Lecture 11a

Files and Streams

OBJECTIVES

In this lecture you will learn:

- To create, read, write and update files.
- The C# streams class hierarchy.
- To use classes `File` and `Directory` to obtain information about files and directories on your computer.
- To use LINQ to search through directories.
- To become familiar with sequential-access file processing.

OBJECTIVES

- To use classes `FileStream`, `StreamReader` and `StreamWriter` to read text from and write text to files.
- To use LINQ and `yield return` to iterate through the records in a file and locate records that match specified criteria.
- To use classes `FileStream` and `BinaryFormatter` to read objects from and write objects to files.

Outline

19.1 Introduction

Files are used for long-term retention of large amounts of data, even after the program that created the data terminates.

Data maintained in files often is called **persistent data**.

Computers store files on **secondary storage devices**, such as magnetic disks, optical disks, flash memory and magnetic tapes.

19.2 Data Hierarchy

The smallest data item that computers support is called a **bit** (short for "**binary digit**"—a digit that can assume one of two values).

Digits, letters and special symbols are referred to as **characters**.

Bytes are composed of eight bits. C# uses the **Unicode® character set** (`www.unicode.org`) in which characters are composed of 2 bytes.

19.2 Data Hierarchy (Cont.)

Just as characters are composed of bits, fields are composed of characters. A **field** is a group of characters that conveys meaning.

Data items processed by computers form a **data hierarchy** (Fig. 19.1), in which data items become larger and more complex in structure.

19.2 Data Hierarchy (Cont.)

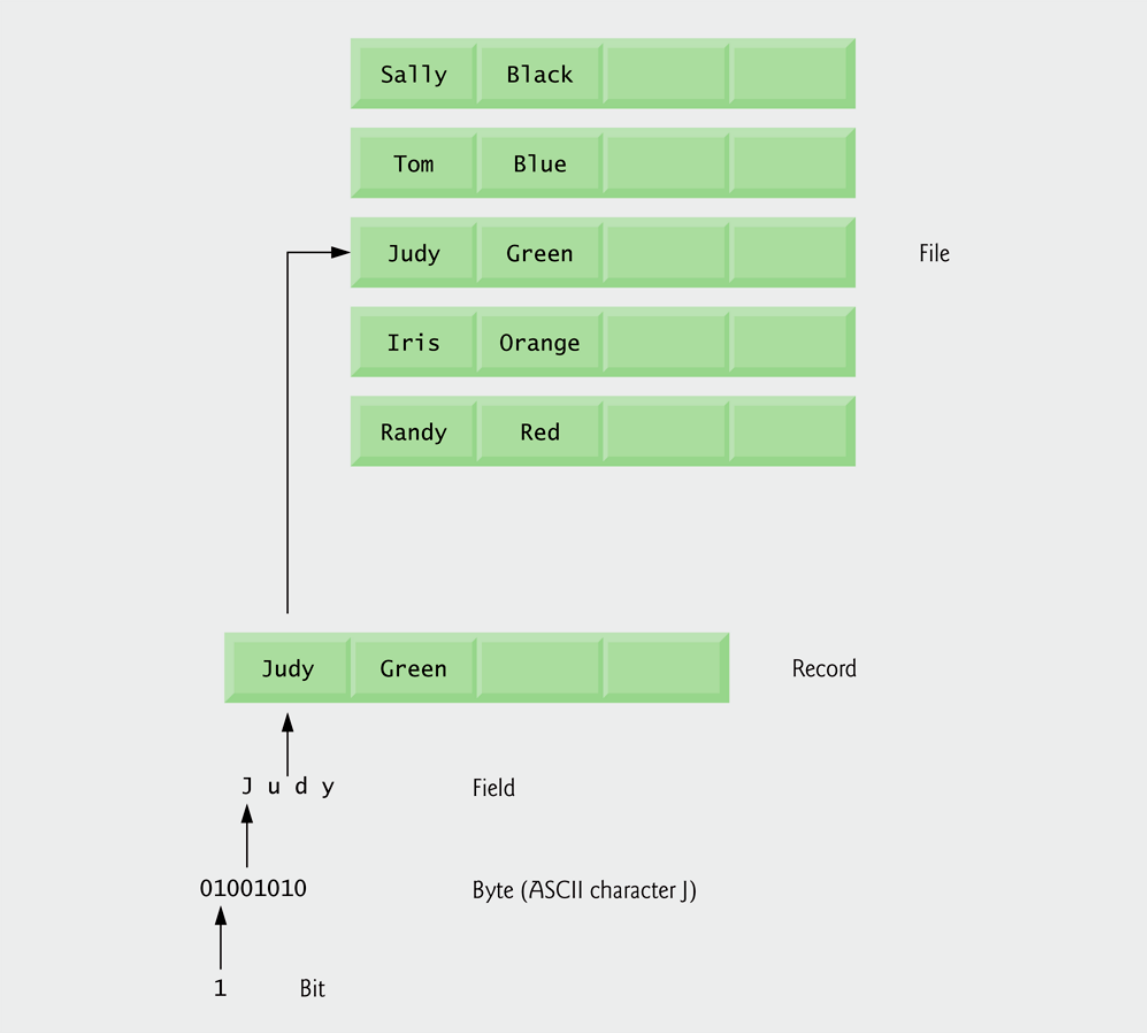


Fig. 19.1 | Data hierarchy.

19.2 Data Hierarchy (Cont.)

Typically, a **record** is composed of several related fields.

A file is a group of related records.

To facilitate the retrieval of specific records from a file, at least one field in each record is chosen as a **record key**, which uniquely identifies a record.

A common file organization is called a **sequential file,** in which records typically are stored in order by a record-key field.

A group of related files often are stored in a **database**.

A collection of programs designed to create and manage databases is called a **database management system (DBMS)**.

19.3 Files and Streams

C# views each file as a sequential **stream** of bytes (Fig. 19.2).

Fig. 19.2 | C#'s view of an *n*-byte file.

Each file ends either with an **end-of-file marker** or at a specific byte number that is recorded in a system-maintained administrative data structure.

When a file is opened, an object is created and a stream is associated with the object.

19.3 Files and Streams (Cont.)

When a console application executes, the runtime environment creates the `Console.Out`, `Console.In` and `Console.Error` streams.

- `Console.In` refers to the **standard input stream object**, which enables a program to input data from the keyboard.
- `Console.Out` refers to the **standard output stream object**, which enables a program to output data to the screen.
- `Console.Error` refers to the **standard error stream object**, which enables a program to output error messages to the screen.

`Console` methods `Write` and `WriteLine` use `Console.Out` to perform output

`Console` methods `Read` and `ReadLine` use `Console.In` to perform input.

19.3 Files and Streams (Cont.)

The `System.IO` namespace includes stream classes such as `StreamReader`, `StreamWriter` and `FileStream` for file input and output.

These stream classes inherit from `abstract` classes `TextReader`, `TextWriter` and `Stream`, respectively.

Abstract class `Stream` provides functionality for representing streams as bytes.

Classes `FileStream`, `MemoryStream` and `BufferedStream` (all from namespace `System.IO`) inherit from class `Stream`.

Class `FileStream` can be used to write data to and read data from files.

19.3 Files and Streams (Cont.)

Class `MemoryStream` enables transfer of data directly to and from memory.

Class `BufferedStream` uses **buffering** to transfer data to or from a stream.

Buffering is an I/O performance-enhancement technique:

- Each output operation is directed to a region in memory, called a **buffer**.
- Actual transfer to the output device is performed in one large **physical output operation** each time the buffer fills
- The output operations directed to the output buffer in memory often are called **logical output operations**.

19.4 Classes `File` and `Directory`

Information is stored in files, which are organized in directories (also called folders).

Classes `File` and `Directory` enable programs to manipulate files and directories on disk.

Class **File** can **determine information about files and can be used to open files for reading or writing**.

Figure 19.3 lists several of class `File`'s `static` methods for manipulating and determining information about files.

19.4 Classes `File` and `Directory` (Cont.)

`static` Method	Description
`AppendText`	Returns a **`StreamWriter`** that appends text to an existing file or creates a file if one does not exist.
`Copy`	Copies a file to a new file.
`Create`	Creates a file and returns its associated **`FileStream`**.
`CreateText`	Creates a text file and returns its associated **`StreamWriter`**.
`Delete`	Deletes the specified file.
`Exists`	Returns `true` if the specified file exists and `false` otherwise.
`GetCreationTime`	Returns a `DateTime` object representing when the file was created.
`GetLastAccessTime`	Returns a `DateTime` object representing when the file was last accessed.

Fig. 19.3 | `File` class `static` methods (partial list). (Part 1 of 2.)

19.4 Classes `File` and `Directory` (Cont.)

`static` Method	Description
`GetLastWriteTime`	Returns a `DateTime` object representing when the file was last modified.
`Move`	Moves the specified file to a specified location.
`Open`	Returns a `FileStream` associated with the specified file and equipped with the specified read/write permissions.
`OpenRead`	Returns a read-only `FileStream` associated with the specified file.
`OpenText`	Returns a `StreamReader` associated with the specified file.
`OpenWrite`	Returns a read/write `FileStream` associated with the specified file.

Fig. 19.3 | `File` class `static` methods (partial list). (Part 2 of 2.)

19.4 Classes `File` and `Directory` (Cont.)

Class `Directory` provides capabilities for manipulating directories. Figure 19.4 lists some of class `Directory`'s `static` methods for directory manipulation.

`static` Method	Description
`CreateDirectory`	Creates a directory and returns its associated `DirectoryInfo` object.
`Delete`	Deletes the specified directory.
`Exists`	Returns `true` if the specified directory exists and `false` otherwise.
`GetDirectories`	Returns a `string` array containing the names of the subdirectories in the specified directory.
`GetFiles`	Returns a `string` array containing the names of the files in the specified directory.

Fig. 19.4 | `Directory` class `static` methods. (Part 1 of 2.)

19.4 Classes `File` and `Directory` (Cont.)

`static` Method	Description
`GetCreationTime`	Returns a `DateTime` object representing when the directory was created.
`GetLastAccessTime`	Returns a `DateTime` object representing when the directory was last accessed.
`GetLastWriteTime`	Returns a `DateTime` object representing when items were last written to the directory.
`Move`	Moves the specified directory to a specified location.

Fig. 19.4 | `Directory` class `static` methods. (Part 1 of 2.)

The `DirectoryInfo` object returned by method `CreateDirectory` contains information about a directory.

Much of the information contained in class `DirectoryInfo` also can be accessed via the methods of class `Directory`.

```
1  using System;
2  using System.IO;
3  using System.Text;
4
5  class Test
6  {
7      public static void Main()
8      {
9          string path = @"c:\temp\MyTest.txt";
10
11         // This text is added only once to the file.
12         if (!File.Exists(path))
13         {
14             // Create a file to write to.
15             string createText = "Hello and Welcome" + Environment.NewLine;
16             File.WriteAllText(path, createText);
17         }
18
19         // This text is always added, making the file longer over time
20         // if it is not deleted.
21         string appendText = "This is extra text" + Environment.NewLine;
22         File.AppendAllText(path, appendText);
23
24         // Open the file to read from.
25         string readText = File.ReadAllText(path);
26         Console.WriteLine(readText);
27     }
28 }
```

```
1  using System;
2  using System.IO;
3  class Test
4  {
5      public static void Main()
6      {
7          string path = @"c:\temp\MyTest.txt";
8
9          // This text is added only once to the file.
10         if (!File.Exists(path))
11         {
12             // Create a file to write to.
13             string[] createText = { "Hello", "And", "Welcome" };
14             File.WriteAllLines(path, createText);
15         }
16
17         // This text is always added, making the file longer over time
18         // if it is not deleted.
19         string appendText = "This is extra text" + Environment.NewLine;
20         File.AppendAllText(path, appendText);
21
22         // Open the file to read from.
23         string[] readText = File.ReadAllLines(path);
24         foreach (string s in readText)
25         {
26             Console.WriteLine(s);
27         }
28     }
29 }
```

Demonstrating Classes `File` and `Directory`

Class **`MainWindow.xaml.cs`** uses `File` and `Directory` methods to access file and directory information.

`FileTestForm.cs`

(1 of 6)

```
1  // Fig  : MainWindow.xaml.cs
2  // Using classes File and Directory.
3  using System.Windows;
4  using System.Windows.Input;
5  using System.IO;
6
7  namespace TestFileProcessing
8  {
9     // displays contents of files and directories
10    public partial class MainWindow : Window
11    {
12       // parameterless constructor
13       public MainWindow ()
14       {
15          InitializeComponent();
16       } // end constructor
17
```

Fig. 19.5 | Testing classes `File` and `Directory`. (Part 1 of 6.)

```
18      // invoked when user presses key
19      private void TxtInput_KeyDown( object sender, KeyEventArgs e )
20      {
21         // determine whether user pressed Enter key
22         if ( e.Key == Key.Enter )
23         {
24            // get user-specified file or directory
25             string fileName = inputTextBox.Text;
26
27            // determine whether fileName is a file
28            if ( File.Exists( fileName ) )
29            {
30               // get file's creation date, modification date, etc.
31               GetInformation( fileName );
32               StreamReader stream = null; // declare StreamReader
33
34               // display file contents through StreamReader
35               try
36               {
37                  // obtain reader and file contents
38                  using ( stream = new StreamReader( fileName ) )
39                  {
40                     outputTextBox.AppendText( stream.ReadToEnd() );
41                  } // end using
42               } // end try
```

FileTestForm.cs

(2 of 6)

Use `File` method `Exists` to determine whether the user-specified text is the name of an existing file.

The `StreamReader` constructor takes as an argument a `string` containing the name of the file to open. May be replaced with `File.OpenText(filename)`

`StreamReader` method `ReadToEnd` read the entire contents of the file as a `string`.

Fig. 19.5 | Testing classes `File` and `Directory`. (Part 2 of 6.)

```
43             catch ( IOException )
44             {
45                MessageBox.Show( "Error reading from file",
46                   "File Error", MessageBoxButton.OK,
47                    MessageBoxImage.Error );
48             } // end catch
49          } // end if
50          // determine whether fileName is a directory
51          else if ( Directory.Exists( fileName ) )
52          {
53             // get directory's creation date,
54             // modification date, etc.
55             GetInformation( fileName );
56
57             // obtain file/directory list of specified directory
58             string[] directoryList =
59                Directory.GetDirectories( fileName );
60
61             outputTextBox.AppendText( "Directory contents:\n" );
62
```

FileTestForm.cs

(3 of 6)

Determine whether the user-specified text is a directory using `Directory` method **Exists**.

Call `Directory` method **GetDirectories** to obtain an array of subdirectories in the specified directory.

Fig. 19.5 | Testing classes `File` and `Directory`. (Part 3 of 6.)

FileTestForm.cs

(4 of 6)

```
63                // output directoryList contents
64                foreach ( var directory in directoryList )
65                   outputTextBox.AppendText( directory + "\n" );
66             } // end else if
67             else
68             {
69                // notify user that neither file nor directory exists
70                MessageBox.Show( inputTextBox.Text +
71                   " does not exist", "File Error",
72                   MessageBoxButton.OK, MessageBoxImage.Error );
73             } // end else
74          } // end if
75       } // end method inputTextBox_KeyDown
76
77       // get information on file or directory,
78       // and output it to outputTextBox
79       private void GetInformation( string fileName )
80       {
81          outputTextBox.Clear();
82
```

Call `File` methods `GetCreationTime`, `GetLastWriteTime` and `GetLastAccessTime` to access file information.

Fig. 19.5 | Testing classes `File` and `Directory`. (Part 4 of 6.)

```
83          // output that file or directory exists
84          outputTextBox.AppendText( fileName + " exists\n" );
85
86          // output when file or directory was created
87          outputTextBox.AppendText( "Created: " +
88             File.GetCreationTime( fileName ) + "\n" );
89
90          // output when file or directory was last modified
91          outputTextBox.AppendText( "Last modified: " +
92             File.GetLastWriteTime( fileName ) + "\n" );
93
94          // output when file or directory was last accessed
95          outputTextBox.AppendText( "Last accessed: " +
96             File.GetLastAccessTime( fileName ) + "\n" );
97       } // end method GetInformation
98    } // end class FileTestForm
99 } // end namespace FileTest
```

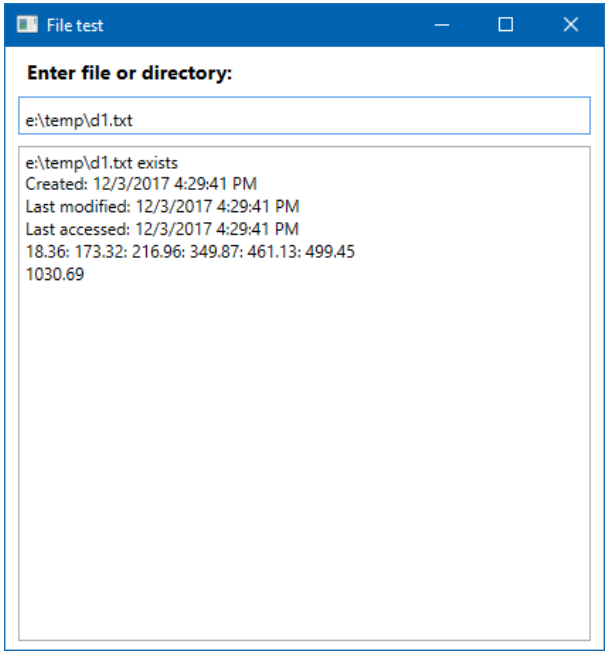
FileTestForm.cs

(5 of 6)

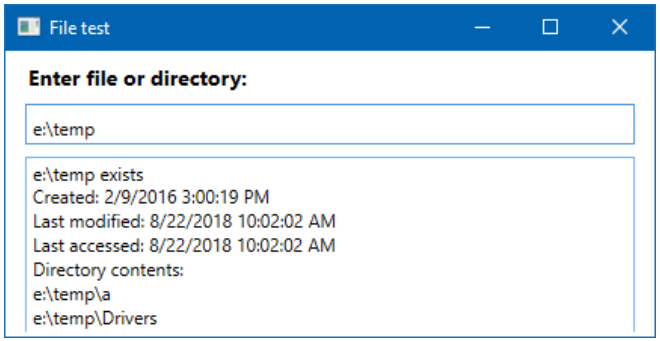
Call `File` methods `GetCreationTime`, `GetLastWriteTime` and `GetLastAccessTime` to access file information.

Fig. 19.5 | Testing classes `File` and `Directory`. (Part 5 of 6.)

(a) Viewing the contents of file “d1.txt"



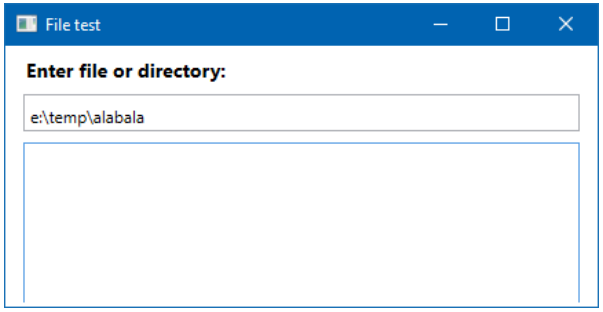
(b) Viewing all files in directory E:\Temp



`MainWindow.xaml.cs`

(6 of 6)

(c) User gives invalid input



(d) Error message is displayed

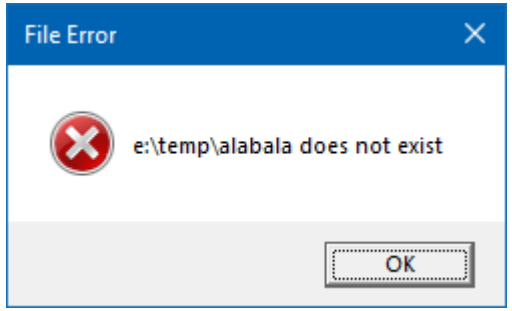


Fig. 19.5 | Testing classes `File` and `Directory`. (Part 6 of 6.)

19.4 Classes `File` and `Directory` (Cont.)

- The `StreamReader` constructor takes as an argument a `string` containing the name of the file to open.
- `StreamReader` method `ReadToEnd` read the entire contents of the file as a `string`.
- Call `Directory` method `GetDirectories` to obtain an array of subdirectories in the specified directory.

19.4 Classes File and Directory (Cont.)

File.ReadAllLines(String filename) **opens** a text file, **reads** all lines of the file into **a string array**, and then **closes** the file.

Data parallelism refers to scenarios in which the same operation is performed concurrently (that is, in parallel) on elements in a source collection or array. In **data parallel operations**, the **source collection is partitioned** so that **multiple threads can operate on different segments concurrently**

Use Data parallelism to process a long text file line by line as follows:

```
string[] lines = File.ReadAllLines(txtProxyListPath.Text);
List<string> listLines = new List<string>(lines);
Parallel.ForEach(listLines, line => { //Your stuff });
```

Searching Directories with LINQ

Outline

Class `MainWindow` (Fig. 19.6) uses LINQ with classes `File`, `Path` and `Directory` to report the number of files of each file type that exist in a directory.

`MainWindow.xaml.cs`

(1 of 8)

```
1  // Fig. 19.6: LINQToFileDirectoryForm.cs
2  // Using LINQ to search directories and determine file types.
3  using System;
4  using System.Collections.Generic;
5  using System.Linq;
6  using System.Windows;
7  using System.IO;
8
9  namespace LINQToFileDirectory
10 {
11    public partial class MainWindow : Window
12    {
13       string currentDirectory; // directory to search
14
15       // store extensions found, and number of each extension found
16       Dictionary<string, int> found = new Dictionary<string, int>();
17
18       // parameterless constructor
19       public MainWindow ()
20       {
21          InitializeComponent();
22       } // end constructor
```

A `Dictionary` is a collection of key/value pairs, in which each key has a corresponding value.

Fig. 19.6 | Using LINQ to search directories and determine file types. (Part 1 of 8.)

```
23
24       // handles the Search Directory Button's Click event
25       private void SearchButton_Click( object sender, RoutedEventArgs e )
26       {
27          // check whether user specified path exists
28          if ( pathTextBox.Text != string.Empty && !Directory.Exists( pathTextBox.Text )
29           ||Directory.GetAccessControl(TxtInputPath.Text).AreAccessRulesProtected )
30          {
31             // show error if user does not specify valid directory
32             MessageBox.Show( "Invalid Directory", "Error",
33                MessageBoxButton.OK, MessageBoxImage.Error );
34          } // end if
35          else
36          {
37             // use current directory if no directory is specified
38             if ( pathTextBox.Text == string.Empty )
39                currentDirectory = Directory.GetCurrentDirectory();
40             else
41                currentDirectory = pathTextBox.Text;
42
43             directoryTextBox.Text = currentDirectory; // show directory
```

MainWindow.xaml.cs

(2 of 8)

Fig. 19.6 | Using LINQ to search directories and determine file types. (Part 2 of 8.)

MainWindow.xaml.cs

(3 of 8)

```
44
45          // clear TextBoxes
46          pathTextBox.Clear();
47          resultsTextBox.Clear();
48
49          SearchDirectory( currentDirectory ); // search the directory
50
51          // allow user to delete .bak files
52          CleanDirectory( currentDirectory );
53
54          // summarize and display the results
55          foreach ( var current in found.Keys )
56          {
57             // display the number of files with current extension
58             resultsTextBox.AppendText( string.Format(
59                "* Found {0} {1} files.\r\n",
60                found[ current ], current ) );
61          } // end foreach
62
63          found.Clear(); // clear results for new search
64       } // end else
65    } // end method searchButton_Click
```

`Dictionary` property `Keys` gets all the keys in the `Dictionary`.

`Dictionary` method `Clear` to delete the contents of the `Dictionary`.

Fig. 19.6 | Using LINQ to search directories and determine file types. (Part 3 of 8.)

```
66
67       // search directory using LINQ
68       private void SearchDirectory( string folder )
69       {
70          // files contained in the directory
71          string[] files = Directory.GetFiles( folder );
72
73          // subdirectories in the directory
74          string[] directories = Directory.GetDirectories( folder );
75
76          // find all file extensions in this directory
77          var extensions =
78             ( from file in files
79               select Path.GetExtension( file ) ).Distinct();
80
81          // count the number of files using each extension
82          foreach ( var extension in extensions )
83          {
84             var temp = extension;
85
```

MainWindow.xaml.cs

(4 of 8)

Call `Directory` method `GetFiles` to get a `string` array containing file names in the specified directory.

Call `Directory` method `GetDirectories` to get a `string` array containing the subdirectory names in the specified directory.

Use LINQ to get the `Distinct` file-name extensions in the `files` array.

Fig. 19.6 | Using LINQ to search directories and determine file types. (Part 4 of 8.)

```
86             // count the number of files with the extension
87             var extensionCount =
88                ( from file in files
89                  where Path.GetExtension( file ) == temp
90                  select file ).Count();
91
92             // if the Dictionary already contains a key for the extension
93             if ( found.ContainsKey( extension ) )
94                found[ extension ] += extensionCount; // update the count
95             else
96                found.Add( extension, extensionCount ); // add new count
97          } // end foreach
98
99          // recursive call to search subdirectories
100         foreach ( var subdirectory in directories )
101            SearchDirectory( subdirectory );
102      } // end method SearchDirectory
103
```

MainWindow.xaml.cs

(5 of 8)

`Dictionary` method `ContainsKey` determines whether the specified key is already in the `Dictionary`.

`Dictionary` method `Add` inserts a new key/value pair into the `Dictionary`.

Fig. 19.6 | Using LINQ to search directories and determine file types. (Part 5 of 8.)

MainWindow.xaml.cs

(6 of 8)

```
104     // allow user to delete backup files (.bak)
105     private void CleanDirectory( string folder )
106     {
107        // files contained in the directory
108        string[] files = Directory.GetFiles( folder );
109
110        // subdirectories in the directory
111        string[] directories = Directory.GetDirectories( folder );
112
113        // select all the backup files in this directory
114        var backupFiles =
115           from file in files
116           where Path.GetExtension( file ) == ".bak"
117           select file;
118
119        // iterate over all backup files (.bak)
120        foreach ( var backup in backupFiles )
121        {
122           MessageBoxResult result = MessageBox.Show( "Found backup file " +
123              Path.GetFileName( backup ) + ". Delete?", "Delete Backup",
124              MessageBoxButton.YesNo, MessageBoxImage.Question );
```

Fig. 19.6 | Using LINQ to search directories and determine file types. (Part 6 of 8.)

MainWindow.xaml.cs

(7 of 8)

```
125
126            // delete file if user clicked 'yes'
127            if ( result == MesssageBoxResult.Yes )
128            {
129               File.Delete( backup ); // delete backup file
130               --found[ ".bak" ]; // decrement count in Dictionary
131
132               // if there are no .bak files, delete key from Dictionary
133               if ( found[ ".bak" ] == 0 )
134                  found.Remove( ".bak" );
135            } // end if
136         } // end foreach
137
138         // recursive call to clean subdirectories
139         foreach ( var subdirectory in directories )
140            CleanDirectory( subdirectory );
141      } // end method CleanDirectory
142   } // end class LINQToFileDirectoryForm
143} // end namespace LINQToFileDirectory
```

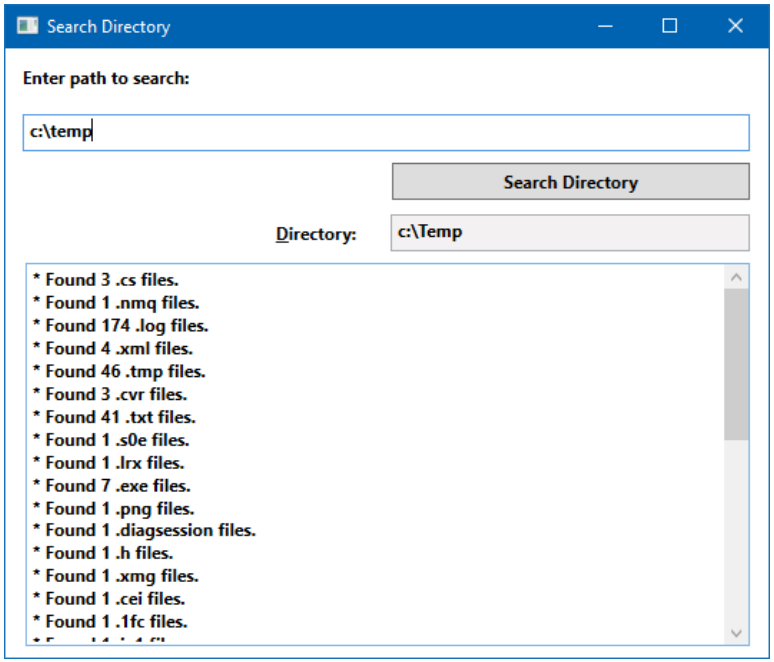
Use `File` method `Delete` to remove the file from disk.

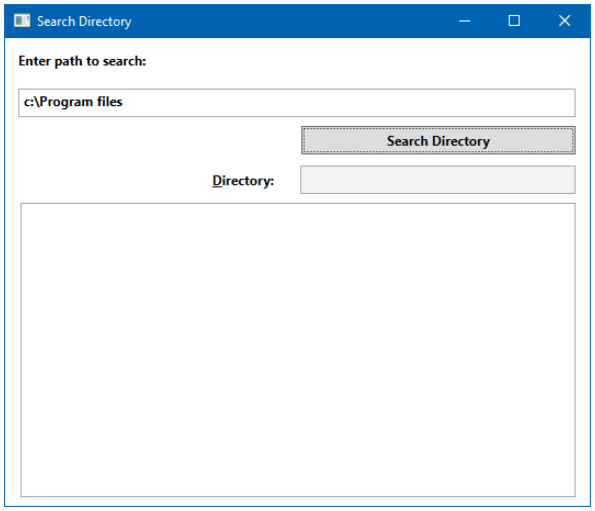
`Dictionary` method `Remove` deletes a key/value pair from the `Dictionary`.

Fig. 19.6 | Using LINQ to search directories and determine file types. (Part 7 of 8.)

MainWindow.xaml.cs

(8 of 8)





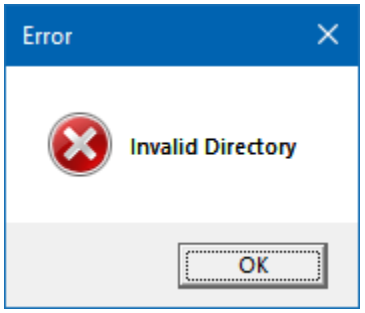


Fig. 19.6 | Using LINQ to search directories and determine file types. (Part 8 of 8.)

19.4 Classes File and Directory (Cont.)

- Path method GetExtension obtains the extension for the specified file name.
- A Dictionary is a collection of key/value pairs, in which each key has a corresponding value.
- Class Dictionary is a generic class.
- Dictionary method ContainsKey determines whether the specified key is already in the Dictionary.
- Dictionary method Add inserts a new key/value pair into the Dictionary. Throws an Exception if the new key is already in the dictionary.

19.4 Classes File and Directory (Cont.)

Use File method Delete to remove the file from disk.

Dictionary method Remove deletes a key/value pair from the Dictionary.

Dictionary property Keys gets all the keys in the Dictionary.

Dictionary property Values gets all the values in the Dictionary.

Dictionary method Clear to delete the contents of the Dictionary.

19.4 Classes `File` and `Directory` (Cont.)

The `Dictionary` indexer can be used to change the value associated with a key.

If a key does not exist, setting the `Dictionary` indexer for that key adds a new key/value pair.(alternative to the `Add()` method)

The `Dictionary` indexer get property throws an exception if the requested key is not in the `Dictionary`

C# imposes no structure on files. Thus, the concept of a "record" does not exist in C# files.

Class `BankUIForm`

We created reusable UserControl `BankUIForm` (Fig. 19.7) to encapsulate a base-class GUI.

`BankUIForm.xaml.cs`

(1 of 5)

```
1  // Fig.  : BankUIForm.xaml.cs
2  // A reusable Windows Form for the following examples  .
3  using System;
4  using System.Windows;
5  using System.Windows.Controls;
6  namespace BankLibrary
7  {
8     public partial class BankUIForm : UserControl
9     {
10       protected int TextBoxCount = 4; // number of TextBoxes on Form
11
12       // enumeration constants specify TextBox indices
13       public enum TextBoxIndices
14       {
15          ACCOUNT,
16          FIRST,
17          LAST,
18          BALANCE
19       } // end enum
```

Fig. 19.7 | Base class for GUIs in our file-processing applications. (Part 1 of 5.)

BankUIForm.xaml.cs

(2 of 5)

```
20
21       // parameterless constructor
22       public BankUIForm()
23       {
24          InitializeComponent();
25       } // end constructor
26
27       // clear all TextBoxes
28       public void ClearTextBoxes()
29       {
30          // iterate through every Control on GrdMain
31          foreach ( UIElement guiControl in GrdMain.Children )
32          {
33             // determine whether Control is TextBox
34             if ( guiControl is TextBox )
35             {
36                // clear TextBox
37                ( ( TextBox ) guiControl ).Clear();
38             } // end if
39          } // end for
40       } // end method ClearTextBoxes
```

Fig. 19.7 | Base class for GUIs in our file-processing applications. (Part 2 of 5.)

BankUIForm.xaml.cs

(3 of 5)

```
41
42       // set text box values to string-array values
43       public void SetTextBoxValues( string[] values )
44       {
45          // determine whether string array has correct length
46          if ( values.Length != TextBoxCount )
47          {
48             // throw exception if not correct length
49             throw ( new ArgumentException( "There must be " +
50                ( TextBoxCount + 1 ) + " strings in the array" ) );
51          } // end if
52          // set array values if array has correct length
53          else
54          {
55             // set array values to textbox values
56             accountTextBox.Text = values[ ( int )
57                TextBoxIndices.ACCOUNT ];
```

Fig. 19.7 | Base class for GUIs in our file-processing applications. (Part 3 of 5.)

BankUIForm.cs

(4 of 5)

```
58             firstNameTextBox.Text = values[ ( int )
59                TextBoxIndices.FIRST ];
60             lastNameTextBox.Text = values[ ( int ) TextBoxIndices.LAST ];
61             balanceTextBox.Text = values[ ( int )
62                TextBoxIndices.BALANCE ];
63          } // end else
64       } // end method SetTextBoxValues
65
66       // return textbox values as string array
67       public string[] GetTextBoxValues()
68       {
69          string[] values = new string[ TextBoxCount ];
70
71          // copy textbox fields to string array
72          values[ ( int ) TextBoxIndices.ACCOUNT ] = accountTextBox.Text;
73          values[ ( int ) TextBoxIndices.FIRST ] = firstNameTextBox.Text;
74          values[ ( int ) TextBoxIndices.LAST ] = lastNameTextBox.Text;
75          values[ ( int ) TextBoxIndices.BALANCE ] = balanceTextBox.Text;
76
77          return values;
78       } // end method GetTextBoxValues
79    } // end class BankUIForm
80 } // end namespace BankLibrary
```

Fig. 19.7 | Base class for GUIs in our file-processing applications. (Part 4 of 5.)

BankUIForm.cs

(5 of 5)

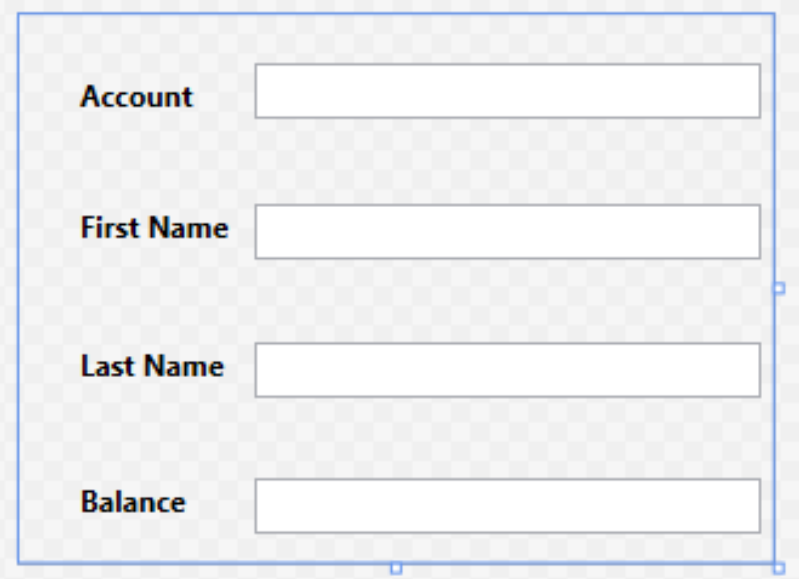


Fig. 19.7 | Base class for GUIs in our file-processing applications. (Part 5 of 5.)

Class Record

Outline

Figure 19.8 contains class `Record` that the next few examples use for maintaining the information in each record that is written to or read from a file.

Record.cs

(1 of 2)

```
1  // Fig. 19.8: Record.cs
2  // Class that represents a data record.
3
4  namespace BankLibrary
5  {
6     public class Record
7     {
8        // auto-implemented Account property
9        public int Account { get; set; }
10
11       // auto-implemented FirstName property
12       public string FirstName { get; set; }
13
14       // auto-implemented LastName property
15       public string LastName { get; set; }
16
17       // auto-implemented Balance property
18       public decimal Balance { get; set; }
```

Fig. 19.8 | Record for sequential-access file-processing applications. (Part 1 of 2.)

Record.cs

(2 of 2)

```
19
20       // parameterless constructor sets members to default values
21       public Record()
22          : this( 0, string.Empty, string.Empty, 0M )
23       {
24       } // end constructor
25
26       // overloaded constructor sets members to parameter values
27       public Record( int accountValue, string firstNameValue,
28          string lastNameValue, decimal balanceValue )
29       {
30          Account = accountValue;
31          FirstName = firstNameValue;
32          LastName = lastNameValue;
33          Balance = balanceValue;
34       } // end constructor
35    } // end class Record
36 } // end namespace BankLibrary
```

Fig. 19.8 | Record for sequential-access file-processing applications. (Part 2 of 2.)

Using a Character Stream to Create an Output File

Class `MainWindow` uses instances of class `Record` to create a sequential-access file.

`CreateFileForm.xaml.cs`

(1 of 11)

```
1  // Fig. : CreateFile
2  // Creating a sequential-access file.
3  using System;
4  using System.Windows;
5  using System.IO;
6  using BankLibrary;
7  using Microsoft.Win32;
8  namespace CreateFile
9  {
10    public partial class MainWindow : Window
11    {
12       private StreamWriter fileWriter; // writes data to text file
13
14       // parameterless constructor
15       public MainWindow()
16       {
17          InitializeComponent();
18       } // end constructor
19
```

Fig. 19.9 | Creating and writing to a sequential-access file. (Part 1 of 11.)

CreateFileForm.xaml.cs

(2 of 11)

```
20      // event handler for Save Button
21      private void BtnSaveAs_Click( object sender, RoutedEventArgs e )
22      {
23         // create and show dialog box enabling user to save file
24         bool? result; // result of SaveFileDialog
25         string fileName; // name of file containing data
26
27         SaveFileDialog fileChooser = new SaveFileDialog() ;
28         // cannot use using, SaveFileDialog is not IDisposable
29            fileChooser.CheckFileExists = false; // let user create file
30            result = fileChooser.ShowDialog();
31            fileName = fileChooser.FileName; // name of file to save data
32
33
```

Class `SaveFileDialog` is used for selecting files.

`SaveFileDialog` method `ShowDialog` displays the dialog and returns a `DialogResult` specifying which button was clicked to close the dialog.

Fig. 19.9 | Creating and writing to a sequential-access file. (Part 2 of 11.)

```
34          // ensure that user clicked "OK"
35          if ( result.HasValue )
36          {
37             // show error if user specified invalid file
38             if ( fileName == string.Empty )
39                MessageBox.Show( "Invalid File Name", "Error",
40                   MessageBoxButton.OK, MessageBoxImage.Error );
41             else
42             {
43                // save file via FileStream if user specified valid file
44                try
45                {  // open file with write access
46                   // overwrites existing records one by one
47                   FileStream output = new FileStream( fileName,
48                      FileMode.OpenOrCreate, FileAccess.Write );
49
50                   // sets file to where data is written
51                   fileWriter = new StreamWriter( output );
52
53                   // disable Save button and enable Enter button
54                   BtnSaveAs.IsEnabled = false;
55                   BtnEnter.IsEnabled = true;
56                } // end try
```

CreateFileForm.xaml.cs

(3 of 11)

The constant `FileMode.OpenOrCreate` indicates that the `FileStream` should open the file if it exists or create the file if it does not.

Fig. 19.9 | Creating and writing to a sequential-access file. (Part 3 of 11.)

```
57              // handle exception if there is a problem opening the file
58              catch ( IOException )
59              {
60                 // notify user if file does not exist
61                 MessageBox.Show( "Error opening file", "Error",
62                    MessageBoxButton.OK, MessageBoxImage.Error );
63              } // end catch
64           } // end else
65        } // end if
66     } // end method saveButton_Click
67
68     // handler for enterButton Click
69     private void BtnEnter_Click( object sender, RoutedEventArgs e )
70     {
71        // store TextBox values string array
72        string[] values = BankUIForm.GetTextBoxValues();
73
74        // Record containing TextBox values to serialize
75        Record record = new Record();
76
```

CreateFileForm.cs

(4 of 11)

An `IOException` is thrown if there is a problem opening the file or creating the `StreamWriter`.

Fig. 19.9 | Creating and writing to a sequential-access file. (Part 4 of 11.)

```
77          // determine whether TextBox account field is empty
78          if ( values[ ( int ) BankUIForm.TextBoxIndices.ACCOUNT ] != string.Empty )
79          {
80             // store TextBox values in Record and serialize Record
81             try
82             {
83                // get account-number value from TextBox
84                int accountNumber = Int32.Parse(
85                   values[ ( int ) BankUIForm.TextBoxIndices.ACCOUNT ] );
86
87                // determine whether accountNumber is valid
88                if ( accountNumber > 0 )
89                {
90                   // store TextBox fields in Record
91                   record.Account = accountNumber;
92                   record.FirstName = values[ ( int )
93                      BankUIForm.TextBoxIndices.FIRST ];
94                   record.LastName = values[ ( int )
95                      BankUIForm.TextBoxIndices.LAST ];
96                   record.Balance = Decimal.Parse(
97                      values[ ( int ) BankUIForm.TextBoxIndices.BALANCE ] );
```

CreateFile (5 of 11)

Fig. 19.9 | Creating and writing to a sequential-access file. (Part 5 of 11.)

```
98
99                // write Record to file, fields separated by commas
100               fileWriter.WriteLine(
101                  record.Account + "," + record.FirstName + "," +
102                  record.LastName + "," + record.Balance );
103            } // end if
104            else
105            {
106               // notify user if invalid account number
107               MessageBox.Show( "Invalid Account Number", "Error",
108                  MessageBoxButton.OK, MessageBoxImage.Error );
109            } // end else
110         } // end try
111         // notify user if error occurs in serialization
112         catch ( IOException )
113         {
114            MessageBox.Show( "Error Writing to File", "Error",
115               MessageBoxButton.OK, MessageBoxImge.Error );
116         } // end catch
```

CreateFile

(6 of 11)

`StreamWriter` method `WriteLine` writes a sequence of characters to a file.

Fig. 19.9 | Creating and writing to a sequential-access file. (Part 6 of 11.)

```
117            // notify user if error occurs regarding parameter format
118            catch ( FormatException )
119            {
120               MessageBox.Show( "Invalid Format", "Error",
121                  MessageBoxButtons.OK, MessageBoxIcon.Error );
122            } // end catch
123         } // end if
124
125         BankUIForm.ClearTextBoxes(); // clear TextBox values
126      } // end method enterButton_Click
127
128      // handler for exitButton Click
129      private void BtnExit_Click( object sender, RoutedEventArgs e )
130      {
131         // determine whether file exists
132         if ( fileWriter != null )
133         {
134            try
135            {
136               // close StreamWriter and underlying file
137               fileWriter.Close();
138            } // end try
```

CreateFile

(7 of 11)

Fig. 19.9 | Creating and writing to a sequential-access file. (Part 7 of 11.)

```
            // notify user of error closing file
            catch ( IOException )
            {
               MessageBox.Show( "Cannot close file", "Error",
                  MessageBoxButton.OK, MessageBoxImage.Error );
            } // end catch
         } // end if

         System.Environment.Exit();
      } // end method exitButton_Click
   } // end class CreateFileForm
} // end namespace CreateFile
```

CreateFile

(8 of 11)

a) BankUI graphical user interface with three additional controls

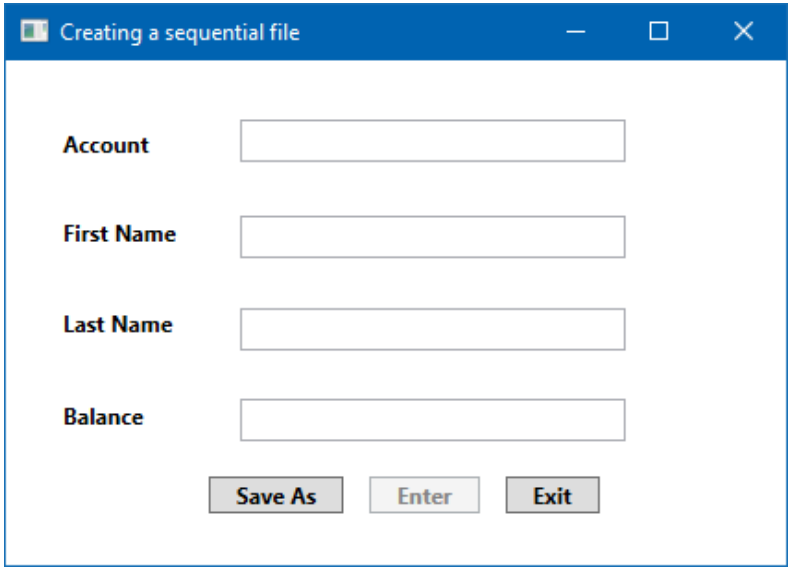


Fig. 19.9 | Creating and writing to a sequential-access file. (Part 8 of 11.)

b) Save File dialog

`CreateFile`

(9 of 11)

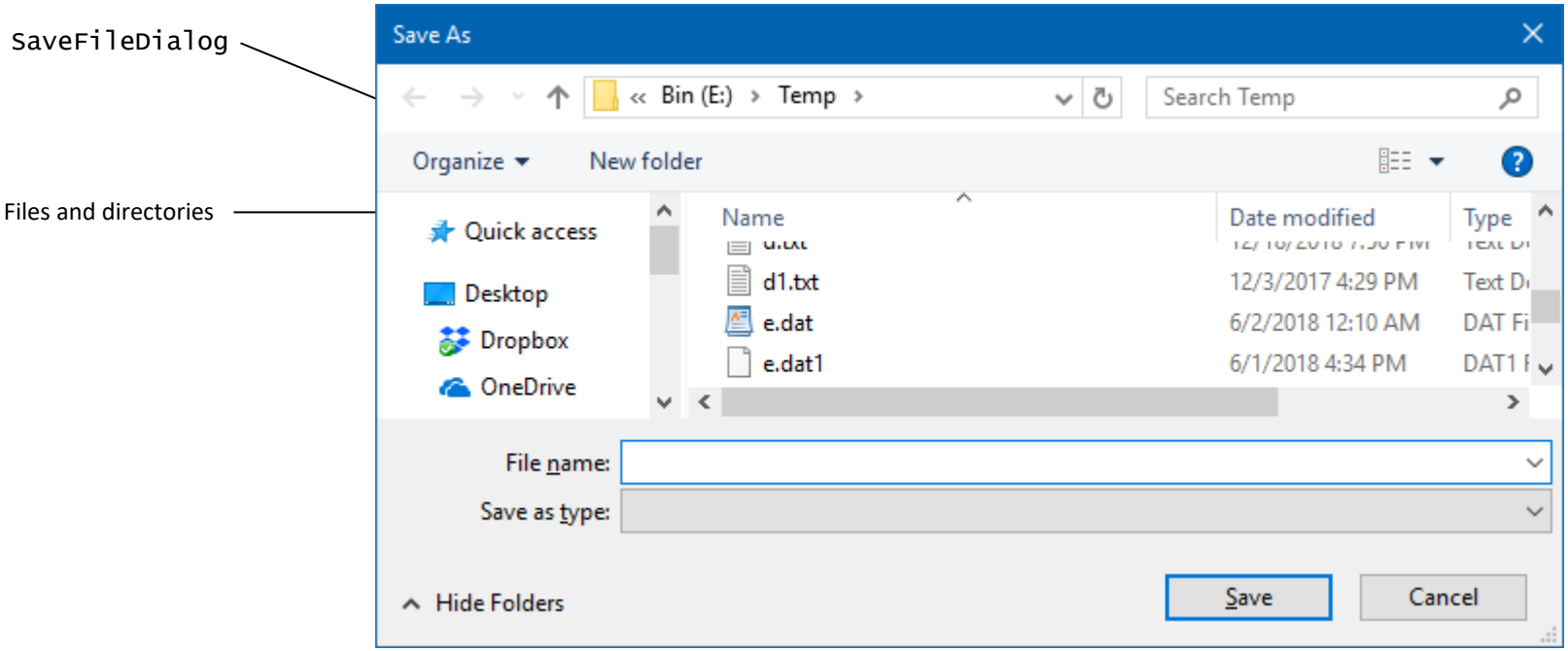
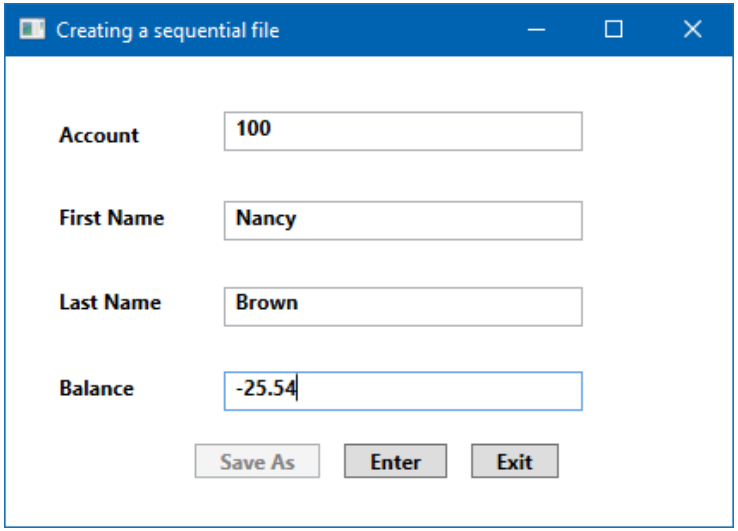


Fig. 19.9 | Creating and writing to a sequential-access file. (Part 9 of 11.)

c) Account 100, "Nancy Brown", saved with a balance of -25.54



d) Account 200, "Stacey Dunn", saved with a balance of 314.33

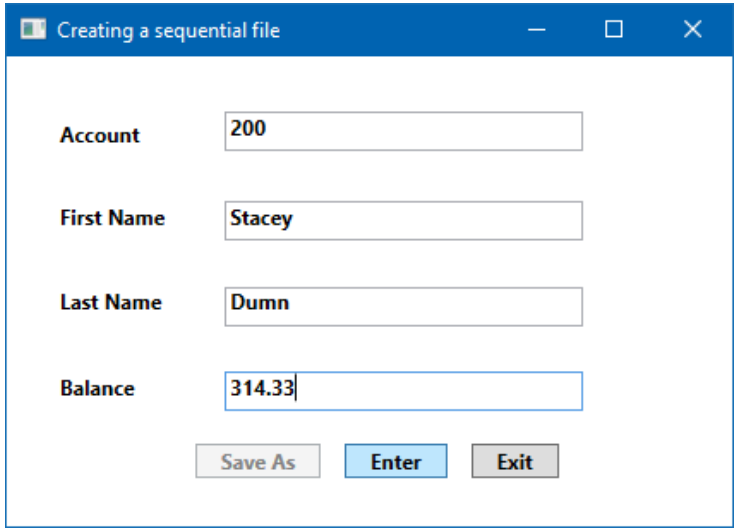
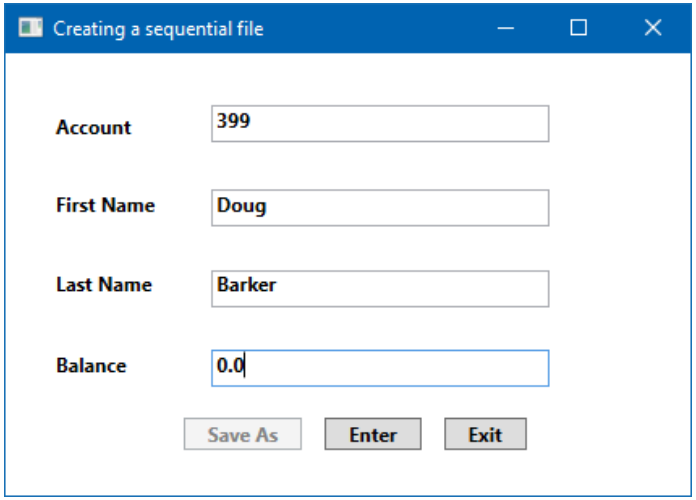
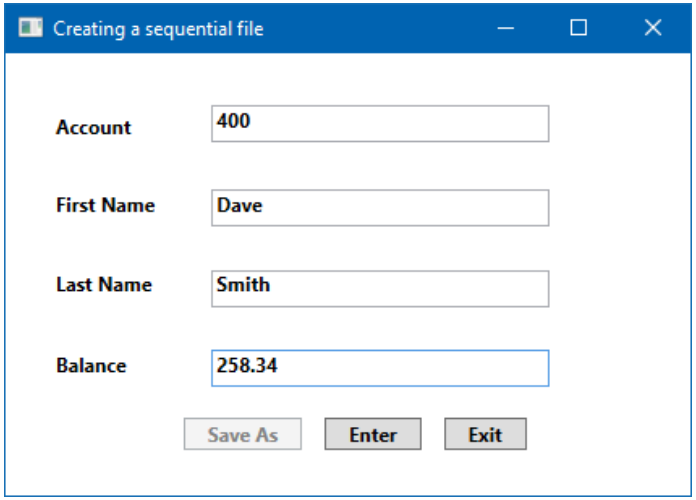


Fig. 19.9 | Creating and writing to a sequential-access file. (Part 10 of 11.)

e) Account 399, "Doug Barker", saved with a balance of 0



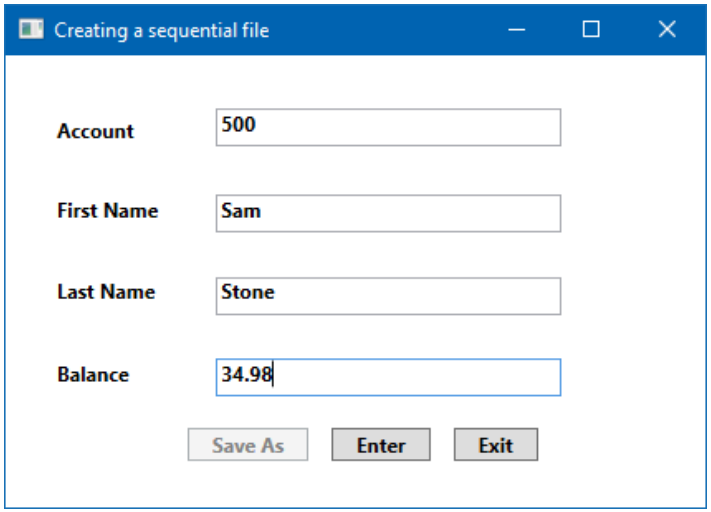
f) Account 400, "Dave Smith", saved with a balance of 258.34



`CreateFile`

(11 of 11)

(g) Account 500, "Sam Stone", saved with a balance of 34.98



(h) Once all accounts are saved, the Exit button closes the application

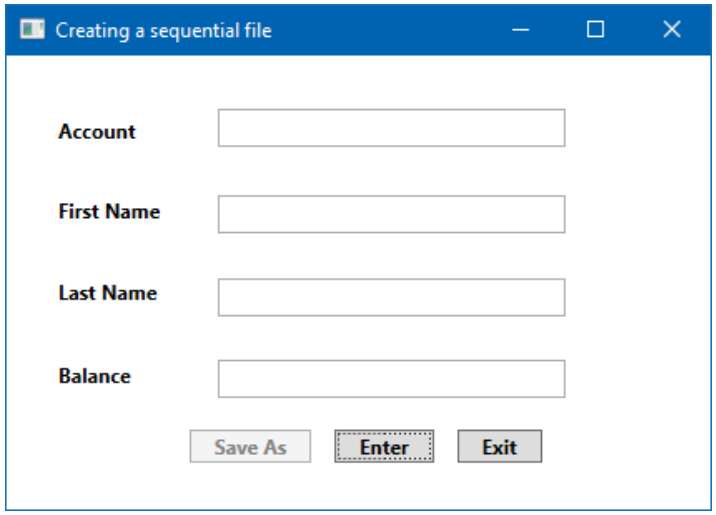


Fig. 19.9 | Creating and writing to a sequential-access file. (Part 11 of 11.)

In this application, the account number is used as the record key—files are created and maintained in account-number order.
This program assumes that the user enters records in account-number order.
Class `CreateFileForm`'s GUI enhances that of class `BankUIForm` with buttons **Save As**, **Enter** and **Exit**..

19.5 Creating a Sequential-Access Text File (Cont.)

Class `SaveFileDialog` is used for selecting files.

`SaveFileDialog` method `ShowDialog` displays the dialog and returns a `DialogResult` specifying which button was clicked to close the dialog.

A `SaveFileDialog` is a `modal dialog`—it prevents the user from interacting with any other window in the program until the user closes it.

Method `ShowDialog` returns a `DialogResult` specifying which button (**Save** or **Cancel**) the user clicked to close the dialog.

You can open files to perform text manipulation by creating objects of class `FileStream`.

19.5 Creating a Sequential-Access Text File (Cont.)

The constant `FileMode.OpenOrCreate` indicates that the `FileStream` should open the file if it exists or create the file if it does not.

To preserve the original contents of a file, use `FileMode.Append`.

The constant `FileAccess.Write` indicates that the program can perform only write operations with the `FileStream` object.

There are two other `FileAccess` constants—`FileAccess.Read` for read-only access and `FileAccess.ReadWrite` for both read and write access.

An `IOException` is thrown if there is a problem opening the file or creating the `StreamWriter`.

19.5 Creating a Sequential-Access Text File (Cont.)

Good Programming Practice 19.1

When opening files, use the `FileAccess` enumeration to control user access to these files.

Good Programming Error 19.1

Failure to open a file before attempting to use it in a program is a logic error.

19.5 Creating a Sequential-Access Text File (Cont.)

`StreamWriter` method `WriteLine` writes a sequence of characters to a file.

The `StreamWriter` object is constructed with a `FileStream` argument that specifies the file to which the `StreamWriter` will output text.

Method `Close` throws an `IOException` if the file or stream cannot be closed properly.

19.5 Creating a Sequential-Access Text File (Cont.)

Performance Tip 19.1

Close each file explicitly when the program no longer needs to reference it. This can reduce resource usage in programs that continue executing long after they finish using a specific file. The practice of explicitly closing files also improves program clarity.

Performance Tip 19.2

Releasing resources explicitly when they are no longer needed makes them immediately available for reuse by other programs, thus improving resource utilization.

19.5 Creating a Sequential-Access Text File (Cont.)

In the sample execution for the program , we entered information for the five accounts

Account number	First name	Last name	Balance
100	Nancy	Brown	-25.54
200	Stacey	Dunn	314.33
300	Doug	Barker	0.00
400	Dave	Smith	258.34
500	Sam	Stone	34.98

Fig. 19.10 | Sample data for the program of Fig. 19.9.

Class `MainWindow` reads records from the file created by the program, then displays the contents of each record.

`ReadFile` (1 of 8)

```
1  // Fig: ReadSequentialAccessFile
2  // Reading a sequential-access file.
3  using System;
4  using System.Windows;
5  using System.IO;
6  using BankLibraryUI;
7  using Microsoft.Win32;
8  namespace ReadFile
9  {
10    public partial class MainWindow : Window
11    {
12       private StreamReader fileReader; // reads data from a text file
13
14       // parameterless constructor
15       public MainWindow()
16       {
17          InitializeComponent();
18       } // end constructor
19
```

Fig. 19.11 | Reading sequential-access files. (Part 1 of 8.)

ReadFile (2 of 8)

```
20       // invoked when user clicks the Open button
21       private void BtnOpen_Click( object sender, RoutedEventArgs e )
22       {
23          // create and show dialog box enabling user to open file
24          bool? result; // result of OpenFileDialog
25          string fileName; // name of file containing data
26
27          OpenFileDialog fileChooser = new OpenFileDialog() ;
28
29             result = fileChooser.ShowDialog();
30             fileName = fileChooser.FileName; // get specified name
31
32
33          // ensure that user clicked "OK"
34          if ( result.HasValue )
35          {
36             BankUIForm.ClearTextBoxes();// method of class BankUIForm
37
```

Create an `OpenFileDialog`, and call its `ShowDialog` method to display the **Open** dialog.

Fig. 19.11 | Reading sequential-access files. (Part 2 of 8.)

ReadFile (3 of 8)

```
38          // show error if user specified invalid file
39          if ( fileName == string.Empty )
40             MessageBox.Show( "Invalid File Name", "Error",
41                MessageBoxButton.OK, MessageBoxImage.Error );
42          else
43          {
44             try
45             {
46                // create FileStream to obtain read access to file
47                FileStream input = new FileStream(
48                   fileName, FileMode.Open, FileAccess.Read );
49
50                // set file from where data is read
51                fileReader = new StreamReader( input );
52
53                BtnOpen.IsEnabled = false; // disable Open File button
54                BtnNext.IsEnabled = true; // enable Next Record button
55             } // end try
```

Create a `FileStream` object, passing constant `FileMode.Open` as the second argument to the `FileStream` constructor.

Fig. 19.11 | Reading sequential-access files. (Part 3 of 8.)

ReadFile (4 of 8)

```
56                catch ( IOException )
57                {
58                   MessageBox.Show( "Error reading from file",
59                      "File Error", MessageBoxButton.OK,
60                      MessageBoxImage.Error );
61                } // end catch
62             } // end else
63          } // end if
64       } // end method openButton_Click
65
66       // invoked when user clicks Next button
67       private void BtnNext_Click( object sender, RoutedEventArgs e )
68       {
69          try
70          {
71             // get next record available in file
72             string inputRecord = fileReader.ReadLine();
73             string[] inputFields; // will store individual pieces of data
74
75             if ( inputRecord != null )
76             {  // read next record
77                inputFields = inputRecord.Split( ',' );
78
```

`StreamReader` method `ReadLine` reads the next line from the file.

Fig. 19.11 | Reading sequential-access files. (Part 4 of 8.)

```
79             Record record = new Record(
80                Convert.ToInt32( inputFields[ 0 ] ), inputFields[ 1 ],
81                inputFields[ 2 ], Convert.ToDecimal(
82                inputFields[ 3 ] ) );
83
84             // copy string-array values to TextBox values
85             BankUIForm.SetTextBoxValues( inputFields );
86          } // end if
87          else
88          {  // end of file reached
89             // close StreamReader and underlying file
90             fileReader?.Close();
91             BtnOpen.IsEnabled = true; // enable Open File button
92             BtnNexxt.IsEnabled = false; // disable Next Record button
93             ClearTextBoxes();
94
95             // notify user if no Records in file
96             MessageBox.Show( "No more records in file", string.Empty,
97                MessageBoxButton.OK, MessageBoxImage.Information );
98          } // end else
99       } // end try
```

ReadFile(5 of 8)

Construct a `Record` object using the data from the file.

Display the `Record` values in the `TextBox`es.

Fig. 19.11 | Reading sequential-access files. (Part 5 of 8.)

```
100         catch ( IOException )
101         {
102            MessageBox.Show( "Error Reading from File", "Error",
103               MessageBoxButtons.OK, MessageBoxIcon.Error );
104         } // end catch
105      } // end method BtnNext_Click
106   } // end class
107} // end namespace
```

ReadFile (6 of 8)

a) BankUI graphical user interface with an Open File button

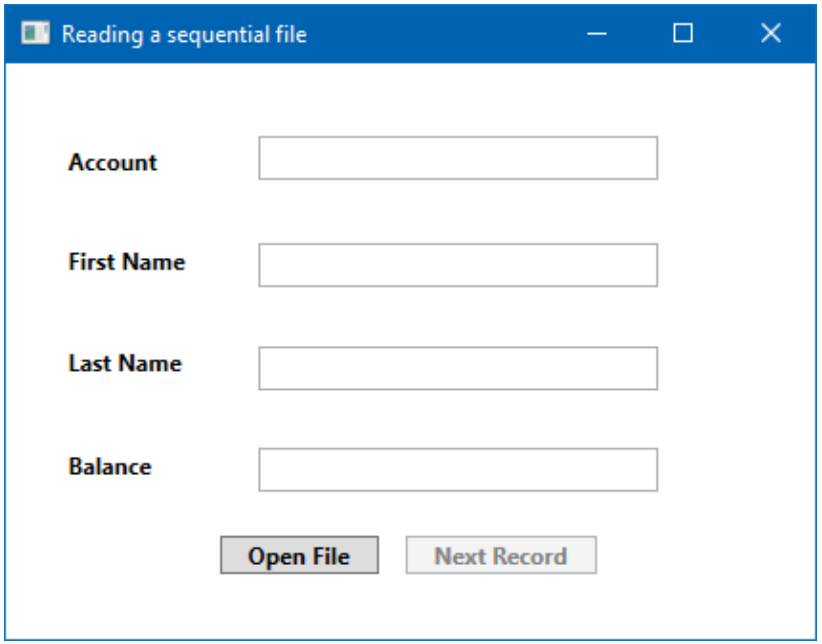
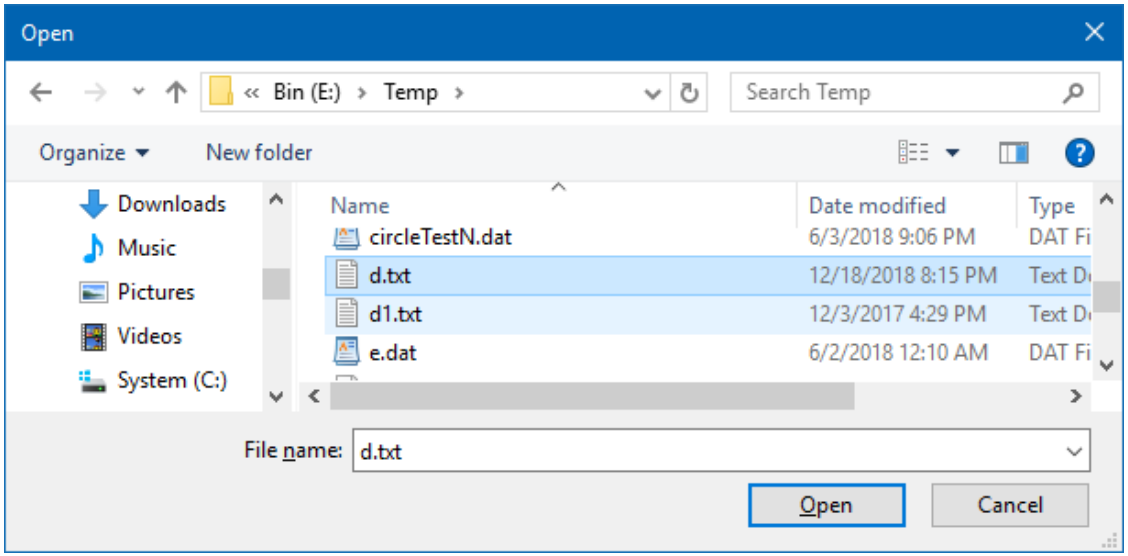


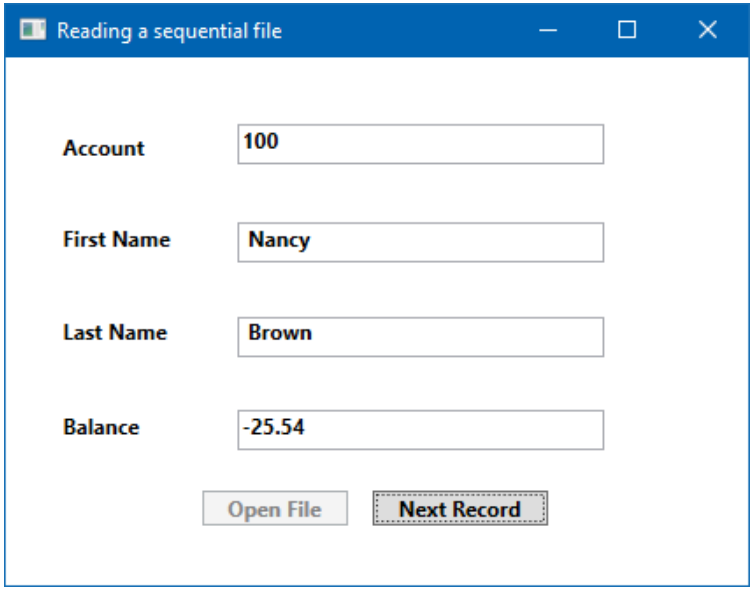
Fig. 19.11 | Reading sequential-access files. (Part 6 of 8.)

ReadFile (7 of 8)

b) OpenFileDialog window



c) Reading account 100



d) Reading account 500

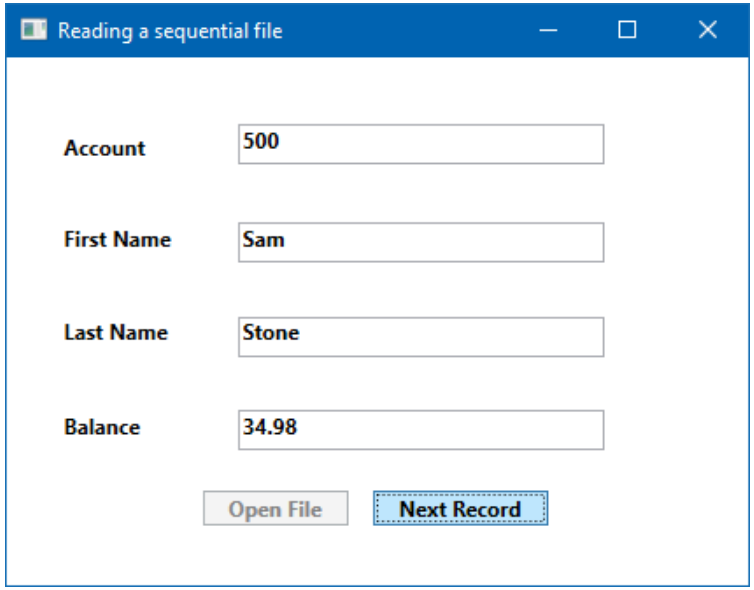
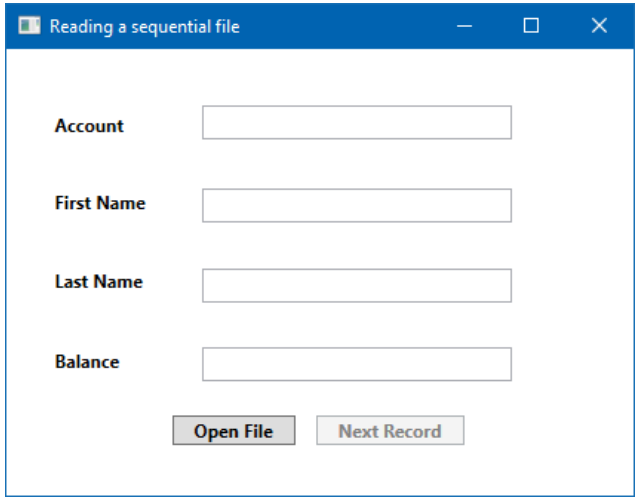


Fig. 19.11 | Reading sequential-access files. (Part 7 of 8.)

`ReadFile` (8 of 8)

e) Reading account 500



f) User is shown a messagebox when all records have been read

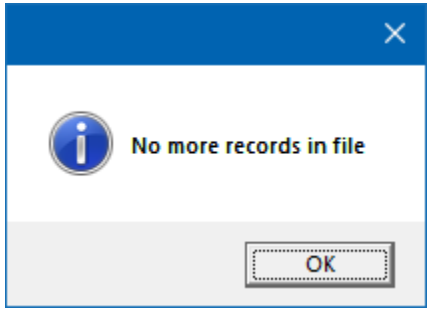


Fig. 19.11 | Reading sequential-access files. (Part 8 of 8.)

19.6 Reading Data from a Sequential-Access Text File (Cont.)

The behavior and GUI for the **`Save`** and **`Open`** dialog types are identical, except that **`Save`** is replaced by **`Open`**.

Specify read-only access to a file by passing constant `FileAccess.Read` as the third argument to the `FileStream` constructor.

Error-Prevention Tip 19.1

Open a file with the `FileAccess.Read` file-open mode if its contents should not be modified. This prevents unintentional modification of the contents.

- `StreamReader` method `ReadLine` reads the next line from the file.

Outline

A `FileStream` object can reposition its **file-position pointer** to any position in the file.

When a `FileStream` object is opened, its file-position pointer is set to byte position `0`.

Class `MainWindow` is a credit-inquiry program that enables a credit manager to search for and display account information for customers.

`CreditInquiry`
(1 of 10)

```
1  // Fig: CreditInquiry Read a file sequentially and display contents based on
2  // account type specified by user ( credit, debit or zero balances ).
3  using BankLibraryUI;
4  using System;
5  using System.Windows;
6  using System.IO;
7  using System.Linq;
8  using System.Collections.Generic;
9  using Microsoft.Win32;
10 namespace CreditInquiry
11 {
12    public partial class MainWindow : Window
13    {
14       private FileStream input; // maintains the connection to the file
15       private StreamReader fileReader; // reads data from text file
16
```

Fig. 19.12 | Credit-inquiry program. (Part 1 of 10.)

Outline

CreditInquiry

(2 of 10)

```
17       // name of file that stores credit, debit and zero balances
18       private string fileName;
19
20       // parameterless constructor
21       public MainWindow()
22       {
23          InitializeComponent();
24       } // end constructor
25
26       // invoked when user clicks Open File button
27       private void BtnOpenFile_Click( object sender, RoutedEventArgs e )
28       {
29          // create dialog box enabling user to open file
30          bool? result;
31
32          OpenFileDialog fileChooser = new OpenFileDialog() ;
33
34             result = fileChooser.ShowDialog();
35             fileName = fileChooser.FileName;
36
37
```

Fig. 19.12 | Credit-inquiry program. (Part 2 of 10.)

CreditInquiry
(3 of 10)

```
38          // exit event handler if user clicked Cancel
39          if ( result.HasValue )
40          {
41             // show error if user specified invalid file
42             if ( fileName == string.Empty )
43                MessageBox.Show( "Invalid File Name", "Error",
44                   MessageBoxButton.OK, MessageBoxImage.Error );
45             else
46             {
47                // create FileStream to obtain read access to file
48                input = new FileStream( fileName,
49                   FileMode.Open, FileAccess.Read );
50
51                // set file from where data is read
52                fileReader = new StreamReader( input );
53
54                // enable all GUI buttons, except for Open File button
55                BtnOpenFile.IsEnabled = false;
56                BtnCrediteBalances.IsEnabled = true;
57                BtnDebitBalances.IsEnabled = true;
58                BtnZeroBalances.IsEnabled = true;
59             } // end else
60          } // end if
61       } // end method BtnOpenFile_Click
```

Fig. 19.12 | Credit-inquiry program. (Part 3 of 10.)

Outline

```
62
63       // invoked when user clicks credit balances,
64       // debit balances or zero balances button
65       private void GetBalances_Click( object sender, RoutedEventArgs e )
66       {
67          // delegate used to check a balance against a certain condition
68          Func< decimal, bool > balanceChooser;
69
70          // convert sender explicitly to object of type button
71          Button senderButton = ( Button ) sender;
72
73          // determine the condition the account balances must satisfy
74          switch ( senderButton.Content )
75          {
76             case "Credit Balances":  // positive balances
77                balanceChooser = balance => balance > 0M;
78                break;
79             case "Debit Balances": // negative balances
80                balanceChooser = balance => balance < 0M;
81                break;
82             default: // zero balances
83                balanceChooser = balance => balance == 0;
84                break;
85          } // end switch
```

CreditInquiry

(4 of 10)

Use class `Func` to declare variable `balanceChooser` as a delegate to a function that receives a `decimal` and returns a `bool`.

The `sender` parameter represents the control that generated the event.

Obtains the `Button` object's text to determine which type of accounts to display.

Create lambda expressions that determine the appropriate balances to select.

Fig. 19.12 | Credit-inquiry program. (Part 4 of 10.)

Outline

CreditInquiry

(5 of 10)

```
86
87          // read and display file information
88          try
89          {
90             TxtOutput.Text = "The accounts are:\n";
91
92             // select records that match account type
93             var balanceQuery =
94                from line in fileReader.Lines()
95                let record = line.Split( ',' ) as string[]
96                where balanceChooser( Convert.ToDecimal( record[ 3 ] ) )
97                select new Record
98                {
99                   Account = Convert.ToInt32( record[ 0 ] ),
100                  FirstName = record[ 1 ],
101                  LastName = record[ 2 ],
102                  Balance = Convert.ToDecimal( record[ 3 ] )
103               };
104
```

Get the lines from the file, `split` each record, use the delegate to determine whether a record should be selected, and create a `Record` object for each selected record.

Fig. 19.12 | Credit-inquiry program. (Part 5 of 10.)

Outline

CreditInquiry

(6 of 10)

```
105            // display each selected Record
106            foreach ( var creditRecord in balanceQuery )
107            {
108               // display the Record's information in the RichTextBox
109               TxtOutput.AppendText(
110                  string.Format( "{0}\t{1}\t{2}\n", creditRecord.Account,
111                  creditRecord.FirstName, creditRecord.LastName ) );
112            } // end foreach
113         } // end try
114         // handle exception when file cannot be read
115         catch ( IOException )
116         {
117            MessageBox.Show( "Cannot Read File", "Error",
118               MessageBoxButton.OK, MessageBoxImage.Error );
119         } // end catch
120      } // end method GetBalances_Click
121
```

Fig. 19.12 | Credit-inquiry program. (Part 6 of 10.)

```
122     // invoked when user clicks Done button
123     private void BtnDone_Click( object sender, RoutedEventArgs e )
124     {
125
126
127           // close file and StreamReader
128           try
129           {
130              // close StreamReader and underlying file
131              fileReader?.Close();
132           } // end try
133           // handle exception if FileStream does not exist
134           catch ( IOException )
135           {
136              // notify user of error closing file
137              MessageBox.Show( "Cannot close file", "Error",
138                 MessageBoxButton.OK, MessageBoxImage.Error );
139           } // end catch
140
141
142        System.Environment.Exit(0);
143     } // end method BtnDone_Click
144  } // end class CreditInquiryForm
```

CreditInquiry

(7 of 10)

Fig. 19.12 | Credit-inquiry program. (Part 7 of 10.)

Outline

```
145
146   // static class containing extension methods for class StreamReader
147   public static class StreamReaderExtensions
148   {
149      // iterate over each line in a file
150      public static IEnumerable<string> Lines( this StreamReader source )
151      {
152         // check for null reference
153         if ( source == null )
154            throw new ArgumentNullException( "StreamReader is null" );
155
156         // start at the beginning of the file
157         source.BaseStream.Seek( 0, SeekOrigin.Begin );
158
159         string line; // a line of text
160
161         // while there are lines left in the file
162         while ( ( line = source.ReadLine() ) != null )
163         {
164            yield return line; // return one line of the file as a string
165         } // end while
166      } // end extension method Lines
167   } // end static class StreamReaderExtensions
168} // end namespace CreditInquiry
```

CreditInquiry

(8 of 10)

Extension method `Lines` acts as an iterator for the lines of text being read from a `StreamReader`.

Use `StreamReader` property `BaseStream` to invoke the `Seek` method of the underlying `FileStream` to reset the file-position pointer back to the beginning of the file.

Read one line at a time from the file until the end of file is reached.

The **yield return** statement returns one line of text, then waits for the next item to be requested from the client code using method `Lines`.

Fig. 19.12 | Credit-inquiry program. (Part 8 of 10.)

19.7 Case Study: Credit Inquiry Program Using LINQ (Cont.)

When you use the **`yield`** keyword in a statement, you indicate that the **method**, **operator**, or **get** accessor in which it appears is an iterator.

You use a **`yield return`** statement to return **each element one at a time**.

You consume an iterator method by using a **`foreach`** statement or **`LINQ`** query. Each iteration of the **`foreach`** loop calls the iterator method. When a **`yield return`** statement is reached in the iterator method, expression is returned, and the current location in code is retained. Execution is restarted from that location the next time that the iterator function is called.

You can use a **`yield break`** statement to end the iteration.

Using yield to execute SQL command

```
public IEnumerable<T> Read<T>(string sql, Func<IDataReader, T> make, params object[] parms)
{
    using (var connection = CreateConnection())
    {
        using (var command = CreateCommand(CommandType.Text, sql, connection, parms))
        {
            command.CommandTimeout = dataBaseSettings.ReadCommandTimeout;
            using (var reader = command.ExecuteReader())
            {
                while (reader.Read())
                {   // read and process one record at a time
                    yield return make(reader);
                }
            }
        }
    }
}
```

19.7 Case Study: Credit Inquiry Program Using LINQ (Cont.)

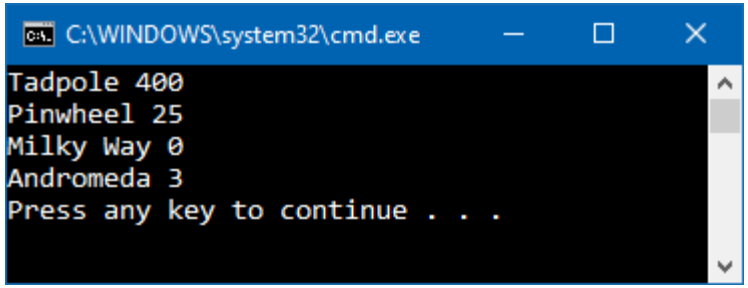
```
public class PowersOf2
{
    static void Main()
    {
        // Display powers of 2 up to the exponent of 8:
        foreach (int i in Power(2, 8))
        {
            Console.Write("{0} ", i);
        }
    }

    public static System.Collections.Generic.IEnumerable<int> Power(int number, int exponent)
    {
        int result = 1;

        for (int i = 0; i < exponent; i++)
        {
            result = result * number;
            yield return result;
        }
    }

    // Output: 2 4 8 16 32 64 128 256
}
```

```
public static class GalaxyClass
{
    static void Main(string[] args)
    {
        ShowGalaxies();
    }
    public static void ShowGalaxies()
    {
        var theGalaxies = new Galaxies();
        foreach (Galaxy theGalaxy in theGalaxies.NextGalaxy)
        {
            Console.WriteLine(theGalaxy.Name + " " + theGalaxy.MegaLightYears.ToString());
        }
    }
    public class Galaxies
    {
        public System.Collections.Generic.IEnumerable<Galaxy> NextGalaxy
        {
            get
            {
                yield return new Galaxy { Name = "Tadpole", MegaLightYears = 400 };
                yield return new Galaxy { Name = "Pinwheel", MegaLightYears = 25 };
                yield return new Galaxy { Name = "Milky Way", MegaLightYears = 0 };
                yield return new Galaxy { Name = "Andromeda", MegaLightYears = 3 };
            }
        }
    }
    public class Galaxy
    {
        public String Name { get; set; }
        public int MegaLightYears { get; set; }
    }
}
```

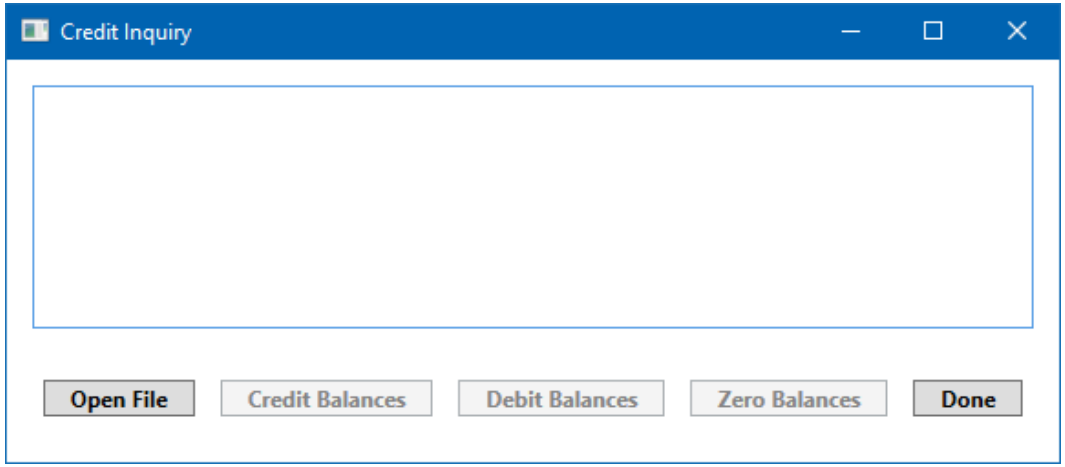


Outline

`CreditInquiry`

(9 of 10)

a)



b)

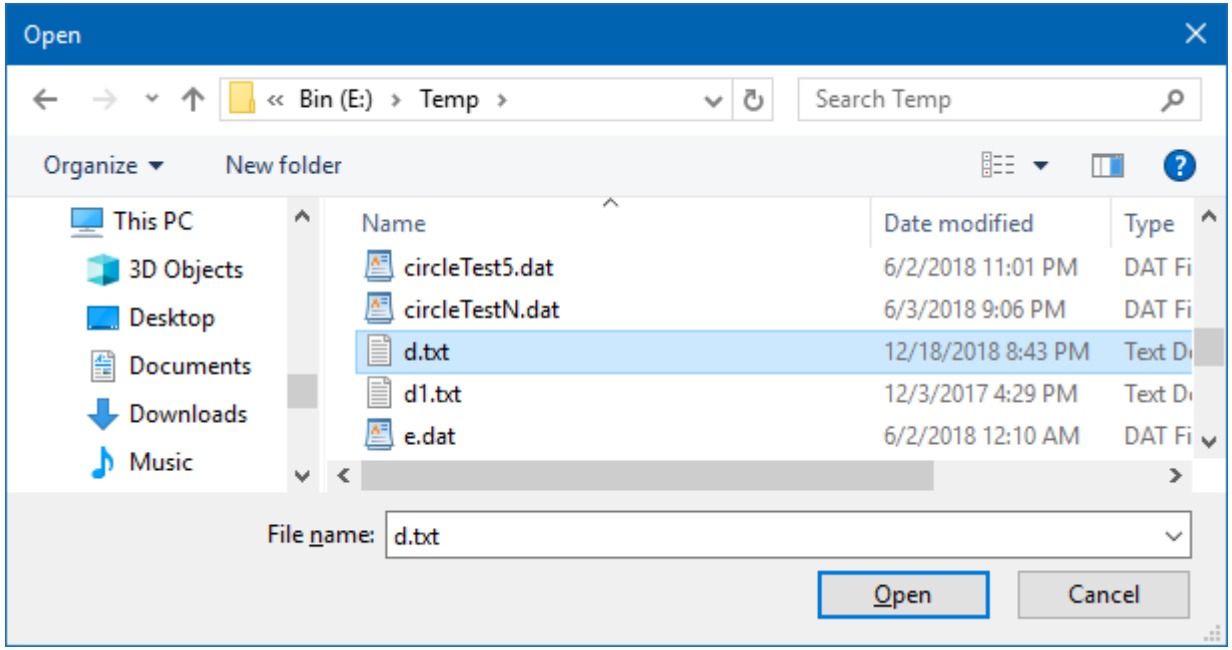


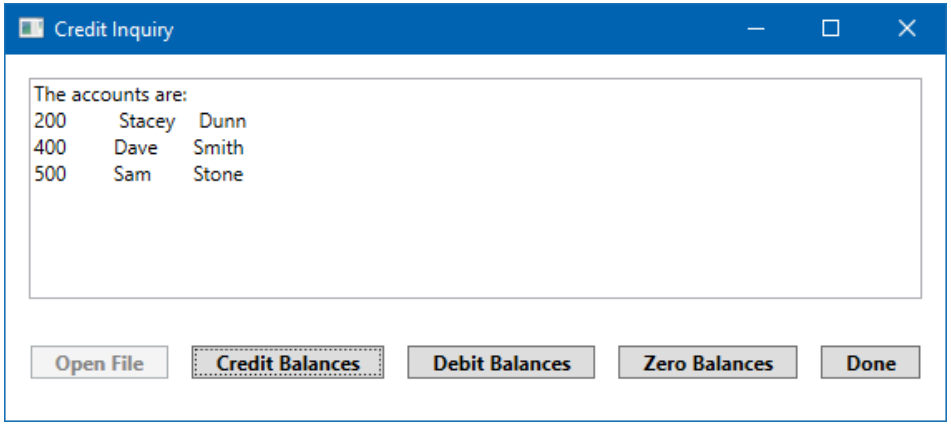
Fig. 19.12 | Credit-inquiry program. (Part 9 of 10.)

Outline

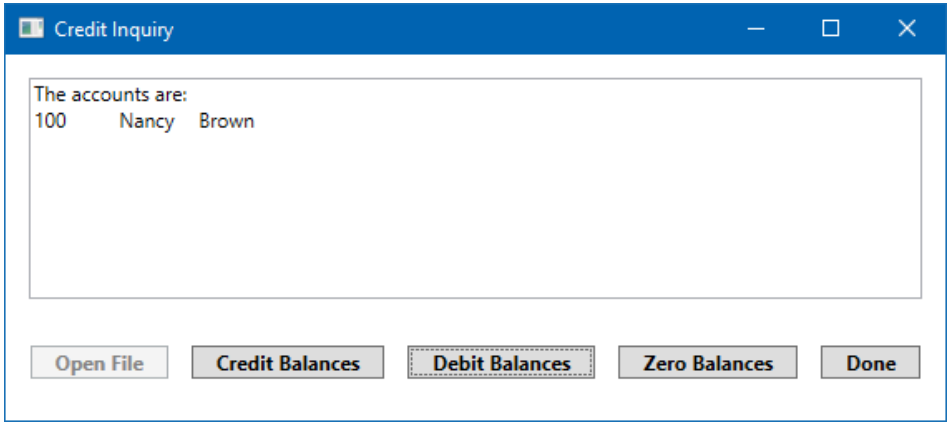
CreditInquiry

(10 of 10)

c)



d)



e)

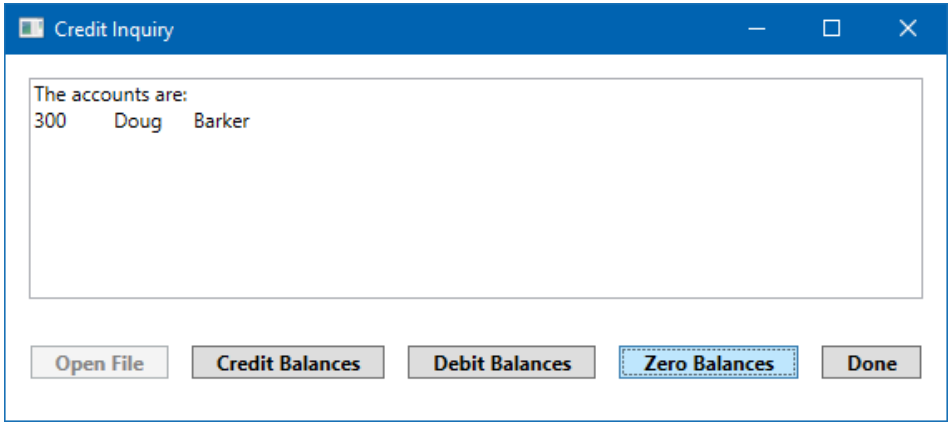


Fig. 19.12 | Credit-inquiry program. (Part 10 of 10.)

19.7 Case Study: Credit Inquiry Program Using LINQ (Cont.)

`RichTextBox`es provide more functionality than regular `TextBox`es.

`RichTextBox`es display multiple lines of text by default.

The `sender` parameter represents the control that generated the event.

`yield return` signals to the compiler that the method in which it appears is an `iterator` block

http://msdn.microsoft.com/en-us/library/9k7k7cf0.aspx

19.7 Case Study: Credit Inquiry Program Using LINQ (Cont.)

As a good example for **`local methods`** are methods implemented as iterators. **They commonly need a non-iterator wrapper method for eagerly checking the arguments at the time of the call.** (The iterator itself doesn't start running until **`MoveNext`** is called).

Local methods are perfect for this **scenario**:

19.7 Case Study: Credit Inquiry Program Using LINQ (Cont.)

```
public IEnumerable<T> Filter<T>(IEnumerable<T> source, Func<T, bool> filter)
{
    if (source == null) throw new ArgumentNullException(nameof(source));
    if (filter == null) throw new ArgumentNullException(nameof(filter));

    return Iterator();

    IEnumerable<T> Iterator()
    {
        foreach (var element in source)
        {
            if (filter(element)) { yield return element; }
        }
    }
}
```

19.7 Case Study: Credit Inquiry Program Using LINQ (Cont.)

If **`Iterator`** had been a private method next to **`Filter`**, it would have been available for other members to accidentally use directly (without argument checking). Also, it would have needed to take all the same arguments as **`Filter`** instead of having them just be in scope.

19.8 Serialization

Sometimes it is easier to read or write entire objects than to read and write individual fields.

C# provides such a mechanism, called **object serialization**.

A **serialized object** is an object represented as a sequence of bytes that includes the object's data, its type and the types of data stored in the object.

After a serialized object has been written to a file, it can be read from the file and **deserialized**.

19.8 Serialization (Cont.)

Class `BinaryFormatter` enables entire objects to be written to or read from a stream.

`BinaryFormatter` method `Serialize` writes an object's representation to a file.

`BinaryFormatter` method `Deserialize` reads this representation from a file and reconstructs the original object.

Both methods throw a `SerializationException` if an error occurs during serialization or deserialization.

Defining the `RecordSerializable` Class

Class `RecordSerializable` is marked with the `[Serializable]` attribute, which indicates that `RecordSerializable` objects can be serialized.

Record Serializable.cs

(1 of 2)

```
1  // Fig: RecordSerializable.cs
2  // Serializable class that represents a data record.
3  using System;
4
5  namespace BankLibrary
6  {
7     [Serializable]
8     public class RecordSerializable
9     {
10       // automatic Account property
11       public int Account { get; set; }
12
```

Fig. 19.13 | `RecordSerializable` class for serializable objects. (Part 1 of 2.)

```
13       // automatic FirstName property
14       public string FirstName { get; set; }
15
16       // automatic LastName property
17       public string LastName { get; set; }
18
19       // automatic Balance property
20       public decimal Balance { get; set; }
21
22       // default constructor sets members to default values
23       public RecordSerializable()
24          : this( 0, string.Empty, string.Empty, 0M )
25       {
26       } // end constructor
27
28       // overloaded constructor sets members to parameter values
29       public RecordSerializable( int accountValue, string firstNameValue,
30          string lastNameValue, decimal balanceValue )
31       {
32          Account = accountValue;
33          FirstName = firstNameValue;
34          LastName = lastNameValue;
35          Balance = balanceValue;
36       } // end constructor
37    } // end class RecordSerializable
38 } // end namespace BankLibrary
```

Record Serializable.cs

(2 of 2)

Fig. 19.13 | `RecordSerializable` class for serializable objects. (Part 2 of 2.)

19.9 Creating a Sequential-Access File Using Object Serialization (Cont.)

The classes for objects that we wish to serialize must include this attribute in their declarations or must implement interface `ISerializable`.

In a serializable class, you must ensure that every instance variable of the class is also serializable.

All **simple-type variables and `string`s are serializable**.

For variables of **reference types, their types must be serializable**.

By default, **array objects are serializable**. However, if the array contains references to **other objects, those objects may or may not be serializable.**

Using a Serialization Stream to Create an Output File

Now let's create a sequential-access file with serialization.

CreateFile

(1 of 11)

```
1  // Fig: CreateFileForm
2  // Creating a sequential-access file using serialization.
3  using System;
4  using System.Windows;
5  using System.IO;
6  using System.Runtime.Serialization.Formatters.Binary;
7  using System.Runtime.Serialization;
8  using BankLibraryUI;
9  using Microsoft.Win32;
10 namespace CreateFile
11 {
12    public partial class MainWindow : Window
13    {
14       // object for serializing Records in binary format
15       private BinaryFormatter formatter = new BinaryFormatter();
16       private FileStream output; // stream for writing to a file
17
```

Create a `BinaryFormatter` for writing serialized objects.

Fig. 19.14 | Sequential file created using serialization. (Part 1 of 10.)

CreateFile

(2 of 11)

```
18      // parameterless constructor
19      public MainWindow()
20      {
21         InitializeComponent();
22      } // end constructor
23
24      // handler for saveButton_Click
25      private void BtnSave_Click( object sender, RoutedEventArgs e )
26      {
27         // create and show dialog box enabling user to save file
28         bool? result;
29         string fileName; // name of file to save data
30
31         SaveFileDialog fileChooser = new SaveFileDialog()[
32
33            fileChooser.CheckFileExists = false; // let user create file
34
35            // retrieve the result of the dialog box
36            result = fileChooser.ShowDialog();
37            fileName = fileChooser.FileName; // get specified file name
38
```

Fig. 19.14 | Sequential file created using serialization. (Part 2 of 10.)

Outline

CreateFile

(3 of 11)

```
39
40          // ensure that user clicked "OK"
41          if ( result.HasValue )
42          {
43
44             // show error if user specified invalid file
45             if ( fileName == string.Empty )
46                MessageBox.Show( "Invalid File Name", "Error",
47                   MessageBoxButton.OK, MessageBoxImage.Error );
48             else
49             {
50                // save file via FileStream if user specified valid file
51                try
52                {
53                   // open file with write access
54                   output = new FileStream( fileName,
55                      FileMode.OpenOrCreate, FileAccess.Write );
56
57                   // disable Save button and enable Enter button
58                   BtnSaveFile.IsEnabled = false;
59                   BtnEnter.IsEnabled = true;
60                } // end try
```

Fig. 19.14 | Sequential file created using serialization. (Part 3 of 10.)

CreateFile (4 of 11)

```
61                // handle exception if there is a problem opening the file
62                catch ( IOException )
63                {
64                   // notify user if file could not be opened
65                   MessageBox.Show( "Error opening file", "Error",
66                      MessageBoxButton.OK, MessageBoxImage.Error );
67                } // end catch
68             } // end else
69          } // end if
70       } // end method saveButton_Click
71
72       // handler for enterButton Click
73       private void BtnEnter_Click( object sender, RoutedEventArgs e )
74       {
75          // store TextBox values string array
76          string[] values = BankUIForm.GetTextBoxValues();
77
78          // Record containing TextBox values to serialize
79          RecordSerializable record = new RecordSerializable();
80
```

Fig. 19.14 | Sequential file created using serialization. (Part 4 of 10.)

Outline

CreateFile

(5 of 11)

```
81          // determine whether TextBox account field is empty
82          if ( values[ ( int ) BankUIForm.TextBoxIndices.ACCOUNT ] != string.Empty )
83          {
84             // store TextBox values in Record and serialize Record
85             try
86             {
87                // get account-number value from TextBox
88                int accountNumber = Int32.Parse(
89                   values[ ( int ) BankUIForm.TextBoxIndices.ACCOUNT ] );
90
91                // determine whether accountNumber is valid
92                if ( accountNumber > 0 )
93                {
94                   // store TextBox fields in Record
95                   record.Account = accountNumber;
96                   record.FirstName = values[ ( int )
97                      BankUIForm.TextBoxIndices.FIRST ];
98                   record.LastName = values[ ( int )
99                      BankUIForm.TextBoxIndices.LAST ];
100                  record.Balance = Decimal.Parse( values[
101                     ( int ) BankUIForm.TextBoxIndices.BALANCE ] );
102
```

Fig. 19.14 | Sequential file created using serialization. (Part 5 of 10.)

Outline

```
103                // write Record to FileStream ( serialize object )
104                formatter.Serialize( output, record );
105             } // end if
106             else
107             {
108                // notify user if invalid account number
109                MessageBox.Show( "Invalid Account Number", "Error",
110                   MessageBoxButton.OK, MessageBoxImage.Error );
111             } // end else
112          } // end try
113          // notify user if error occurs in serialization
114          catch ( SerializationException )
115          {
116             MessageBox.Show( "Error Writing to File", "Error",
117                MessageBoxButton.OK, MessageBoxImage.Error );
118          } // end catch
119          // notify user if error occurs regarding parameter format
120          catch ( FormatException )
121          {
122             MessageBox.Show( "Invalid Format", "Error",
123                MessageBoxButton.OK, MessageBoxImage.Error );
124          } // end catch
125       } // end if
```

CreateFile

(6 of 11)

Call method `Serialize` to write the `RecordSerializable` object to the output file.

Fig. 19.14 | Sequential file created using serialization. (Part 6 of 10.)

CreateFile (7 of 11)

```
126
127        BankUIForm.ClearTextBoxes(); // clear TextBox values
128     } // end method enterButton_Click
129
130     // handler for exitButton Click
131     private void BtnExit_Click( object sender, RoutedEventArgs e )
132     {
133        // determine whether file exists
134        if ( output != null )
135        {
136           // close file
137           try
138           {
139              output.Close(); // close FileStream
140           } // end try
141           // notify user of error closing file
142           catch ( IOException )
143           {
144              MessageBox.Show( "Cannot close file", "Error",
145                 MessageBoxButton.OK, MessageBoxImage.Error );
146           } // end catch
147        } // end if
```

Fig. 19.14 | Sequential file created using serialization. (Part 7 of 10.)

Outline

CreateFile

(8 of 11)

```
148
149         System.Environment.Exit(0);
150      } // end method BtnExit_Click
151   } // end class
152} // end namespace
```

a) BankUI graphical user interface with three additional controls

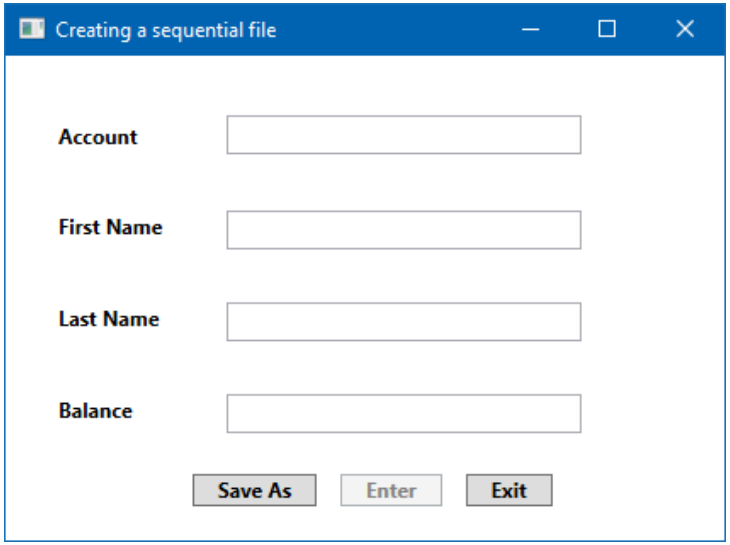


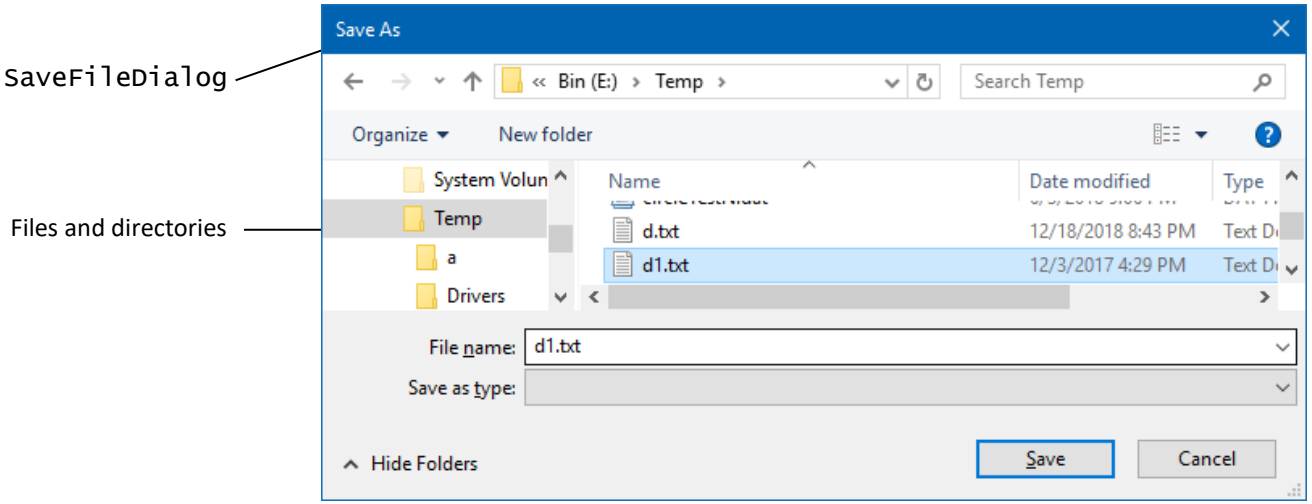
Fig. 19.14 | Sequential file created using serialization. (Part 8 of 10.)

Outline

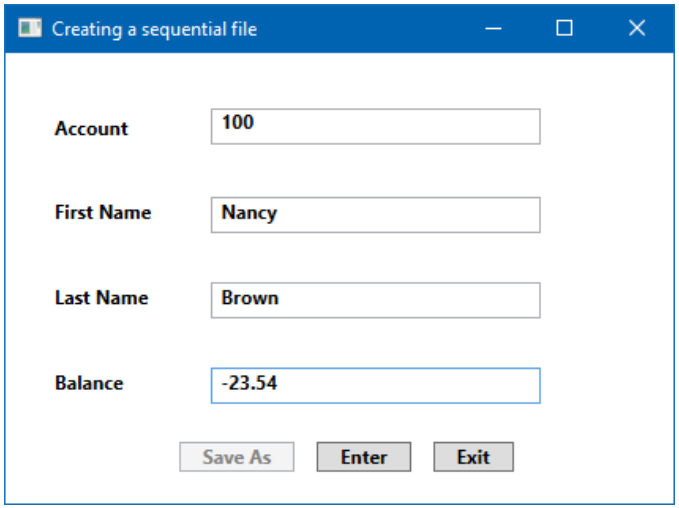
`CreateFile`

(9 of 11)

b) Save File dialog



c) Account 100, "Nancy Brown", saved with a balance of -25.54



d) Account 200, "Stacey Dunn", saved with a balance of 314.33

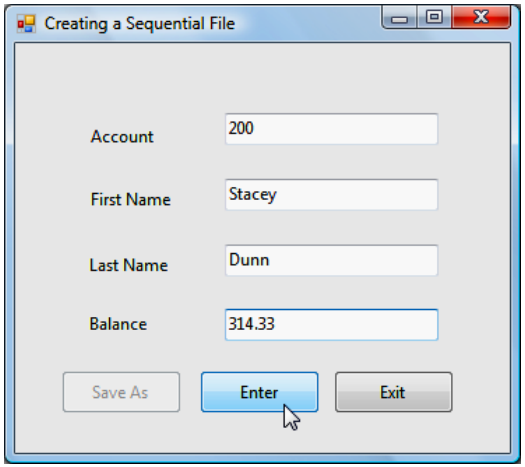
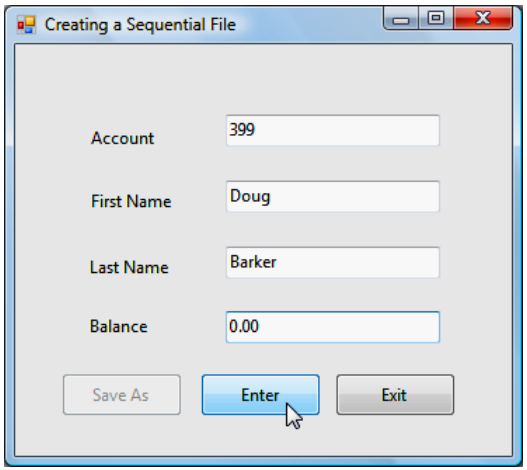


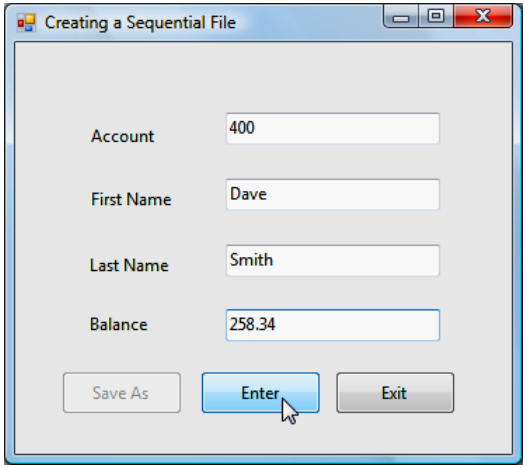
Fig. 19.14 | Sequential file created using serialization. (Part 9 of 10.)

Outline

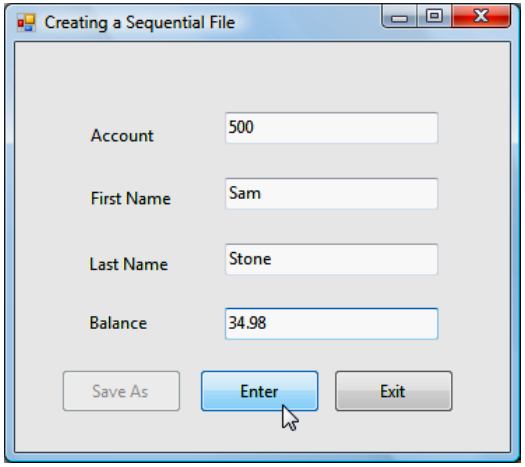
e) Account 399, “Doug Barker”, saved with a balance of 0



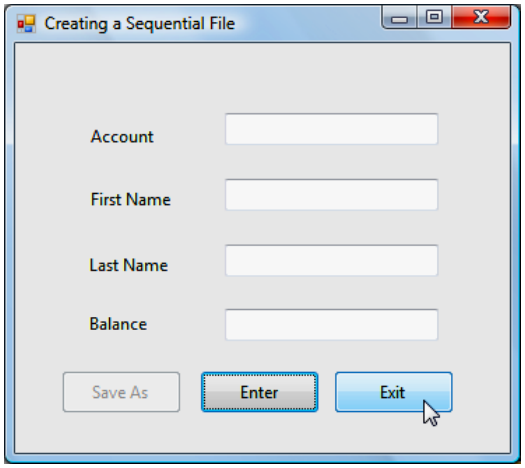
f) Account 400, “Dave Smith”, saved with a balance of 258.34



g) Account 500, “Sam Stone”, saved with a balance of 34.98



h) Once all accounts are saved, the Exit button closes the application



`CreateFile`

(10 of 11)

Fig. 19.14 | Sequential file created using serialization. (Part 10 of 10.)

Common Programming Error 19.2

It is a logic error to open an existing file for output when the user wishes to preserve the file. The original file's contents will be lost.

- Method `Serialize` takes the `FileStream` object as the first argument so that the `BinaryFormatter` can write its second argument to the correct file.
- Remember that we are now using binary files, which are not human readable.

Outline

The application reads and displays the contents of the file created by the program.

ReadSequential

(1 of 8)

```
1  // Fig.  : ReadSequentialAccessFile
2  // Reading a sequential-access file using deserialization.
3  using System;
4  using System.Windows;
5  using System.IO;
6  using System.Runtime.Serialization.Formatters.Binary;
7  using System.Runtime.Serialization;
8  using BankLibraryUI;
9  using Microsoft.Win32;
10 namespace WriteFileSerializable
11 {
12    public partial class MainWindow : Window
13    {
14       // object for deserializing Record in binary format
15       private BinaryFormatter reader = new BinaryFormatter();
16       private FileStream input; // stream for reading from a file
17
```

Create the `BinaryFormatter` that will be used to read objects.

Fig. 19.15 | Sequential file read using deserialization. (Part 1 of 8.)

Outline

ReadSequential

(2 of 8)

```
18       // parameterless constructor
19       public MainWindow()
20       {
21          InitializeComponent();
22       } // end constructor
23
24       // invoked when user clicks the Open button
25       private void BtnOpen_Click( object sender, RoutedEventArgs e )
26       {
27          // create and show dialog box enabling user to open file
28          bool? result; // result of OpenFileDialog
29          string fileName; // name of file containing data
30
31          OpenFileDialog fileChooser = new OpenFileDialog();
32
33             result = fileChooser.ShowDialog();
34             fileName = fileChooser.FileName; // get specified name
35
36
```

Fig. 19.15 | Sequential file read using deserialization. (Part 2 of 8.)

Outline

ReadSequential

(3 of 8)

```
37          // ensure that user clicked "OK"
38          if ( result.HasValue )
39          {
40             BankUIForm.ClearTextBoxes();
41
42             // show error if user specified invalid file
43             if ( fileName == string.Empty )
44                MessageBox.Show( "Invalid File Name", "Error",
45                   MessageBoxButton.OK, MessageBoxImage.Error );
46             else
47             {
48                // create FileStream to obtain read access to file
49                input = new FileStream(
50                   fileName, FileMode.Open, FileAccess.Read );
51
52                BtnOpen.IsEnabled = false; // disable Open File button
53                BtnNext.IsEnabled = true;  // enable Next Record button
54             } // end else
55          } // end if
56       } // end method BtnOpen_Click
57
```

Open the file for input by creating a `FileStream` object.

Fig. 19.15 | Sequential file read using deserialization. (Part 3 of 8.)

ReadSequential

(4 of 8)

```
58       // invoked when user clicks Next button
59       private void BtnNext_Click( object sender, RoutedEventArgs e )
60       {
61          // deserialize Record and store data in TextBoxes
62          try
63          {
64             // get next RecordSerializable available in file
65             RecordSerializable record =
66                ( RecordSerializable ) reader.Deserialize( input );
67
68             // store Record values in temporary string array
69             string[] values = new string[] {
70                record.Account.ToString(),
71                record.FirstName.ToString(),
72                record.LastName.ToString(),
73                record.Balance.ToString()
74             };
75
76             // copy string-array values to TextBox values
77             BankUIForm.SetTextBoxValues( values );
78          } // end try
```

We use method `Deserialize` (of the `BinaryFormatter`) to read the data.

Fig. 19.15 | Sequential file read using deserialization. (Part 4 of 8.)

ReadSequential

(5 of 8)

```
79          // handle exception when there are no Records in file
80          catch ( SerializationException )
81          {
82             input?.Close(); // close FileStream if no Records in file
83             BtnOpen.IsEnabled = true; // enable Open File button
84             BtnNext.IsEnabled = false; // disable Next Record button
85
86             BankUIForm.ClearTextBoxes();
87
88             // notify user if no Records in file
89             MessageBox.Show( "No more records in file", string.Empty,
90                MessageBoxButton.OK, MessageBoxImage.Information );
91          } // end catch
92       } // end method nextButton
93    } // end class ReadSequentialAccessFileF
94 } // end namespace ReadSequentialAccessFile
```

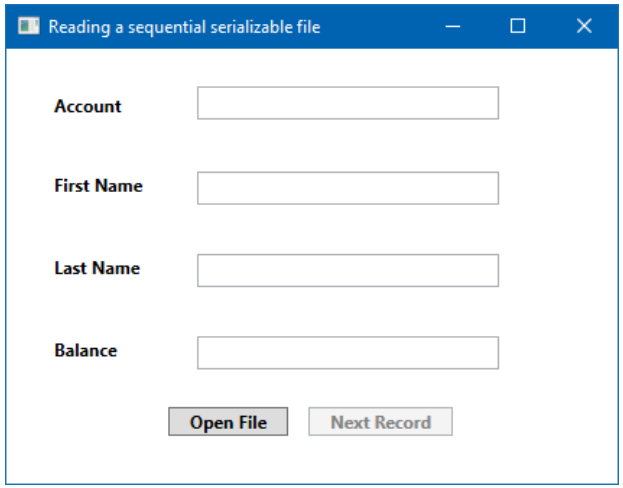
Fig. 19.15 | Sequential file read using deserialization. (Part 5 of 8.)

Outline

ReadSequential

(6 of 8)

a) BankUI graphical user interface with an Open File button



b) `OpenFileDialog` window

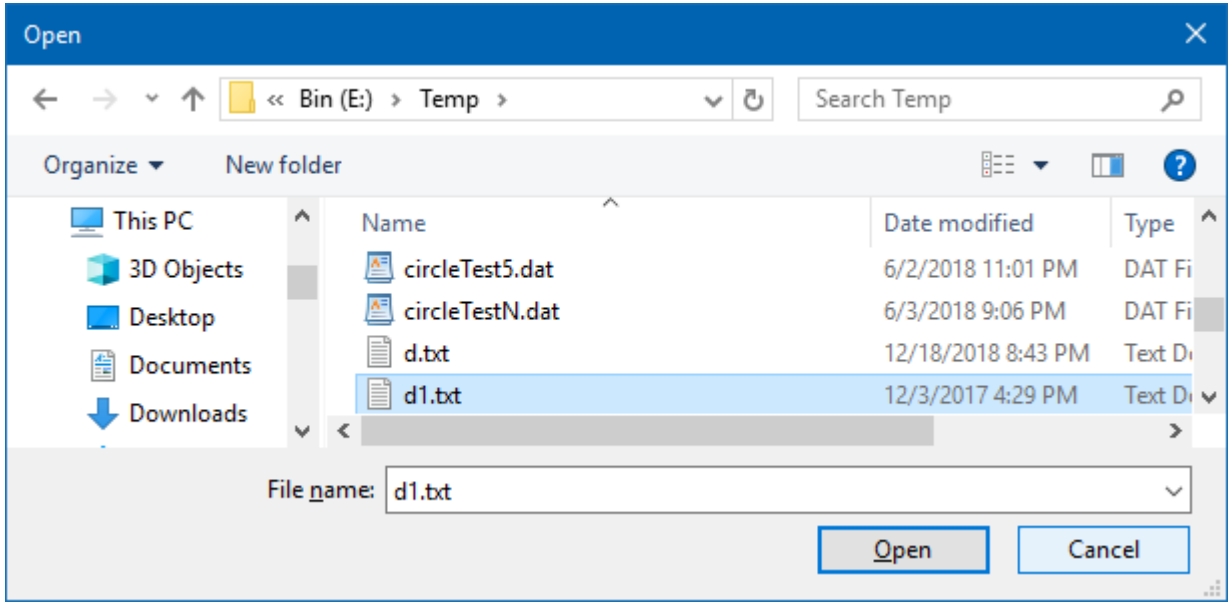


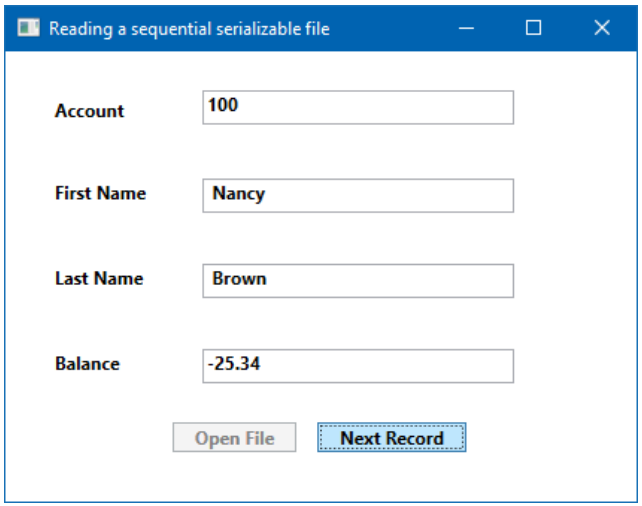
Fig. 19.15 | Sequential file read using deserialization. (Part 6 of 8.)

Outline

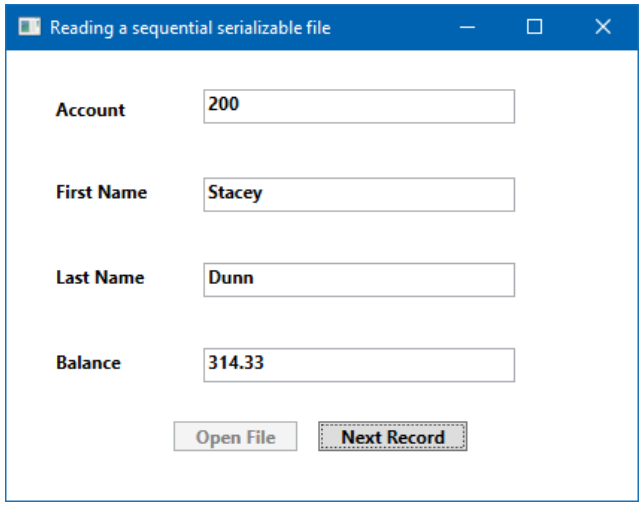
ReadSequential

(7 of 8)

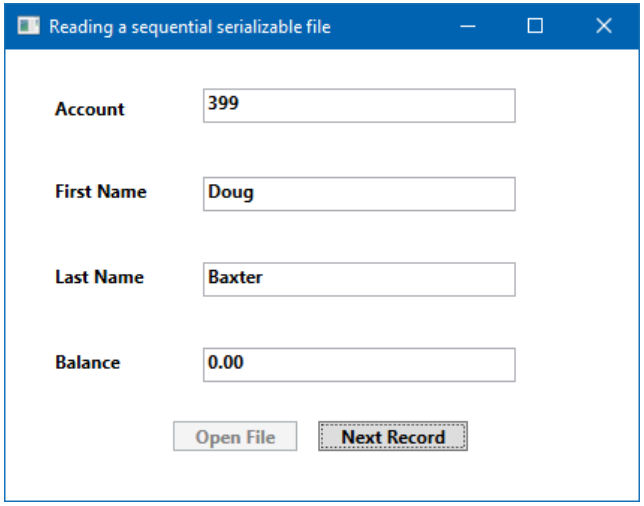
c) Reading account 100



d) Reading account 200



e) Reading account 399



f) Reading account 400

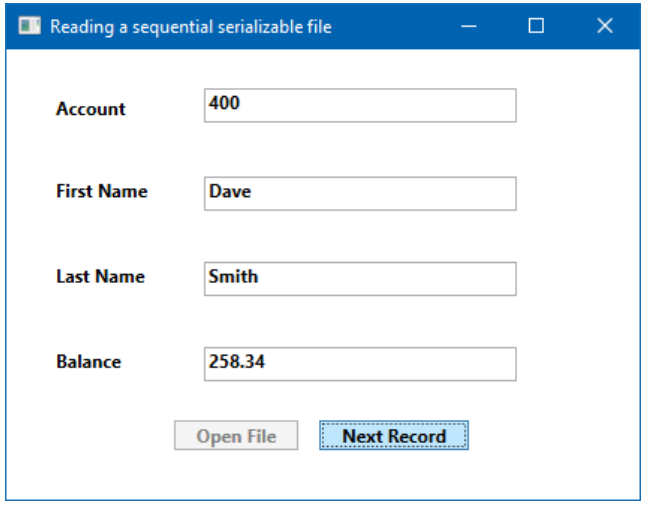
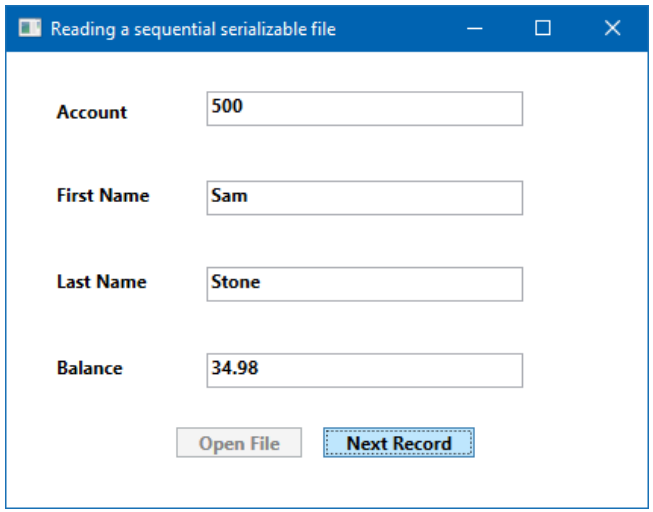


Fig. 19.15 | Sequential file read using deserialization. (Part 7 of 8.)

g) Reading account 500



h) User is shown a messagebox when all records have been read

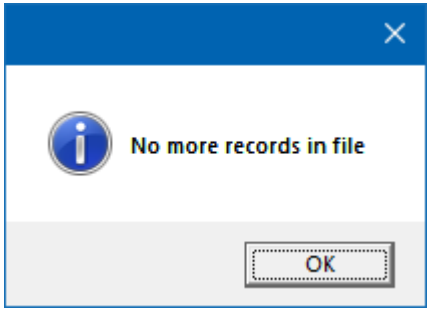


Fig. 19.15 | Sequential file read using deserialization. (Part 8 of 8.)

`Deserialize` returns a reference of type `object`.

If an error occurs during deserialization, a `SerializationException` is thrown.

19.9 XML Serialization

.NET XML serialization enables an object's public fields and properties to be saved and loaded to/from an XML file.

XML serialization is the process of converting an object's public properties and fields to a serial format (in this case, XML) for storage or transport. **Deserialization** re-creates the object in its original state from the XML output.

19.9 XML Serialization

The **data** in your objects are **described using programming language constructs like classes, fields, properties, primitive types, arrays**, and even embedded XML in the form of XmlElement or XmlAttribute objects. Optionally, you may create classes, annotated with attributes, or using the XML Schema Definition Tool (Xsd.exe) to generate the classes based on an existing XML Schema definition (XSD) document. The Xsd.exe tool allows to produce from a given XML Schema the set of classes that are strongly typed in that schema and annotate them with attributes matching the XML Schema when serialized.

19.9 XML Serialization

The transfer of data between objects and XML requires a mapping from the programming language constructs to XML schema and from the XML schema to the programming language constructs. The **XmlSerializer** and related tools like **Xsd.exe** provide the bridge between these two technologies at both design time and runtime. At design time, use the Xsd.exe to produce an XML schema document **(.xsd)** from your custom classes or to produce classes from a given schema.

19.9 XML Serialization

Once the classes are annotated with custom attributes, the **XmlSerializer** "understands" how to map between the XML schema system and the common language runtime.

This way, at runtime, instances of the classes can be serialized into XML documents that follow the given schema. Likewise, these XML documents can be deserialized into runtime objects. Note that the XML schema is optional, and not required at design time or runtime.

For a complete example, study projects **XMLserialization** and **TestXMLreader** in the attached sample code for this lecture.

19.9a Class annotations

Before we can serialize an object to XML, the object's class code must include various custom metadata attributes.

The **XmlRoot** attribute allows you to set an alternate name (**PurchaseOrder**) for the XML root element and its namespace. By default, the **XmlSerializer** uses the class name.

The attribute also allows you to set the **XML namespace** for the element. Lastly, the attribute sets the **IsNullable** property, which specifies whether the **xsi:null** attribute appears if the class instance is set to a null reference.

19.9a Class annotations

```
[XmlRoot ("PurchaseOrder",
          Namespace = "http://fmi.uni-sofia.bg",
          IsNullable = false)]
public class PurchaseOrder
{
  // Set this 'DateTimeValue' field to be an attribute
  // of the root node.
  [XmlAttributeAttribute(DataType = "date")]
  public System.DateTime DateTimeValue;

  // Without specifying any custom Metadata Attributes,
  // fields will be created as an element by default.
  public int CustomerID;
}
```

19.9a Class annotations

Custom Metadata Attributes may be used to rename a field name in the XML document or define an array of elements.

For example

```
// The XmlArray attribute changes the XML element name
// from the default of "OrderedItems" to "Items".
[XmlArray("Items")]
public OrderedItem[]? OrderedItems;

// Serializes an ArrayList as a "Hobbies" array of XML elements
// of type string named "Hobby".
[XmlArray("Hobbies"), XmlArrayItem("Hobby", typeof(string))]
public List<string> Hobbies = new  ();
```

19.9b The XmlSerializer class

The **XmlSerializer** class is used to serialize and deserialize objects into and from XML documents. The **XmlSerializer** enables you to control how objects are encoded into XML.

The constructor

XmlSerializer(*typeof(<classname>)***)**

initializes a new instance of the **XmlSerializer** class that can serialize objects of the specified type into XML documents, and deserialize XML documents into objects of the specified type using its methods

Serialize(*Stream*, *object***)** and **Deserialize(***Stream***).**

19.9b The XmlSerializer class

The **XmlSerializer** class publishes events that may be used to handle unknown element (**UnknownElement**) or attribute (**UnknownAttribute**) .

For example

```
XmlSerializer ser = new XmlSerializer(typeof(PurchaseOrder));

// Add a delegate to handle unknown element events.
ser.UnknownElement+=new XmlElementEventHandler(Serializer_UnknownElement);

// Add a delegate to handle unknown element events.
ser.UnknownAttribute+=new XmlAttributeEventHandler(Serializer_UnknownAttribute);
```

19.9c The XmlReader class

XmlReader represents a reader that provides fast, noncached, **forward-only** access to XML data.

The **XmlReader** is available in the **System.Xml** namespace.

XmlReader methods let you move through XML data and read the contents of a node. The properties of the class reflect the value of the current node, which is where the reader is positioned.

Use the **Create()** method to create an **XmlReader** instance.

Next locate the position in the XML from where reading should start (for example, use method **ReadStartElement()** to check that the current node is an element and advance the reader to the next node)

Further on, use an **XmlSerializer** instance to read sequentially the XML document using its method **Deserialize().**

19.9c The XmlReader class

```
// create FileStream to obtain read access to file
var input = new FileStream(fileName,
                           FileMode.Open, FileAccess.Read);
var xmlReader = XmlReader.Create(input);
// Read the root element
xmlReader.ReadStartElement();
XmlSerializer serializer = new XmlSerializer(typeof(PurchaseOrder));
// read sequentially the XML document until the end of document
if (xmlReader?.NodeType != XmlNodeType.EndElement)
  var purchase =
              serializer?.Deserialize(xmlReader!) as PurchaseOrder;
else{
  // close the XmlReader and the FileStream when done
  xmlReader?.Close();
  input?.Close();
  }
```

19.9c The XmlWriter class

XmlWriter represents a writer that provides a fast, non-cached, forward-only way to generate streams or files that contain XML data.

The **XmlWriter** is available in the **System.Xml** namespace.

XmlWriter class writes XML data to a stream, file, text reader, or string.

Use the **Create()** method to create an **XmlWriter** instance, where the **XmlWriterSettings** class includes several properties that control how **XmlWriter** output is formatted (encoding, indentation etc)

Next indicate the position in the XML from where reading should start (for example, use method **WriteStartElement(***stringOfLocalName***)** to write out a start tag with the specified local name.)

Further on, use an **XmlSerializer** instance to write sequentially the XML document using its method **Serialize().**

19.9c The XmlWriter class

```
// open file with write access
output = new FileStream(fileName,
                        FileMode.OpenOrCreate, FileAccess.Write);
// Default XmlWriterSettings
XmlWriterSettings settings = new XmlWriterSettings();
settings.Async = false;
xmlWriter = XmlWriter.Create(output, settings);
xmlWriter.WriteStartElement("Purchases");
XmlSerializer serializer = new XmlSerializer(typeof(PurchaseOrder));
// write sequentially the XML document
var purchase = new PurchaseOrder();
serializer?.Serialize(xmlWriter!, purchase);

// close the XmlWriter and the FileStream when done
xmlWriter?.Close();
input?.Close();
```

19.9d Sample program

Projects **WriteToXMLfile** and **ReadFromXMLfile** attached in the sample code accompanying this lecture demonstrate XML serialization in WPF.

These projects use the same **BankUI** user control as in the previous case studies for **File** management of class **Record** instances.