over the dog_array and print values
17     *******************************************/
18      Console.WriteLine("-----Original Dog Array Contents---------------------");
19      foreach(Dog d in dog_array){
20        Console.WriteLine(d.Name + ", " + d.Age);
21      }
22
23      /*******************************************
24       Save data to textfile
25     *******************************************/
26      TextWriter writer = null;
27      try{
28        writer = new StreamWriter("dogfile.txt");
29        foreach(Dog d in dog_array){
30         writer.WriteLine(d.Name + "," + d.Birthday.Year + "-" + d.Birthday.Month + "-" + d.Birthday.Day);
31        }
32        writer.Flush();
33      }catch(Exception e){
34        Console.WriteLine(e);
35      }finally{
36        writer.Close();
37      }
38
39      /*******************************************
40       Read data from text file and create objects...
41     *******************************************/
42      TextReader reader = null;
43      Dog[ ]  another_dog_array = new Dog[ 3] ;
44      try{
45        reader = new StreamReader("dogfile.txt");
46        String s = String.Empty;
47        int count = 0;
48        while((s = reader.ReadLine()) != null){
49          String[ ]  line = s.Split(',');
50          String name = line[ 0] ;
51          String[ ]  dob = line[ 1] .Split('-');
52          another_dog_array[ count++]  = new Dog(name, new DateTime(Int32.Parse(dob[ 0] ), Int32.Parse(dob[ 1] ),
53                                                                    Int32.Parse(dob[ 2] )));
54        }
55      }catch(Exception e){
56        Console.WriteLine(e);
57      }finally{
58        reader.Close();
59      }
60
61       Console.WriteLine("-----------After writing to and reading from text file------------");
62       foreach(Dog d in another_dog_array){
63          Console.WriteLine(d.Name + ", " + d.Age);
64       }
65
66    } // end Main()
67  } // end class definition
```

Referring to Example 17-6 — the array of Dog reference is created as before and each dog's name and age is printed to the console. The start of the text file save process begins on line 26 with the declaration of the TextWriter reference named writer. In the body of the `try` block, a new StreamWriter is created passing in the name of the file in which to save the Dog object data. (dogfile.txt) The `foreach` loop iterates over each element of the array and calls the writer.WriteLine() method to write each dog's name and birthday information to disk. Note that in this case I am separating the name field from the birthday field with a comma.

To create a DateTime object later when I read the file, I will need to have the year, month, and day of the dog's birthday. I delimit each piece of the birthday with a hyphen '-'. When I have finished writing all the lines, I call the writer.Flush() method to actually write the data to disk.

The file read process begins on line 42 with the declaration of a TextReader reference. In the body of the `try` block, I create a StreamReader object passing in the name of the text file to read. I then process the text file according to the following algorithm:

- Declare a string variable in which will be stored each line as it is read from the text file.
- Declare a count variable to control the process loop.
- Read the next line of the file and if it's not null, process the line like so:

Declare a string array to hold the individual fields of the string when it is split.
Call the String.Split() method to split the line into tokens based on the field delimiter ‘,’.
Create a string variable called “name” and assign to it the first token of the split string.
Create another string array named dob (short for *date of birth*) to hold the split date field.
Call the String.Split() method on the second line token (i.e., line[1]) to split the dob.
Create the Dog object using the extracted fields.

As you can see, there is considerably more work involved with manipulating lines of text files. Figure 17-7 gives the results of running this program. Example 17.7 shows the contents of the dogfile.txt file.

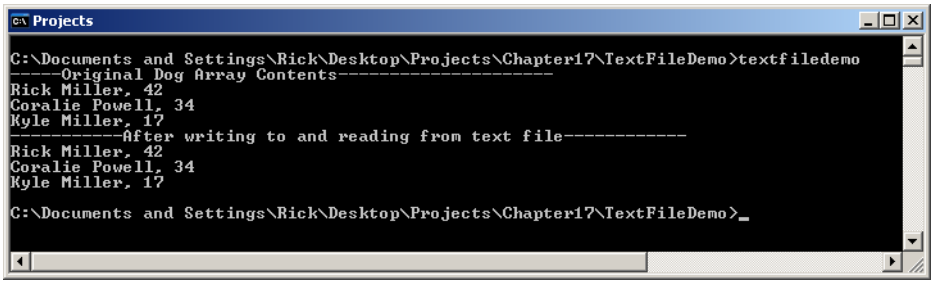


Figure 17-7: Results of Running Example 17.6

*17.7 Contents of dogfile.txt*

```
1   Rick Miller,1965-7-8
2   Coralie Powell,1973-8-10
3   Kyle Miller,1990-5-1
```

## Quick Review

The StreamReader and StreamWriter classes let you read and write text files. Text files are usually processed line-by-line. Lines of text are terminated with the special characters *carriage-return and line-feed* (\r\n). Each line can contain one or more fields *delimited* by some character. The comma ‘,’ is a commonly used field delimiter. Individual fields can be further delimited as required.

Look to the objects in your program to determine the type of information your text file(s) must contain. You’ll need to save enough data to recreate objects.

Process a text file by reading each line and breaking it into tokens with the String.Split() method. If one or more fields are also delimited, use the String.Split() method to tokenize the data as required.

## Working With Binary Data

You can read and write binary data to a file with the help of the *BinaryReader* and *BinaryWriter* classes. The BinaryWriter class provides an overloaded Write() method that is used to write each of the simple types including strings and arrays of bytes and characters. The BinaryReader class provides an assortment of Read*Typename*() methods where *Typename* may be any one of the simple types to include strings and arrays of bytes and characters.

Example 17.8 shows the BinaryWriter and BinaryReader classes in action.

*17.8 BinaryDataDemo.cs*

```
1   using System;
2   using System.IO;
3
4   public class BinaryDataDemo {
5     public static void Main(){
6
7        int record_count = 5;
8        int record_number = 0;
9        int int_val = 125;
10       double double_val = -4567.00;
11       String string_val = "I love C#!";
12       bool bool_val = true;
13
14      /*********************************************************
15        Create the file and write the data with a BinaryWriter
16      *********************************************************/
```

  

```
17        BinaryWriter writer = null;
18         try{
19          writer = new BinaryWriter(File.Open("binaryfile.dat", FileMode.Create));
20          writer.Write(record_count);
21          for(int i=0; i<record_count; i++){
22            writer.Write(++record_number);
23            writer.Write(int_val);
24            writer.Write(double_val);
25            writer.Write(string_val);
26            writer.Write(bool_val);
27          }
28
29         }catch(Exception e){
30           Console.WriteLine(e);
31         }finally{
32           writer.Close();
33         }
34
35         /*******************************************************
36           Open the file and read the data with a BinaryReader
37          *******************************************************/
38        BinaryReader reader = null;
39        record_count = 0; // reset record count
40        try{
41          reader = new BinaryReader(File.Open("binaryfile.dat", FileMode.Open));
42          record_count = reader.ReadInt32();
43          for(int i=0; i<record_count; i++){
44            Console.WriteLine("Record #:     " + reader.ReadInt32());
45            Console.WriteLine("Int value:    " + reader.ReadInt32());
46            Console.WriteLine("Double value: " + reader.ReadDouble());
47            Console.WriteLine("String value: " + reader.ReadString());
48            Console.WriteLine("Bool value:   " + reader.ReadBoolean());
49            Console.WriteLine("-------------------------------------------------------");
50          }
51
52        }catch(Exception e){
53          Console.WriteLine(e);
54        }finally{
55          reader.Close();
56        }
57      } // end Main()
58    } // end class definition
```

Referring to Example 17.8 — on lines 7 through 12 I declare a set of variables of various different types. I use the variable named record_count to indicate the number of records I'll be writing to and reading from the file. The variable named record_number is incremented for each record that is written to the file and will thus be different for each record. The rest of the variables remain unchanged for the duration of the program.

The BinaryWriter reference named writer is declared on line 17 and is used to write the various simple-type variable values to a file named binaryfile.dat. The `for` loop starting on line 21 writes five records to the file. In this case the boundary of each record, or set of binary values, is demarcated only by the combined length of data written to the file during each iteration of the `for` loop. Also, in this case, the combined length of data written to the file with each iteration of the `for` loop is constant because I don't modify the length of the string variable. If I did, then you'd have variable length records.

The BinaryReader reference named reader is declared on line 38 and is used to read the binary values from the file. How does the reader object know where to read? This is where the concept of a *file position pointer* comes into play. The file position pointer is a variable within the reader object that keeps track of the start of the next read location. It is advanced to the next location based on the length of the type that was just read. For example, if you read an integer value, the file position pointer is advanced 4 bytes. If the next value read is a string, the pointer is advanced to a point equal to the length of the string. That's why it's important to know exactly what type you are reading and where in the file you are reading it from. In the case of Example 17.8 above, the `for` loop starting on line 43 simply reads the values from the file in the order in which they were written. Figure 17-8 shows the results of running this program.

## Quick Review

Use the BinaryReader and BinaryWriter classes to read and write binary data to disk. The BinaryWriter class provides an overloaded Write() method that is used to write each of the simple types including strings and arrays of

  

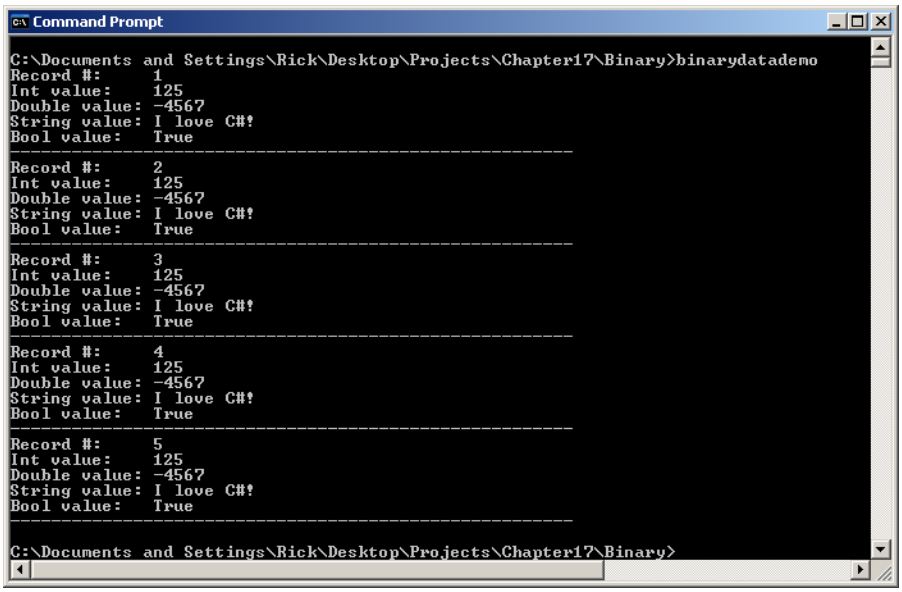


Figure 17-8: Results of Running Example 17.8

bytes and characters. The BinaryReader class provides an assortment of Read*Typename*() methods where *Typename* may be any one of the simple types to include strings and arrays of bytes and characters.

## Random Access File I/O

You can conduct *random access file operations* with the help of the *BinaryReader*, *BinaryWriter*, and *FileStream* classes. The FileStream class provides a Seek() method that allows you to position the file pointer at any point within a file. As you learned in the previous section, the BinaryReader and BinaryWriter classes provide methods for reading and writing binary, string, byte, and character array data.

There are many ways to go about random access file operations, but generally speaking, you must know a little something about how data is organized in a file so that you know where to find what you are looking for. When seeking a specific record location, you must know where one record ends and another begins. This is not the same as reading lines of text where line terminators provide clues as to where one line ends and a new one begins. In most random access file situations, record length is fixed. (*i.e.,* fixed-length records) A fixed-length record can contain a mixture of binary and character data, but each field within the record is a known size. Seeking the location of a particular record within the file requires the setting of the file position pointer value to a multiple of the record length. The number of records a file contains can be calculated by dividing the file length in bytes by the record length in bytes. You could, of course, randomly seek to any position in a file, but who knows what data you will find there!

In this section I'm going to show you a rather extended example of random access file operations. The example code and resulting application provides a solution to the legacy datafile adapter project specification given in Figure 17-9. Please take some time now to review the project specification before proceeding to the next section.

### Towards An Approach To The Adapter Project

Given the project specification and the three supporting artifacts, you may be wondering where to begin. Using the guidance offered by the project-approach strategy in Chapter 1, I recommend devoting some time to studying the schema definition and compare it to what you see in the example data file. You will note that although some of the text appears to read OK, there are a few characters here and there that seem out of place. For instance, you can make out the header information, but the header appears to start with a letter 'z'. Studying the schema definition closely you note that the data file begins with a two-byte file identifier number. But what's the value of this number?

  

```
                                Legacy Datafile Adapter
                                 Project Specification

Objectives:
   - Demonstrate your ability to conduct random access file I/O operations using the BinaryReader, Binary-
     Writer, and FileStream classes
   - Demonstrate your ability to implement a non-trivial interface
   - Demonstrate your ability to translate low-level exceptions into higher-level, user-defined, applica-
     tion-specific exception abstractions
   - Demonstrate your ability to coordinate file I/O operations via object synchronization

Tasks:
   - You are a junior programmer working in the IT department of a retail bookstore. The CEO wants to begin
     migrating legacy systems to the web using .NET technology. A first step in this initiative is to create
     C# adapters to existing legacy data stores. Given an interface definition, example legacy data file,
     and legacy data file schema definition, write a C# class that serves as an adapter object to a legacy
     data file.

Given:
   - C# interface file specifying adapter operations
   - Legacy data file schema definition
   - Example legacy data file


Legacy Data File Schema Definition:

The legacy data file contains three sections:

1) The file identification section is a two-byte value that identifies the file as a data file.

2) The schema description section immediately follows the first section and contains the field text name
   and two-byte field length for each field in the data section.

3) The data section contains fixed-field-length record data elements arranged according to the following
   schema: (length is in bytes)

   Field Name      Length     Description

   --------------|---------|-----------------------------------------------------
   deleted       |    1    |  numeric - 0 if valid, 1 if deleted
   --------------|---------|-----------------------------------------------------
   title         |   50    |  text - book title
   --------------|---------|-----------------------------------------------------
   author        |   50    |  text - author full name
   --------------|---------|-----------------------------------------------------
   pub_code      |    4    |  numeric - publisher code
   --------------|---------|-----------------------------------------------------
   ISBN          |   13    |  text - International Standard Book Number
   --------------|---------|-----------------------------------------------------
   price         |    8    |  text - retail price in following format: $nnnn.nn
   --------------|---------|-----------------------------------------------------
   qoh           |    4    |  numeric - quantity on hand
   --------------|---------|-----------------------------------------------------
```

Figure 17-9: Legacy Datafile Adapter Project Specification

## Start Small And Take Baby Steps

One way to find out is to write a short program that reads the first two bytes of the file and converts it to a number. The BinaryReader class has a method named ReadInt16(). The method name derives from the System.Int16 structure that represents the short data type in the .NET Framework. A short is a two-byte value. The ReadInt16() method would be an excellent method to use to read the first two bytes of the file in an effort to determine their value.

The next phase of your discovery would be to try and read the rest of the file, or at least try and read the complete header and one complete record using the schema definition as a guide. You may find that a more detailed analysis of the header and record lengths are in order. Figure 17-10 shows a simple analysis performed with a spreadsheet.

Referring to Figure 17-10 — the simple analysis reveals that the length of the header section of the legacy data file is 54 bytes long and each record is 130 bytes long. These figures, as well as the individual field lengths, will come in handy when you write the adapter.

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| 2 | Header | Section 1 | Magic Cookie | 2 | | | |
| 3 | | | | | | | |
| 4 | | Section 2 | deleted | 7 | | | |
| 5 | | | *field length* | 2 | | | |
| 6 | | | title | 5 | | | |
| 7 | | | *field length* | 2 | | | |
| 8 | | | author | 6 | | | |
| 9 | | | *field length* | 2 | | | |
| 10 | | | pub_code | 8 | | | |
| 11 | | | *field length* | 2 | | | |
| 12 | | | ISBN | 4 | | | |
| 13 | | | *field length* | 2 | | | |
| 14 | | | price | 5 | | | |
| 15 | | | *field length* | 2 | | | |
| 16 | | | qoh | 3 | | | |
| 17 | | | *field length* | 2 | | | |
| 18 | | | | | | | |
| 19 | | | Total Header Length | 54 | | | |
| 20 | | | | | | | |
| 21 | | | | | | | |
| 22 | Data | | Field Name | Length in Bytes | | Offset Start | Offset End |
| 23 | | | deleted | 1 | | 0 | 1 |
| 24 | | | title | 50 | | 1 | 51 |
| 25 | | | author | 50 | | 51 | 101 |
| 26 | | | pub_code | 4 | | 101 | 105 |
| 27 | | | ISBN | 13 | | 105 | 118 |
| 28 | | | price | 8 | | 118 | 126 |
| 29 | | | qoh | 4 | | 126 | 130 |
| 30 | | | | | | | |
| 31 | | | Total Record Length | 130 | | | |

Figure 17-10: Header and Record Length Analysis

Armed with some knowledge about the structure of the legacy data file and having gained some experience writing a small test program that reads all or portions of the file, you can begin to create the adapter class incrementally. A good method to start with is the ReadRecord() method specified in the LegacyDatafileInterface.

## Other Project Considerations

This section briefly discusses additional issues which must be considered during the project implementation phase. These considerations include 1) record locking during updates and deletes, and 2) translating low-level I/O exceptions into higher level exceptions as specified in the interface.

### *Locking A Record For Updates And Deletes*

The LegacyDatafileInterface specifies that a record must be locked when it is being updated or deleted. The locking is done via a lock token, which is nothing more that a long value. How might the locking mechanism be implemented? How is the lock_token generated?

To implement the locking mechanism, you must thoroughly understand threads and thread synchronization. (These topics are covered in detail in Chapter 16.) An object can be used as a synchronization point by using the C# `lock` keyword or the Monitor.Enter() and Monitor.Exit() methods. The adapter must ensure that if one thread attempts to update or delete a record (by calling the UpdateRecord() or DeleteRecord() methods), it cannot do so while another thread is in the process of calling either of those methods.

You can adopt several strategies as a means to an ends here. You can 1) apply the synchronized attribute to the entire method in question (UpdateRecord() and DeleteRecord()) or 2) control access only to the critical section of code within each method. Within the locked block, you implement logic to check for a particular condition. If the condition holds, you can proceed with whatever it is you need to do. If the condition does not hold, you will have to wait until it does by calling the Monitor.Wait() method. The Wait() method blocks the current thread and adds it to a list of threads waiting to get a lock on that object.

Conversely, when a thread has obtained a lock on an object and it concludes its business and is ready to release the lock, it can notify other waiting threads to wake up by calling the Monitor.Pulse() method. I have used the `lock` keyword along with Monitor.Wait() and Monitor.Pulse() methods to synchronize access to critical code sections within the DatafileAdapter class.

### *Monitor.Enter()/Monitor.Exit() vs. The lock Keyword*

The `lock` keyword is equivalent to the Monitor.Enter()/Monitor.Exit() method combination. You certainly could use the Monitor.Enter()/Monitor.Exit() combination to control access to a critical code section, but you must take measures to ensure the Monitor.Exit() method gets called at some point. To do this, Microsoft recommends that you

  

use them within the body of a `try/finally` block. The `lock` keyword automatically wraps the Monitor.Enter() and Monitor.Exit() methods in a `try/finally` block for you. Figure 17-11 shows you how the use of the Monitor.Enter()/Monitor.Exit() methods compares to the use of the `lock` keyword.

```
try {                                          lock(Object){
    Monitor.Enter(Object);

    // critical code section                      // critical code section


  } catch(ArgumentNullException e){              }
      // handle  appropriately
  } finally {
        Monitor.Exit(Object);
    }
```

Figure 17-11: Monitor.Enter()/Monitor.Exit() vs. the lock Keyword

### Translating Low-Level Exceptions Into Higher-Level Exception Abstractions

The System.IO package defines several low-level exceptions that can occur when conducting file I/O operations. These exceptions must be handled in the adapter, however, the LegacyDatafileInterface specifies that several higher-level exceptions may be thrown when its methods are called.

To create custom exceptions, extend the Exception class and add any customized behavior required. (Exceptions are discussed in detail in Chapter 15.) In your adapter code, you catch and handle the low-level exception when it occurs, repackage the exception within the context of a custom exception, and then throw the custom exception. Any objects utilizing the services of the adapter class must handle your custom exceptions, not the low-level I/O exceptions.

## Where To Go From Here

The previous sections attempted to address some of the development issues you will typically encounter when attempting this type of project. The purpose of the project is to demonstrate the use of the FileStream, BinaryReader, and BinaryWriter classes in the context of a non-trivial example. I hope also that I have sufficiently illustrated the reality that rarely can one class perform its job without the help of many other classes.

The next section gives the code for the completed project. Keep in mind that the examples listed here represent one particular approach and solution to the problem. As an exercise, I will invite you to attempt a solution on your own terms using the knowledge gained here as a guide.

Explore and study the code. Compile the code and observe its operation. Experiment — make changes to areas you feel can use improvement.

## Complete RandomAccessFile Legacy Datafile Adapter Source Code Listing

This section gives the complete listing for the code that satisfies the requirements of the Legacy Datafile Adapter project.

*17.9 FailedRecordCreationException.cs*

```
1   using System;
2
3   public class FailedRecordCreationException : Exception {
4
5       public FailedRecordCreationException() : base("Failed Record Creation Exception") { }
6
7       public FailedRecordCreationException(String message) : base(message) { }
8
9       public FailedRecordCreationException(String message, Exception inner_exception) :
10                                      base(message, inner_exception) { }
    }
```

*17.10 InvalidDataFileExcepton.cs*

```
1   using System;
2
3   public class InvalidDataFileException : Exception {
4
5       public InvalidDataFileException() : base("Invalid Data File Exception") { }
6
7       public InvalidDataFileException(String message) : base(message) { }
8
9       public InvalidDataFileException(String message, Exception inner_exception) :
10                                    base(message, inner_exception) { }
    }
```

*17.11 NewDatafileException.cs*

```
1   using System;
2
3   public class NewDataFileException : Exception {
4
5       public NewDataFileException() : base("New Data File Exception") { }
6
7       public NewDataFileException(String message) : base(message) { }
8
9       public NewDataFileException(String message, Exception inner_exception) :
10                                  base(message, inner_exception) { }
11  }
```

*17.12 RecordNotFoundException.cs*

```
1   using System;
2
3   public class RecordNotFoundException : Exception {
4
5       public RecordNotFoundException() : base("Record Not Found Exception") { }
6
7       public RecordNotFoundException(String message) : base(message) { }
8
9       public RecordNotFoundException(String message, Exception inner_exception) :
10                                    base(message, inner_exception) { }
    }
```

*17.13 SecurityException.cs*

```
1   using System;
2
3   public class SecurityException : Exception {
4
5       public SecurityException() : base("Security Exception") { }
6
7       public SecurityException(String message) : base(message) { }
8
9       public SecurityException(String message, Exception inner_exception) :
10                                base(message, inner_exception) { }
11  }
```

*17.14 LegacyDatafileInterface.cs*

```
1   using System;
2
3   public interface LegacyDatafileInterface {
4
5
6     /// <summary>
7     /// Read the record indicated by the rec_no and return a string array
8     /// were each element contains a field value.
9     /// </summary>
10    /// <param name="rec_no"></param>
11    /// <returns>A string array containing the record fields</returns>
12    /// <exception cref="RecordNotFoundException"</exception>
13    String[] ReadRecord(long rec_no);
14
15
16    /// <summary>
17    /// Update a record's fields. The record must be locked with the lockRecord()
18    /// method and the lock_token must be valid. The value for field n appears in
19    /// element record[n].
20    /// </summary>
21    /// <param name="rec_no"></param>
22    /// <param name="record"></param>
```

```
23    /// <param name="lock_token"></param>
24    /// <exception cref="RecordNotFoundException"></exception>
25    /// <exception cref="SecurityException"></exception>
26    void UpdateRecord(long rec_no, String[] record, long lock_token);
27
28
29    /// <summary>
30    /// Marks a record for deletion by setting the deleted field to 1. The lock_token
31    /// must be valid otherwise a SecurityException is thrown.
32    /// </summary>
33    /// <param name="rec_no"></param>
34    /// <param name="lock_token"></param>
35    /// <exception cref="RecordNotFoundException" ></exception>
36    /// <exception cref="SecurityException"></exception>
37    void DeleteRecord(long rec_no, long lock_token);
38
39    /// <summary>
40    /// Creates a new datafile record and returns the record number.
41    /// </summary>
42    /// <param name="record"></param>
43    /// <returns>The record number of the newly created record</returns>
44    /// <exception cref="FailedRecordCreationException"></exception>
45    long CreateRecord(String[] record);
46
47
48    /// <summary>
49    /// Locks a record for updates and deletes and returns an integer
50    /// representing a lock token.
51    /// </summary>
52    /// <param name="rec_no"></param>
53    /// <returns>Lock token</returns>
54    /// <exception cref="RecordNotFoundException"></exception>
55    long LockRecord(long rec_no);
56
57
58    /// <summary>
59    /// Unlocks a previously locked record. The lock_token must be valid or a
60    /// SecurityException is thrown.
61    /// </summary>
62    /// <param name="rec_no"></param>
63    /// <param name="lock_token"></param>
64    /// <exception cref="SecurityException"></exception>
65    void UnlockRecord(long rec_no, long lock_token);
66
67
68    /// <summary>
69    ///  Searches the records in the datafile for records that match the String
70    /// values of search_criteria. search_criteria[n] contains the search value
71    /// applied against field n.
72    /// </summary>
73    /// <param name="search_criteria"></param>
74    /// <returns>An array of longs containing the matched record numbers</returns>
75    long[] SearchRecords(String[] search_criteria);
76
77  }//end interface definition
```

*17.15 DataFileAdapter.cs*

```
1   using System;
2   using System.IO;
3   using System.Text;
4   using System.Threading;
5   using System.Collections;
6   using System.Collections.Generic;
7
8
9   public class DataFileAdapter : LegacyDatafileInterface {
10
11      /*************************************
12       * Constants
13       *************************************/
14
15      private const short FILE_IDENTIFIER = 378;
16      private const int HEADER_LENGTH = 54;
17      private const int RECORDS_START = 54;
18      private const int RECORD_LENGTH = 130;
19      private const int FIELD_COUNT = 7;
20
21      private const short DELETED_FIELD_LENGTH = 1;
22      private const short TITLE_FIELD_LENGTH = 50;
23      private const short AUTHOR_FIELD_LENGTH = 50;
```

```
24      private const short PUB_CODE_FIELD_LENGTH = 4;
25      private const short ISBN_FIELD_LENGTH = 13;
26      private const short PRICE_FIELD_LENGTH = 8;
27      private const short QOH_FIELD_LENGTH = 4;
28
29      private const String DELETED_STRING = "deleted";
30      private const String TITLE_STRING = "title";
31      private const String AUTHOR_STRING = "author";
32      private const String PUB_CODE_STRING = "pub_code";
33      private const String ISBN_STRING = "ISBN";
34      private const String PRICE_STRING = "price";
35      private const String QOH_STRING = "qoh";
36
37      private const int TITLE_FIELD = 0;
38      private const int AUTHOR_FIELD = 1;
39      private const int PUB_CODE_FIELD = 2;
40      private const int ISBN_FIELD = 3;
41      private const int PRICE_FIELD = 4;
42      private const int QOH_FIELD = 5;
43
44      private const int VALID = 0;
45      private const int DELETED = 1;
46
47      /*****************************************************
48      * Private Instance Fields
49      *****************************************************/
50      private String _filename = null;
51      private BinaryReader _reader = null;
52      private BinaryWriter _writer = null;
53      private long _record_count = 0;
54      private Hashtable _locked_records_map = null;
55      private Random _token_maker = null;
56      private long _current_record_number = 0;
57      private bool _debug = false;
58
59      /*****************************************************
60      * Properties
61      *****************************************************/
62      public long RecordCount {
63         get {  return _record_count; }
64       }
65
66      /*****************************************************
67      * Instance Methods
68      *****************************************************/
69
70      /// <summary>
71      /// Constructor
72      /// </summary>
73      /// <param name="filename"></param>
74      /// <exception cref="InvalidDataFileException"></exception>
75      public DataFileAdapter(String filename) {
76        try {
77           _filename = filename;
78           if(File.Exists(_filename)){
79             _reader = new BinaryReader(File.Open(filename, FileMode.Open));
80             if ((_reader.BaseStream.Length >= HEADER_LENGTH) && (_reader.ReadInt16() == FILE_IDENTIFIER)) {
81                // it's a valid data file
82                Console.WriteLine(_filename + " is a valid data file...");
83                _record_count = ((_reader.BaseStream.Length - HEADER_LENGTH) / RECORD_LENGTH);
84                Console.WriteLine("Record count is: " + _record_count);
85                InitializeVariables();
86                _reader.Close();
87              } else if (_reader.BaseStream.Length == 0) {  // The file's empty - make it a data file
88                     _reader.Close();
89                     WriteHeader(FileMode.Open);
90                     InitializeVariables();
91                   } else {
92                      _reader.BaseStream.Seek(0, SeekOrigin.Begin);
93                      if (_reader.ReadInt16() != FILE_IDENTIFIER) {
94                        _reader.Close();
95                        Console.WriteLine("Invalid data file. Closing file.");
96                        throw new InvalidDataFileException("Invalid data file identifier...");
97                      }
98                   }
99           }else {
100            CreateNewDataFile(_filename);
101          }
102       }catch (ArgumentException e) {
103           if(_debug){  Console.WriteLine(e.ToString()); }
104           throw new InvalidDataFileException("Invalid argument.",e);
```

```
105         }
106         catch (EndOfStreamException e) {
107            if(_debug){ Console.WriteLine(e.ToString()); }
108            throw new InvalidDataFileException("End of stream exception.",e);
109         }
110         catch (ObjectDisposedException e) {
111            if(_debug){ Console.WriteLine(e.ToString()); }
112            throw new InvalidDataFileException("BinaryReader not initialized.",e);
113         }
114         catch (IOException e) {
115            if(_debug){ Console.WriteLine(e.ToString()); }
116            throw new InvalidDataFileException("General IOException",e);
117         }
118         catch (Exception e) {
119            if(_debug){ Console.WriteLine(e.ToString()); }
120            throw new InvalidDataFileException("General Exception",e);
121         }
122         finally {
123           if (_reader != null) {
124              _reader.Close();
125           }
126         }
127      } // end constructor
128
129
130      /// <summary>
131      /// Default Constructor
132      /// </summary>
133      /// <exception cref="InvalidDataFileException"></exception>
134      public DataFileAdapter():this("books.dat"){ }
135
136
137      /// <summary>
138      /// Create new file
139      /// </summary>
140      /// <param name="filename"></param>
141      /// <exception cref="NewDataFileException"></exception>
142      public void CreateNewDataFile(String filename) {
143        try {
144          _filename = filename;
145          WriteHeader(FileMode.Create);
146          InitializeVariables();
147        } catch (Exception e) {
148            if(_debug) { Console.WriteLine(e); }
149            throw new NewDataFileException(e.ToString());
150          }
151      } // end createNewDataFile method
152
153
154      /// <summary>
155      /// Read the record indicated by the rec_no and return a string array
156      /// were each element contains a field value.
157      /// </summary>
158      /// <param name="rec_no"></param>
159      /// <returns>A populated string array containing record field values</returns>
160      /// <exception cref="RecordNotFoundException"></exception>
161      public String[] ReadRecord(long rec_no) {
162        String[] temp_string = null;
163        if ((rec_no < 0) || (rec_no > _record_count)) {
164          if(_debug){ Console.WriteLine("From ReadRecord(): Requested record out of range!"); }
165           throw new RecordNotFoundException("From ReadRecord(): Requested record out of range");
166        } else {
167            try {
168              _reader = new BinaryReader(File.Open(_filename, FileMode.Open));
169              GotoRecordNumber(_reader, rec_no);
170              if (_reader.ReadByte() == DELETED) {
171                if(_debug){ Console.WriteLine("From ReadRecord(): Record number " + rec_no +
172                                              " has been deleted!"); }
173                throw new RecordNotFoundException("Record " + rec_no + " deleted!");
174              } else {
175                  temp_string = RecordBytesToStringArray(_reader, rec_no);
176                }
177            } catch (ArgumentException e) {
178              if(_debug){ Console.WriteLine(e.ToString()); }
179              throw new RecordNotFoundException("Invalid argument.",e);
180           }
181         catch (EndOfStreamException e) {
182            if(_debug){ Console.WriteLine(e.ToString()); }
183            throw new RecordNotFoundException("End of stream exception.",e);
184         }
185         catch (ObjectDisposedException e) {
```

```
186           if(_debug){ Console.WriteLine(e.ToString()); }
187           throw new RecordNotFoundException("BinaryReader not initialized.",e);
188         }
189        catch (IOException e) {
190           if(_debug){ Console.WriteLine(e.ToString()); }
191           throw new RecordNotFoundException("General IOException",e);
192         }
193        catch (Exception e) {
194           if(_debug){ Console.WriteLine(e.ToString()); }
195           throw new RecordNotFoundException("General Exception",e);
196         }
197             finally {
198                 if (_reader != null) {
199                     _reader.Close();
200                 }
201             }
202        } // end else
203       return temp_string;
204     } // end readRecord()
205
206
207     /// <summary>
208     /// Update a record's fields. The record must be locked with the lockRecord()
209     /// method and the lock_token must be valid. The value for field n appears in
210     /// element record[n]. The call to updateRecord() MUST be preceeded by a call
211     /// to lockRecord() and followed by a call to unlockRecord()
212     /// </summary>
213     /// <param name="rec_no"></param>
214     /// <param name="record"></param>
215     /// <param name="lock_token"></param>
216     /// <exception cref="RecordNotFoundException"></exception>
217     /// <exception cref="SecurityException"></exception>
218     public void UpdateRecord(long rec_no, String[] record, long lock_token) {
219       if (lock_token != ((long)_locked_records_map[rec_no])) {
220         if(_debug){ Console.WriteLine("From UpdateRecord(): Invalid  update record lock token."); }
221         throw new SecurityException("From UpdateRecord(): Invalid update record lock token.");
222       } else {
223           try {
224             _writer = new BinaryWriter(File.Open(_filename, FileMode.Open));
225             GotoRecordNumber(_writer, rec_no); //i.e., goto indicated record
226             _writer.Write((byte)0);
227             _writer.Write(StringToPaddedByteField(record[TITLE_FIELD], TITLE_FIELD_LENGTH));
228             _writer.Write(StringToPaddedByteField(record[AUTHOR_FIELD], AUTHOR_FIELD_LENGTH));
229             _writer.Write(Int16.Parse(record[PUB_CODE_FIELD]));
230             _writer.Write(StringToPaddedByteField(record[ISBN_FIELD], ISBN_FIELD_LENGTH));
231             _writer.Write(StringToPaddedByteField(record[PRICE_FIELD], PRICE_FIELD_LENGTH));
232             _writer.Write(Int16.Parse(record[QOH_FIELD]));
233             _current_record_number = rec_no;
234           } catch (ArgumentException e) {
235              if(_debug){ Console.WriteLine(e.ToString()); }
236              throw new RecordNotFoundException("Invalid argument.",e);
237            }
238            catch (EndOfStreamException e) {
239             if(_debug){ Console.WriteLine(e.ToString()); }
240             throw new RecordNotFoundException("End of stream exception.",e);
241            }
242            catch (ObjectDisposedException e) {
243             if(_debug){ Console.WriteLine(e.ToString()); }
244             throw new RecordNotFoundException("BinaryReader not initialized.",e);
245             }
246            catch (IOException e) {
247             if(_debug){ Console.WriteLine(e.ToString()); }
248             throw new RecordNotFoundException("General IOException",e);
249            }
250            catch (Exception e) {
251             if(_debug){ Console.WriteLine(e.ToString()); }
252             throw new RecordNotFoundException("General Exception",e);
253            }
254             finally {
255               if (_writer != null) {
256                 _writer.Close();
257               }
258             }
259         }// end else
260     }// end updateRecord()
261
262
263     /// <summary>
264     /// Marks a record for deletion by setting the deleted field to 1. The lock_token
265     /// must be valid otherwise a SecurityException is thrown.
266     /// </summary>
```

```
267     /// <param name="rec_no"></param>
268     /// <param name="lock_token"></param>
269     /// <exception cref="RecordNotFoundException"></exception>
270     /// <exception cref="SecurityException"></exception>
271     public void DeleteRecord(long rec_no, long lock_token) {
272       if (lock_token != (long)_locked_records_map[rec_no]) {
273         Console.WriteLine("From DeleteRecord(): Invalid delete record lock token.");
274         throw new SecurityException("From DeleteRecord(): Invalid delete record lock token.");
275       } else {
276           try {
277             _writer = new BinaryWriter(File.Open(_filename, FileMode.Open));
278             GotoRecordNumber(_writer, rec_no); // goto record indicated
279             _writer.Write((byte)1);  // mark for deletion
280           } catch (ArgumentException e) {
281             if(_debug){ Console.WriteLine(e.ToString()); }
282             throw new RecordNotFoundException("Invalid argument.",e);
283            }
284            catch (EndOfStreamException e) {
285             if(_debug){ Console.WriteLine(e.ToString()); }
286             throw new RecordNotFoundException("End of stream exception.",e);
287            }
288            catch (ObjectDisposedException e) {
289             if(_debug){ Console.WriteLine(e.ToString()); }
290             throw new RecordNotFoundException("BinaryReader not initialized.",e);
291            }
292            catch (IOException e) {
293             if(_debug){ Console.WriteLine(e.ToString()); }
294             throw new RecordNotFoundException("General IOException",e);
295            }
296           catch (Exception e) {
297             if(_debug){ Console.WriteLine(e.ToString()); }
298             throw new RecordNotFoundException("General Exception",e);
299           }
300             finally {
301               if (_writer != null) {
302                   _writer.Close();
303               }
304             }
305       }// end else
306     }// end deleteRecord()
307
308
309     /// <summary>
310     /// Creates a new datafile record and returns the record number.
311     /// </summary>
312     /// <param name="record"></param>
313     /// <returns> The record number of the newly created record</returns>
314     /// <exception cref="FailedRecordCreationException"></exception>
315     public long CreateRecord(String[] record) {
316       try {
317         _writer = new BinaryWriter(File.Open(_filename, FileMode.Open));
318         GotoRecordNumber(_writer, _record_count); //i.e., goto end of file
319         _writer.Write((byte)0);
320         _writer.Write(StringToPaddedByteField(record[TITLE_FIELD], TITLE_FIELD_LENGTH));
321         _writer.Write(StringToPaddedByteField(record[AUTHOR_FIELD], AUTHOR_FIELD_LENGTH));
322         _writer.Write(Int16.Parse(record[PUB_CODE_FIELD]));
323         _writer.Write(StringToPaddedByteField(record[ISBN_FIELD], ISBN_FIELD_LENGTH));
324         _writer.Write(StringToPaddedByteField(record[PRICE_FIELD], PRICE_FIELD_LENGTH));
325         _writer.Write(Int16.Parse(record[QOH_FIELD]));
326         _current_record_number = ++_record_count;
327       } catch (ArgumentException e) {
328           if(_debug){ Console.WriteLine(e.ToString()); }
329           throw new FailedRecordCreationException("Invalid argument.",e);
330         }
331        catch (EndOfStreamException e) {
332           if(_debug){ Console.WriteLine(e.ToString()); }
333           throw new FailedRecordCreationException("End of stream exception.",e);
334         }
335        catch (ObjectDisposedException e) {
336           if(_debug){ Console.WriteLine(e.ToString()); }
337           throw new FailedRecordCreationException("BinaryReader not initialized.",e);
338         }
339        catch (IOException e) {
340           if(_debug){ Console.WriteLine(e.ToString()); }
341           throw new FailedRecordCreationException("General IOException",e);
342         }
343        catch (Exception e) {
344           if(_debug){ Console.WriteLine(e.ToString()); }
345           throw new FailedRecordCreationException("General Exception",e);
346         }
347          finally {
```

```
348           if (_writer != null) {
349              _writer.Close();
350           }
351         }
352       return _current_record_number;
353     } // end CreateRecord()
354
355
356     /// <summary>
357     /// Locks a record for updates and deletes  - returns an integer
358     /// representing a lock token.
359     /// </summary>
360     /// <param name="rec_no"></param>
361     /// <returns></returns>
362     /// <exception cref="RecordNotFoundException"></exception>
363     public long LockRecord(long rec_no) {
364       long lock_token = 0;
365       if ((rec_no < 0) || (rec_no > _record_count)) {
366         if(_debug){ Console.WriteLine("Record cannot be locked. Not in valid range."); }
367         throw new RecordNotFoundException("Record cannot be locked. Not in valid range.");
368       } else {
369           lock (_locked_records_map) {
370           while (_locked_records_map.ContainsKey(rec_no)) {
371             try {
372               Monitor.Wait(_locked_records_map);
373             } catch (Exception) { }
374           }
375           lock_token = (long)_token_maker.Next();
376           _locked_records_map.Add(rec_no, lock_token);
377           } // end lock
378         } // end else
379       return lock_token;
380     } // end LockRecord()
381
382
383     /// <summary>
384     /// Unlocks a previously locked record. The lock_token must be valid or a
385     /// SecurityException is thrown.
386     /// </summary>
387     /// <param name="rec_no"></param>
388     /// <param name="lock_token"></param>
389     /// <exception cref="SecurityException"></exception>
390     public void UnlockRecord(long rec_no, long lock_token) {
391       lock (_locked_records_map) {
392         if (_locked_records_map.Contains(rec_no)) {
393           if (lock_token == ((long)_locked_records_map[ rec_no])) {
394              _locked_records_map.Remove(rec_no);
395              Monitor.Pulse(_locked_records_map);
396           } else {
397               if(_debug){ Console.WriteLine("From UnlockRecord(): Invalid lock token."); }
398               throw new SecurityException("From UnlockRecord(): Invalid lock token");
399             }
400           } else {
401               if(_debug){ Console.WriteLine("From UnlockRecord(): Invalid record number."); }
402               throw new SecurityException("From UnlockRecord(): Invalid record number.");
403             }
404         }
405     }// end UnlockRecord()
406
407
408     /// <summary>
409     /// Searches the records in the datafile for records that match the String
410     /// values of search_criteria. search_criteria[ n] contains the search value
411     /// applied against field n. Data files can be searched for Title & Author.
412     /// </summary>
413     /// <param name="search_criteria"></param>
414     /// <returns>An array of long values each indicating a record number match</returns>
415     public long[ ] SearchRecords(String[ ] search_criteria) {
416       List<long> hit_list = new List<long>();
417       for (long i = 0; i < _record_count; i++) {
418         try {
419           if (ThereIsAMatch(search_criteria, ReadRecord(i))) {
420              hit_list.Add(i);
421           }
422         } catch (RecordNotFoundException) { } // ignore deleted records
423       } // end for
424       long[ ] hits = new long[ hit_list.Count] ;
425       for (int i = 0; i < hits.Length; i++) {
426         hits[ i] = hit_list[ i] ;
427       }
428       return hits;
```

```
429     } // end SearchRecords()
430
431
432     /// <summary>
433     /// ThereIsAMatch() is a utility method that actually performs
434     /// the record search. Implements an implied OR/AND search by detecting
435     /// the first character of the Title criteria element.
436     /// </summary>
437     /// <param name="search_criteria"></param>
438     /// <param name="record"></param>
439     /// <returns>A boolean value indicating true if there is a match or false otherwise.</returns>
440     private bool ThereIsAMatch(String[] search_criteria, String[] record) {
441       bool match_result = false;
442       int TITLE = 0;
443       int AUTHOR = 1;
444       for (int i = 0; i < search_criteria.Length; i++) {
445         if ((search_criteria[i].Length == 0) || (record[i + 1].StartsWith(search_criteria[i]))) {
446            match_result = true;
447            break;
448         } //end if
449       } //end for
450
451       if (((search_criteria[TITLE].Length > 1) && (search_criteria[AUTHOR].Length >= 1)) &&
452                                                    (search_criteria[TITLE][0] == '&')) {
453          if (record[TITLE + 1].StartsWith(search_criteria[TITLE].Substring(1,
454                                                                search_criteria[TITLE].Length).Trim())
455              && record[AUTHOR + 1].StartsWith(search_criteria[AUTHOR])) {
456                 match_result = true;
457          } else {
458               match_result = false;
459            }
460       } // end  outer if
461       return match_result;
462     } // end thereIsAMatch()
463
464
465     /// <summary>
466     /// GotoRecordNumber - utility function that handles the messy
467     /// details of seeking a particular record.
468     /// </summary>
469     /// <param name="record_number"></param>
470     /// <exception cref="RecordNotFoundException"></exception>
471     private void GotoRecordNumber(BinaryReader reader, long record_number) {
472       if ((record_number < 0) || (record_number > _record_count)) {
473         throw new RecordNotFoundException();
474       } else {
475           try {
476             reader.BaseStream.Seek(RECORDS_START + (record_number * RECORD_LENGTH), SeekOrigin.Begin);
477           } catch (EndOfStreamException e) {
478               if(_debug){ Console.WriteLine(e.ToString()); }
479               throw new RecordNotFoundException("End of stream exception.",e);
480             }
481             catch (ObjectDisposedException e) {
482               if(_debug){ Console.WriteLine(e.ToString()); }
483               throw new RecordNotFoundException("BinaryReader not initialized.",e);
484             }
485             catch (IOException e) {
486               if(_debug){ Console.WriteLine(e.ToString()); }
487               throw new RecordNotFoundException("General IOException",e);
488             }
489             catch (Exception e) {
490               if(_debug){ Console.WriteLine(e.ToString()); }
491               throw new RecordNotFoundException("General Exception",e);
492             }
493        }// end else
494     } // end GotoRecordNumber()
495
496
497     /// <summary>
498     /// GotoRecordNumber - overloaded utility function that handles the messy
499     /// details of seeking a particular record.
500     /// </summary>
501     /// <param name="record_number"></param>
502     /// <exception cref="RecordNotFoundException"></exception>
503     private void GotoRecordNumber(BinaryWriter writer, long record_number) {
504       if ((record_number < 0) || (record_number > _record_count)) {
505         throw new RecordNotFoundException();
506       } else {
507           try {
508             writer.BaseStream.Seek(RECORDS_START + (record_number * RECORD_LENGTH), SeekOrigin.Begin);
509           } catch (EndOfStreamException e) {
```

  

```
510               if(_debug){ Console.WriteLine(e.ToString()); }
511               throw new RecordNotFoundException("End of stream exception.",e);
512             }
513             catch (ObjectDisposedException e) {
514               if(_debug){ Console.WriteLine(e.ToString()); }
515               throw new RecordNotFoundException("BinaryReader not initialized.",e);
516             }
517             catch (IOException e) {
518               if(_debug){ Console.WriteLine(e.ToString()); }
519               throw new RecordNotFoundException("General IOException",e);
520             }
521             catch (Exception e) {
522               if(_debug){ Console.WriteLine(e.ToString()); }
523               throw new RecordNotFoundException("General Exception",e);
524             }
525         }// end else
526     } // end GotoRecordNumber()
527
528
529     /// <summary>
530     /// stringToPaddedByteField - pads the field to maintain fixed
531     /// field length.
532     /// </summary>
533     /// <param name="s"></param>
534     /// <param name="field_length"></param>
535     /// <returns>A populated byte array containing the string value padded with spaces</returns>
536     protected byte[] StringToPaddedByteField(String s, int field_length) {
537       byte[] byte_field = new byte[field_length];
538       if (s.Length <= field_length) {
539          for (int i = 0; i < s.Length; i++) {
540            byte_field[i] = (byte)s[i];
541          }
542          for (int i = s.Length; i < field_length; i++) {
543            byte_field[i] = (byte)' '; //pad the field
544          }
545       } else {
546           for (int i = 0; i < field_length; i++) {
547             byte_field[i] = (byte)s[i];
548           }
549         }
550       return byte_field;
551     } // end StringToPaddedByteField()
552
553
554     /// <summary>
555     /// RecordBytesToStringArray - reads an array of bytes from a data file
556     /// and converts them to an array of Strings. The first element of the
557     /// returned array is the record number. The length of the byte array
558     /// argument is RECORD_LENGTH -1.
559     /// </summary>
560     /// <param name="record_number"></param>
561     /// <returns></returns>
562     private String[] RecordBytesToStringArray(BinaryReader reader, long record_number) {
563       String[] string_array = new String[FIELD_COUNT];
564       char[] title = new char[TITLE_FIELD_LENGTH];
565       char[] author = new char[AUTHOR_FIELD_LENGTH];
566       char[] isbn = new char[ISBN_FIELD_LENGTH];
567       char[] price = new char[PRICE_FIELD_LENGTH];
568       try {
569         string_array[0] = record_number.ToString();
570         reader.Read(title, 0, title.Length);
571         string_array[TITLE_FIELD + 1] = new String(title).Trim();
572         reader.Read(author, 0, author.Length);
573         string_array[AUTHOR_FIELD + 1] = new String(author).Trim();
574         string_array[PUB_CODE_FIELD + 1] = (reader.ReadInt16()).ToString();
575         reader.Read(isbn, 0, isbn.Length);
576         string_array[ISBN_FIELD + 1] = new String(isbn);
577         reader.Read(price, 0, price.Length);
578         string_array[PRICE_FIELD + 1] = new String(price).Trim();
579         string_array[QOH_FIELD + 1] = (reader.ReadInt16()).ToString();
580       } catch (IOException e) {
581           Console.WriteLine(e.ToString());
582         }
583       return string_array;
584     } // end recordBytesToStringArray()
585
586
587     /// <summary>
588     /// Writes the header information into a data file
589     /// </summary>
590     /// <exception cref="InvalidDataFileException"></exception>
```

```
591     private void WriteHeader(FileMode file_mode) {
592       try {
593         if (_writer != null) {
594           _writer.Close();
595         }
596         _writer = new BinaryWriter(File.Open(_filename, file_mode));
597         _writer.Seek(0, SeekOrigin.Begin);
598         _writer.Write(FILE_IDENTIFIER);
599         _writer.Write(DELETED_STRING.ToCharArray());
600         _writer.Write(DELETED_FIELD_LENGTH);
601         _writer.Write(TITLE_STRING.ToCharArray());
602         _writer.Write(TITLE_FIELD_LENGTH);
603         _writer.Write(AUTHOR_STRING.ToCharArray());
604         _writer.Write(AUTHOR_FIELD_LENGTH);
605         _writer.Write(PUB_CODE_STRING.ToCharArray());
606         _writer.Write(PUB_CODE_FIELD_LENGTH);
607         _writer.Write(ISBN_STRING.ToCharArray());
608         _writer.Write(ISBN_FIELD_LENGTH);
609         _writer.Write(PRICE_STRING.ToCharArray());
610         _writer.Write(PRICE_FIELD_LENGTH);
611         _writer.Write(QOH_STRING.ToCharArray());
612         _writer.Write(QOH_FIELD_LENGTH);
613         _writer.Flush();
614       }catch (ArgumentException e) {
615           if(_debug){ Console.WriteLine(e.ToString()); }
616           throw new InvalidDataFileException("Invalid argument.",e);
617        }
618        catch (EndOfStreamException e) {
619           if(_debug){ Console.WriteLine(e.ToString()); }
620           throw new InvalidDataFileException("End of stream exception.",e);
621        }
622        catch (ObjectDisposedException e) {
623           if(_debug){ Console.WriteLine(e.ToString()); }
624           throw new InvalidDataFileException("BinaryReader not initialized.",e);
625        }
626        catch (IOException e) {
627           if(_debug){ Console.WriteLine(e.ToString()); }
628           throw new InvalidDataFileException("General IOException",e);
629        }
630        catch (Exception e) {
631           if(_debug){ Console.WriteLine(e.ToString()); }
632           throw new InvalidDataFileException("General Exception",e);
633        }
634        finally {
635          if (_writer != null) {
636            _writer.Close();
637          }
638        }
639     } // end WriteHeader()
640
641
642     /// <summary>
643     /// readHeader - reads the header bytes and converts them to
644     /// a string
645     /// </summary>
646     /// <returns> A String containing the file header information</returns>
647     /// <exception cref="InvalidDataFileException"></exception>
648     public String ReadHeader() {
649       StringBuilder sb = new StringBuilder();
650       char[] deleted = new char[DELETED_STRING.Length];
651       char[] title = new char[TITLE_STRING.Length];
652       char[] author = new char[AUTHOR_STRING.Length];
653       char[] pub_code = new char[PUB_CODE_STRING.Length];
654       char[] isbn = new char[ISBN_STRING.Length];
655       char[] price = new char[PRICE_STRING.Length];
656       char[] qoh = new char[QOH_STRING.Length];
657       try {
658         _reader = new BinaryReader(File.Open(_filename, FileMode.Open));
659         _reader.BaseStream.Seek(0, SeekOrigin.Begin);
660         sb.Append(_reader.ReadInt16() + " ");
661         _reader.Read(deleted, 0, deleted.Length);
662         sb.Append(new String(deleted) + " ");
663         sb.Append(_reader.ReadInt16() + " ");
664         _reader.Read(title, 0, title.Length);
665         sb.Append(new String(title) + " ");
666         sb.Append((_reader.ReadInt16()) + " ");
667         _reader.Read(author, 0, author.Length);
668         sb.Append(new String(author) + " ");
669         sb.Append((_reader.ReadInt16()) + " ");
670         _reader.Read(pub_code, 0, pub_code.Length);
671         sb.Append(new String(pub_code) + " ");
```

```
672         sb.Append((_reader.ReadInt16()) + " ");
673         _reader.Read(isbn, 0, isbn.Length);
674         sb.Append(new String(isbn) + " ");
675         sb.Append((_reader.ReadInt16()) + " ");
676         _reader.Read(price, 0, price.Length);
677         sb.Append(new String(price) + " ");
678         sb.Append((_reader.ReadInt16()) + " ");
679         _reader.Read(qoh, 0, qoh.Length);
680         sb.Append(new String(qoh) + " ");
681         sb.Append((_reader.ReadInt16()) + " ");
682       } catch (ArgumentException e) {
683           if(_debug){ Console.WriteLine(e.ToString()); }
684           throw new InvalidDataFileException("Invalid argument.",e);
685         }
686         catch (EndOfStreamException e) {
687           if(_debug){ Console.WriteLine(e.ToString()); }
688           throw new InvalidDataFileException("End of stream exception.",e);
689         }
690         catch (ObjectDisposedException e) {
691           if(_debug){ Console.WriteLine(e.ToString()); }
692           throw new InvalidDataFileException("BinaryReader not initialized.",e);
693         }
694         catch (IOException e) {
695           if(_debug){ Console.WriteLine(e.ToString()); }
696           throw new InvalidDataFileException("General IOException",e);
697         }
698         catch (Exception e) {
699           if(_debug){ Console.WriteLine(e.ToString()); }
700           throw new InvalidDataFileException("General Exception",e);
701         }
702         finally {
703           if (_reader != null) {
704             _reader.Close();
705           }
706         }
707       return sb.ToString();
708     } // end ReadHeader()
709
710
711     /// <summary>
712     /// Utility method used to initialize several important instance fields
713     /// </summary>
714     private void InitializeVariables() {
715       _current_record_number = 0;
716       _locked_records_map = new Hashtable();
717       _token_maker = new Random();
718     }
719
720 } // end DataFileAdapter class definition
```

*17.16 AdapterTestApp.cs*

```
1   using System;
2
3   public class AdapterTesterApp {
4      public static void Main(){
5        try{
6          DataFileAdapter adapter = new DataFileAdapter("books.dat");
7          String[] rec_1 = {"C++ For Artists", "Rick Miller", "0001", "1-932504-02-8", "$59.95", "80"};
8          String[] rec_2 = {"Java For Artists", "Rick Miller", "0002", "1-932504-04-X", "$69.95", "100"};
9          String[] rec_3 = {  "C# For Artists", "Rick Miller", "0003", "1-932504-07-9", "$76.00", "567" };
10         String[] rec_4 = {  "White Saturn", "Rick Miller", "0004", "1-932504-08-7", "$45.00", "234" };
11
12         String[] search_string = {"Java", " "};
13
14         String[] temp_string = null;
15
16         adapter.CreateRecord(rec_1);
17         adapter.CreateRecord(rec_2);
18         adapter.CreateRecord(rec_3);
19         adapter.CreateRecord(rec_1);
20         adapter.CreateRecord(rec_2);
21         adapter.CreateRecord(rec_3);
22         adapter.CreateRecord(rec_1);
23         adapter.CreateRecord(rec_2);
24         adapter.CreateRecord(rec_3);
25
26
27         long lock_token = adapter.LockRecord(2);
```

```
28
29          adapter.UpdateRecord(2, rec_2, lock_token);
30          adapter.UnlockRecord(2, lock_token);
31
32          lock_token = adapter.LockRecord(1);
33          adapter.DeleteRecord(1, lock_token);
34          adapter.UnlockRecord(1, lock_token);
35
36          lock_token = adapter.LockRecord(4);
37          adapter.UpdateRecord(4, rec_4, lock_token);
38          adapter.UnlockRecord(4, lock_token);
39
40          long[] search_hits = adapter.SearchRecords(search_string);
41
42          Console.WriteLine(adapter.ReadHeader());
43
44          for(int i=0; i<search_hits.Length; i++){
45            try{
46            temp_string = adapter.ReadRecord(search_hits[i]);
47            for(int j = 0; j<temp_string.Length; j++){
48             Console.Write(temp_string[j] + " ");
49            }
50            Console.WriteLine();
51               }catch(RecordNotFoundException){ }
52          }
53
54          Console.WriteLine("----------------------------------------");
55          for (int i = 0; i < adapter.RecordCount; i++) {
56            try {
57              temp_string = adapter.ReadRecord(i);
58              for (int j = 0; j < temp_string.Length; j++) {
59                Console.Write(temp_string[j] + " ");
60              }
61              Console.WriteLine();
62            }
63            catch (RecordNotFoundException) {  }
64          }
65        }
66        catch (Exception e) { Console.WriteLine(e.ToString()); }
67      } // end Main()
68  } // end class definition
```

Figure 17-11 shows the results of running the AdapterTestApp program one time. Running it several times back-to-back results in additional records being inserted into the book.dat data file.

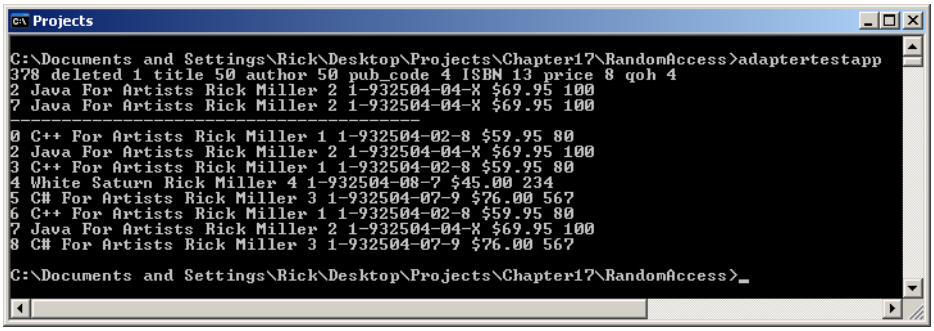


Figure 17-12: Results of Running Example 17.16 Once

## Quick Review

You can conduct random access file I/O with the BinaryReader, BinaryWriter, and FileStream classes. The FileStream class provides a Seek() method that allows you to position the file pointer at any point within a file. As you learned in the previous section, the BinaryReader and BinaryWriter classes provide methods for reading and writing binary, string, byte, and character array data.

## Working With Log Files

The System.IO.Log namespace contains classes, structures, interfaces, and enumerations designed to help you create robust event logging services for your programs. Some of the functionality provided by the contents of the System.IO.Log namespace is only available on Microsoft Windows 2003r2 and Windows Vista or later operating systems. These operating systems come with the Common Log File System (CLFS).

The following three examples together implement a simple logging system. It consists of three classes. The first, LogEntry, given in Example 17.17, represents the type of data that will be saved in the log file. The second, Logger, given in Example 17.18, implements the logging functionality with the help of several classes in the System.IO.Log namespace. The third class, LoggerTestApp, given in Example 17.19, tests the Logger class by writing several entries to the log and then reading the log and writing its contents to the console.

*17.17 LogEntry.cs*

```
1   using System;
2
3   [Serializable]
4   public class LogEntry {
5     private string _subsystem;
6     private int _severity;
7     private string _text;
8     private DateTime _timestamp;
9
10    public DateTime TimeStamp {
11      get { return _timestamp; }
12      set { _timestamp = value; }
13    }
14
15    public string SubSystem {
16      get { return _subsystem; }
17      set { _subsystem = value; }
18    }
19
20    public int Severity {
21      get { return _severity; }
22      set { _severity = value; }
23    }
24
25    public string Text {
26      get { return _text; }
27      set { _text = value; }
28    }
29
30    public LogEntry(DateTime timestamp, string subsystem, int severity, string text){
31      TimeStamp = timestamp;
32      SubSystem = subsystem;
33      Severity = severity;
34      Text = text;
35    }
36
37    public override String ToString(){
38      return TimeStamp.ToString() + " " + SubSystem + " " + Severity + " " + Text;
39    }
40  } // end LogEntry class definition
```

Referring to Example 17-17 — the LogEntry class represents the data that will be captured and written to the log. A log entry will contain a TimeStamp property indicating when the event occurred, a SubSystem property indicating the subsystem of origin, Severity property indicating the severity of the event, and a Text property that contains the string with a detailed description of the event.

*17.18 Logger.cs*

```
1   using System;
2   using System.IO;
3   using System.IO.Log;
4   using System.Collections.Generic;
5   using System.Text;
6   using System.Runtime.Serialization.Formatters.Binary;
7
8   public class Logger {
9     private string _logfilename;
10    private FileRecordSequence _sequence;
11    private SequenceNumber _previous;
12
13
14    public Logger(string logfilename){
```

  

```
15      _logfilename = logfilename;
16      _sequence = new FileRecordSequence(logfilename, FileAccess.ReadWrite);
17      _previous = SequenceNumber.Invalid;
18    }
19
20    public Logger():this("logfile.log"){ }
21
22    public void Append(LogEntry entry){
23      _previous = _sequence.Append(ToArraySegment(entry), SequenceNumber.Invalid,
24                                   _previous, RecordAppendOptions.ForceFlush);
25    }
26
27    public ArraySegment<byte> ToArraySegment(LogEntry entry) {
28      MemoryStream stream = new MemoryStream();
29      BinaryFormatter formatter = new BinaryFormatter();
30      formatter.Serialize(stream, entry);
31      stream.Flush();
32      return new ArraySegment<byte>(stream.GetBuffer());
33    }
34
35    public String GetLogRecords() {
36      StringBuilder sb = new StringBuilder();
37      BinaryFormatter formatter = new BinaryFormatter();
38      IEnumerable<LogRecord> records = _sequence.ReadLogRecords(_sequence.BaseSequenceNumber,
39                                                                LogRecordEnumeratorType.Next);
40      foreach (LogRecord record in records) {
41        LogEntry entry = (LogEntry) formatter.Deserialize(record.Data);
42        sb.Append(entry.ToString() + "\r\n");
43      }
44      return sb.ToString();
45    }
46
47    public void Dispose(){
48      _sequence.Dispose();
49    }
50  } // end class definition
```

Referring to the Example 17.18 — note that the Logger class uses a host of classes found in other namespaces. From the System.IO.Log namespace it uses the FileRecordSequence and SequenceNumber classes. The FileRecordSequece represents a sequence of log records stored in a simple file. SequenceNumbers are not numbers per se. They represent unique pointers from one log entry to the next within a sequence of log entries.

The Logger.Append() method on line 22 takes a LogEntry reference and in turn calls the FileRecordSequence.Append() method, which actually does the heavy lifting. The FileRecordSequence.Append() method has several overloaded variations. The one I use here requires that the log data being written be presented to it in the first argument as an array segment of bytes. *(i.e.*, ArraySegment<byte>) You'll find the ArraySegment generic structure in the System namespace. The Logger.ToArraySegment() method beginning on line 27 does the dirty work of converting a LogEntry object to a ArraySegment<byte> object.

The Logger.GetLogRecords() method on line 35 uses the FileRecordSequence.ReadLogRecords() method to read the records, converts them back into LogEntry objects, appends their string representation to a StringBuilder object, and ultimately returns the whole lot of them as one long string.

*17.19 LoggerTestApp.cs*

```
1   using System;
2   using System.Collections.Generic;
3   using System.Text;
4
5    public class LoggeTestApp {
6      static void Main(string[] args) {
7        Logger logger = new Logger();
8        LogEntry entry1 = new LogEntry(DateTime.Now, "Reactor Coolant", 3, "Main coolant pump speed limited");
9        LogEntry entry2 = new LogEntry(DateTime.Now, "Main Engine", 3, "Main condenser loss of vacuum");
10       LogEntry entry3 = new LogEntry(DateTime.Now, "Reactor Coolant", 3, "Main coolant pump speed limited");
11       LogEntry entry4 = new LogEntry(DateTime.Now, "Reactor", 1, "Loss of control rod control");
12       LogEntry entry5 = new LogEntry(DateTime.Now, "Reactor Coolant", 3, "Main coolant pump speed limited");
13
14       logger.Append(entry1);
15       logger.Append(entry2);
16       logger.Append(entry3);
17       logger.Append(entry4);
18       logger.Append(entry5);
19       Console.Write(logger.GetLogRecords());
20       logger.Dispose();
21     } // end Main()
22  } // end class definition
```

Referring to Example 17.19 — the LoggerTestApp creates five LogEntry objects and calls the Logger.Append() method to insert each entry into the log. It then calls the Logger.GetLogRecords() and prints the results to the console.

To compile this program on Windows XP you'll need to do a couple of things. First, you'll need to have installed the .NET Framework 3.0 Redistributable. Second, locate the System.IO.Log.dll in the C:\Program Files\Reference Assemblies\Microsoft\Framework\v3.0 directory and add this path to your path environment variable. (See Creating Environment Variables in Chapter 2.) Once you set your path you'll need to compile the source files with the `/reference` switch to compile the files along with the System.IO.Log.dll like so:

```
csc /r:System.IO.Log.dll *.cs
```

Figure 17-13 shows the results of running the LoggerTestApp program one time. Running the program multiple times results in repeated log entries.

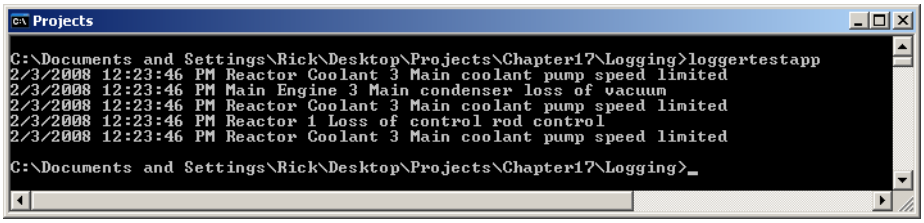


Figure 17-13: Results of Running Example 17.19

## Quick Review

The System.IO.Log namespace contains classes, structures, interfaces, and enumerations designed to help you create robust event logging services for your programs. Some of the functionality provided by the contents of the System.IO.Log namespace is only available on Microsoft Windows 2003r2 and Windows Vista or later operating systems. These operating systems come with the Common Log File System (CLFS).

## Using FileDialogs

As you know by now, the .NET Framework provides a large collection of GUI components that make programming rich graphical user interfaces relatively painless. Most of these classes can be found in the System.Windows.Forms namespace. Two of those classes: OpenFileDialog and SaveFileDialog make it easy to graphically select and open or save files. The following example uses the OpenFileDialog class to select one or more files to open and display several file properties in a TextBox. The example consists of two classes: GUI and MainApp.cs.

*17.20 GUI.cs*

```
1   using System;
2   using System.Windows.Forms;
3   using System.Drawing;
4
5   public class GUI : Form {
6
7     private SplitContainer _splitContainer1;
8     private TextBox _textBox1;
9     private Button _button1;
10
11    public String TextBoxText {
12      get { return _textBox1.Text;  }
13      set { _textBox1.Text = value; }
14    }
15
16    public GUI(MainApp ma){
17       this.InitializeComponent(ma);
18    }
19
20    private void InitializeComponent(MainApp ma) {
21      _splitContainer1 = new SplitContainer();
22      _textBox1 = new TextBox();
23      _button1 = new Button();
24      _splitContainer1.Panel1.SuspendLayout();
25      _splitContainer1.Panel2.SuspendLayout();
```

```
26      _splitContainer1.SuspendLayout();
27      this.SuspendLayout();
28
29      _splitContainer1.Dock = DockStyle.Fill;
30      _splitContainer1.Location = new Point(0, 0);
31      _splitContainer1.Panel1.Controls.Add(_textBox1);
32      _splitContainer1.Panel2.Controls.Add(_button1);
33      _splitContainer1.Size = new Size(292, 273);
34      _splitContainer1.SplitterDistance = 161;
35      _splitContainer1.TabIndex = 0;
36
37      _textBox1.Location = new Point(3, 3);
38      _textBox1.AutoSize = true;
39      _textBox1.Anchor = (AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right);
40      _textBox1.Multiline = true;
41      _textBox1.Name = "textBox1";
42      _textBox1.Size = new Size(155, 267);
43      _textBox1.TabIndex = 0;
44
45      _button1.Location = new Point(27, 12);
46      _button1.Size = new System.Drawing.Size(75, 23);
47      _button1.TabIndex = 0;
48      _button1.Text = "Open File";
49      _button1.UseVisualStyleBackColor = true;
50      _button1.Click += new System.EventHandler(ma.Button1_Click);
51
52      this.AutoScaleMode = AutoScaleMode.None;
53      this.ClientSize = new System.Drawing.Size(292, 273);
54      this.Controls.Add(_splitContainer1);
55
56      this.Text = "FileDialog Demo";
57      _splitContainer1.Panel1.ResumeLayout(false);
58      _splitContainer1.Panel1.PerformLayout();
59      _splitContainer1.Panel2.ResumeLayout(false);
60      _splitContainer1.ResumeLayout(false);
61      this.ResumeLayout(false);
62    } // End InitializeComponent()
63  } // End class definition
```

Referring to Example 17.20 — the GUI class inherits from Form and uses a SplitContainer to hold a TextBox and a Button. The TextBox.MultiLine property is set to true and its Anchor property is set to anchor to all four sides of its containing panel. The button's Click event is set to invoke the MainApp.Button1_Click() method.

*17.21 MainApp.cs*

```
1   using System;
2   using System.Windows.Forms;
3   using System.Text;
4   using System.IO;
5
6   public class MainApp {
7     private OpenFileDialog _fileDialog;
8     private GUI _gui;
9
10    public MainApp(){
11      _gui = new GUI(this);
12      _fileDialog = new OpenFileDialog();
13      _fileDialog.Multiselect = true;
14      Application.Run(_gui);
15    }
16
17    public void Button1_Click(Object o, EventArgs e){
18      _fileDialog.ShowDialog();
19      String[] filenames = _fileDialog.FileNames;
20      StringBuilder sb = new StringBuilder();
21      foreach(String s in filenames){
22        FileInfo file = new FileInfo(s);
23        sb.Append("FileName:" +  file.Name + "\r\n");
24        sb.Append("Directory:" + file.DirectoryName + "\r\n");
25        sb.Append("Size:" + file.Length + " Bytes\r\n");
26        sb.Append("\r\n");
27      }
28      _gui.TextBoxText = sb.ToString();
29    }
30
31    public static void Main(){
32      new MainApp();
33    }
34  } // end class definition
```

  

Referring to Example 17.21 — the MainApp class plays host to the Main() method and the Button1_Click() event handler method. In the body of the MainApp constructor the OpenFileDialog object is created and it's Multiselect property is set to true. This allows the user to select multiple files to open at the same time.

When the button is clicked in the GUI, the Button1_Click() event handler method calls the OpenFileDialog's ShowDialog() method. This displays the dialog and lets users select the file(s) they wish to open. At this point the program effectively blocks until the user clicks the Open button on the OpenFileDialog window.

The OpenFileDialog.FileNames property returns a string array containing the names of the file(s) selected by the user. The `foreach` statement starting on line 21 iterates over each filename, creates a FileInfo object, extracts the required information about each file, and appends it to a StringBuilder object. When the `foreach` statement finishes, the file information contained in the StringBuilder object is written to the GUI.TextBoxText property, which in turn sets its TextBox's Text property.

Figure 17-14 shows the results of running this program and selecting three files named GUI.cs, MainApp.cs, and MainApp.exe. Your results will differ depending on what files you select.

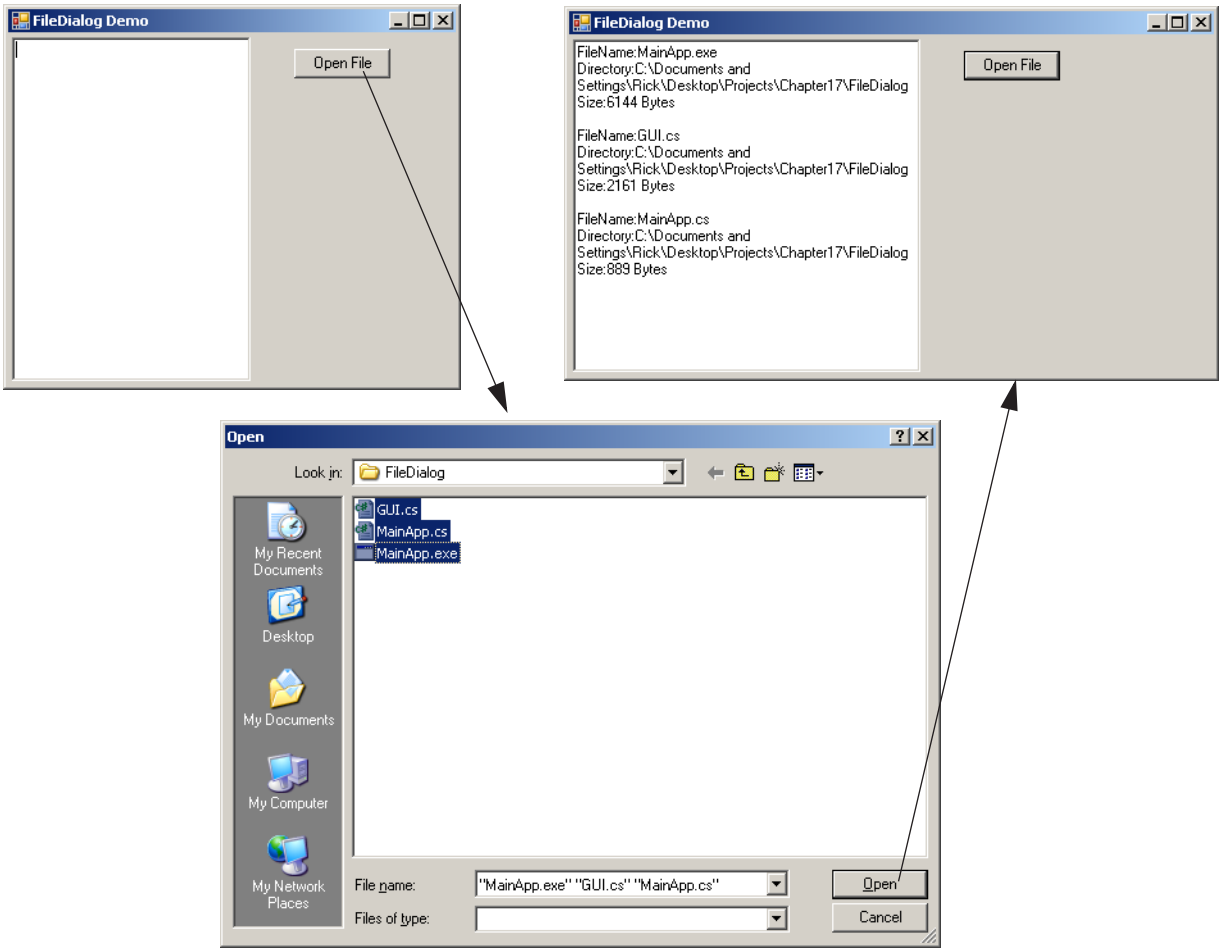


Figure 17-14: Results of Running Example 17.21 and Selecting Three Files

## Quick Review

Use the OpenFileDialog and SaveFileDialog classes to graphically select and open/save files. The OpenFileDialog can be used to select multiple files simultaneously. When used in this manner, the OpenFileDialog.FileNames property returns a string array containing the names of the files selected.

## Summary

In most all cases, data generated by an application and stored on an auxiliary storage device such as a hard disk, is saved as an organized, related collection of bits in a structure commonly referred to as a *file*.

It is the operating system's responsibility to manage the organization, reading, and writing of files. When a new storage device is added to your computer, it must first be formatted in a way that allows the operating system to access its data.

The file, from the operating system's point of view, is the fundamental storage organizational element. An application's associated data can be stored in one or more files. A file is located in another organizational element called a *directory.* A directory is a special type of file that contains a list of files and directories. A directory contained inside another directory is called a *subdirectory.*

The topmost directory structure is referred to as the *root* directory. The root directory of a particular drive is indicated by the name of the drive followed by a colon ':', followed by a backward slash character '\'. The root directory of the C drive would be "C:\".

The location of a particular file within a directory structure is indicated by a string of characters called a *path*. The path to the file's location can be *absolute* or *relative*. An *absolute path* includes the name or letter of the drive and all directory and subdirectory names required to pinpoint the file's location. A *relative path* is the path to a file from some arbitrary starting point, usually a working directory.

You can easily create and manipulate directories and files with the help of several classes provided in the .NET Framework System.IO namespace. These include the *Path*, *File*, *FileInfo*, *Directory*, *DirectoryInfo*, and *DriveInfo* classes.

*Verbatim strings* are formulated by preceding the string with the '*@*' character which signals the compiler to interpret the string literally, including special characters and line breaks.

*Object serialization* provides an easy, convenient way for you to persist application data to disk. Object serialization is also the least flexible way to store application data because you can't edit the resulting file. Use a FileStream object and a BinaryFormatter to serialize objects to disk. Before an object can be serialized it must be tagged as being serializable with the serializable attribute. Place the serializable attribute above the class declaration line.

When *serializing* a collection of objects, remember that all objects contained within the collection must be serializable. You don't have to worry about the collections themselves, including ordinary arrays, as they are already tagged as being serializable.

You can get around the limitation of ordinary serialization by serializing objects to disk in XML format. Use the StreamWriter and XMLSerializer classes to serialize objects to disk in XML format. Use a FileStream and XMLSerializer to deserialize objects from an XML file.

The StreamReader and StreamWriter classes let you read and write text files. Text files are usually processed line-by-line. Lines of text are terminated with the special characters *carriage-return and line-feed* (\r\n). Each line can contain one or more fields *delimited* by some character. The comma ',' is a commonly used field delimiter. Individual fields can be further delimited as required.

Look to the objects in your program to determine the type of information your text file(s) must contain. You'll need to save enough data to recreate objects.

Process a text file by reading each line and breaking it into *tokens* with the String.Split() method. If one or more fields are also delimited, use the String.Split() method to tokenize the data as required.

Use the BinaryReader and BinaryWriter classes to read and write binary data to disk. The BinaryWriter class provides an overloaded Write() method that is used to write each of the simple types including strings and arrays of bytes and characters. The BinaryReader class provides an assortment of Read*Typename*() methods where *Typename* may be any one of the simple types to include strings and arrays of bytes and characters.

You can conduct *random access file I/O* with the BinaryReader, BinaryWriter, and FileStream classes. The FileStream class provides a Seek() method that allows you to position the *file pointer* at any point within a file. As you learned in the previous section, the BinaryReader and BinaryWriter classes provide methods for reading and writing binary, string, byte, and character array data.

Use the OpenFileDialog and SaveFileDialog classes to graphically select and open/save files. The OpenFileDialog can be used to select multiple files simultaneously. When used in this manner the OpenFileDialog.FileNames property returns a string array containing the names of the files selected.

## Skill-Building Exercises

1. **API Drill:** Explore the contents of the System.IO namespace. List each entry and note its purpose.

2. **API Drill:** Explore the contents of the System.Runtime.Serialization and System.Runtime.Serialization.Formatters.Binary namespaces. List each entry and note its purpose.

3. **Programming Exercise:** Compile and run the examples in this chapter. Note their behavior. Experiment by making changes to each program to get different results.

4. **Create Sequence Diagrams:** Step through the code examples in this chapter and follow the paths of execution. Select a part of each program and create a detailed UML sequence diagram that shows objects used, method calls, and return values.

5. **API Drill:** Research the System.ArraySegment<T> generic class and note its purpose.

## Suggested Projects

1. **Employee Database:** Write a GUI application that lets users create a database of employees. Use the Employee code given in Chapter 11. Users should be able to create employees and save their information to a file. Your application should have fields for entering employee information and some way of displaying a list of employees currently in the data base.

2. **Robot Rat:** Write a version of the robot rat program that records each movement the rat makes to disk. Create a feature called Auto-Playback that lets the robot rat read and execute a series of stored movements from a file.

3. **File Lister:** Write a GUI application that recursively traverses a directory and any subdirectories it might contain. Write to file a list of all the files contained within the directories along with any other data about each file users have selected from a set of menu options.

4. **Picture Display:** Write a GUI application that opens image files and displays their contents in a PictureBox.

5. **Asynchronous File I/O:** The FileStream class supports asynchronous file I/O with its BeginRead()/EndRead() and BeginWrite()/ EndWrite() methods. Research these methods and review asynchronous method calling in Chapter 16. Modify the Picture Display program described in suggested project 4 above to asynchronously read large image files.

## Self-Test Questions

1. Before an object can be serialized, with what attribute must it be tagged?

2. (True/False) Before a collection of objects can be serialized, all objects contained within that collection must be serializable.

3. What must be done to a freshly deserialized object before being used in a program?

4. What three classes can be used together to perform random access file I/O?

5. What's the difference between the File class and the FileInfo class?

6. What's the difference between the Directory class and the DirectoryInfo class?

7. What must be done to a new storage device before the computer can use it to read and write data?

8. Describe in your own words the definition of the term *file*.

9. What term is used to describe the topmost directory?

10. Another word that's synonymous with directory is _____.

11. The location of a particular file within a directory structure is indicated by a string of characters called a _____.

12. What's the difference between an absolute path and a relative path?

13. What's the advantage of using a verbatim string to formulate file paths?

14. A character used to separate individual fields in a text file record is called a _____.

## References

Microsoft Developer Network (MSDN) .NET Framework 3.0 Documentation [http://www.msdn.com]

## Notes