

Software Development: Object Oriented Programming

(H171 35)

Files and Streams

Objectives:

* To be able to create, read, write and update files
* To understand the C# streams class hierarchy
* To be able to use classes File and Directory
* To be able to use the FileStream and BinaryFormatter classes to read objects from, and write objects to, files
* To become familiar with sequential-access and random-access file processing

**A direct extract from Chapter 17,”C# How To Program”, Deitel (ISBN 0-13-062221-4)**

Contents Page

Introduction Page 1

Data Hierarchy Page 2

Files and Streams Page 5

Classes File and Directory Page 7

Fig. 17.5 Testing classes File and Directory Page 9

Fig. 17.5 Testing classes File and Directory code walkthrough Page 12

Fig. 17.6 Regular expression used to determine file types Page 13

Fig. 17.6 Regular expression used to determine file types

code walkthrough Page 17

Creating a Sequential -Access File Page 19

Fig. 17.7 Base class for GUIs in our file-processing applications Page 19

Fig. 17.7 Base class for GUIs in our file-processing applications

code walkthrough Page 22

Fig. 17.8 Record for sequential-access file-processing applications Page 22

Fig. 17.8 Record for sequential-access file-processing applications

code walkthrough Page 24

Fig. 17.9 Create and write to a sequential-access file Page 25

Fig. 17.9 Create and write to a sequential-access file

code walkthrough Page 29

Reading Data from a Sequential - Access File Page 32

Fig. 17.11 Reading sequential-access files Page 32

Fig. 17.11 Reading sequential-access files code walkthrough Page 35

Fig. 17.12 Credit-inquiry program Page 37

Fig. 17.12 Credit-inquiry program code walkthrough Page 41

Random-Access Files Page 42

Fig. 17.14 Record for random-access file-processing applications Page 43

Fig. 17.14 Record for random-access file-processing applications

code walkthrough Page 46

Creating a Random-Access File Page 47

Fig. 17.15 Creating files for random-access file-processing applications Page 47

Fig. 17.15 Creating files for random-access file-processing applications

code walkthrough Page 49

Writing Data Randomly to a Random-Access File Page 51

Fig. 17.16 Writing records to random-access files Page 51

Fig. 17.16 Writing records to random-access files code walkthrough Page 54

Reading Data Sequentially from a Random-Access File Page 55

Fig. 17.17 Reading records from random-access files sequentially Page 55

Fig. 17.17 Reading records from random-access files sequentially

code walkthrough Page 58

Case Study: A Transaction-Processing Program Page 60

Fig. 17.18 Record-transaction class

for the transaction-processor case study Page 61

Fig. 17.19 TransactionProcessorForm class runs

the transaction-processor application Page 65

Fig. 17.20 StartDialogForm class enables users to access dialog boxes

associated with various transactions Page 66

Fig. 17.21 UpdateDialogForm class enables users to update records

in transaction-processor case study Page 69

Fig. 17.22 NewDialogForm class enables users to create records

in transaction-processor case study Page 74

Fig. 17.23 DeleteDialogForm class enables users to remove

records from files in transaction-processor case study Page 77

Case Study discussion and Figs. 17.18 through to 17.23

code walkthroughs Page 80

Files and Streams: A Summary Page 86

Self-Review Exercises Page 90

Introduction

Variables and arrays offer only temporary storage of data - the data are lost when an object is garbage collected or when the program terminates

By contrast, files are used for long-term storage of large amounts of data and can retain data even after the program that created the data terminates

Data maintained in files often are called persistent data

Computers can store files on secondary storage devices, such as magnetic disks, optical disks and magnetic tapes

In this booklet, we explain how to create, update and process data files in C# programs

We consider both “sequential-access” files and “random-access” files, indicating the kinds of applications for which each is best suited

File processing is one of a programming language’s most important capabilities, because it enables a language to support commercial applications that typically process massive amounts of persistent data

This booklet discusses C#’s powerful and abundant file-processing and stream-input/output features

Data Hierarchy

Ultimately, all data items processed by a computer are reduced to combinations of zeros and ones

This is because it is simple and economical to build electronic devices that can assume two stable states - 0 represents one state, and 1 represents the other

It is remarkable that the impressive functions performed by computers involve only the most fundamental manipulations of 0s and 1s

The smallest data items that computers support are called bits

(short for “binary digit” - a digit that can assume one of two values)

Each data item, or bit, can assume either the value 0 or the value 1

Computer circuitry performs various simple bit manipulations, such as examining the value of a bit, setting the value of a bit and reversing a bit (from 1 to 0 or from 0 to 1)

Programming with data in the low-level form of bits is cumbersome

It is preferable to program with data in forms such as **decimal digits**

(i.e., 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9),

**letters**

(i.e., A through Z and a through z) and

**special symbols**

(i.e., $, @, %, &, \*, (, ), -, +, ", :, ?, / and many others)

Digits, letters and special symbols are referred to as characters

The set of all characters used to write programs and represent data items on a particular computer is called that computer’s character set

Because computers can process only 1s and 0s, every character in a computer’s character set is represented as a pattern of 1s and 0s

Bytes are composed of eight bits (characters in C# are Unicode characters, which are composed of 2 bytes)

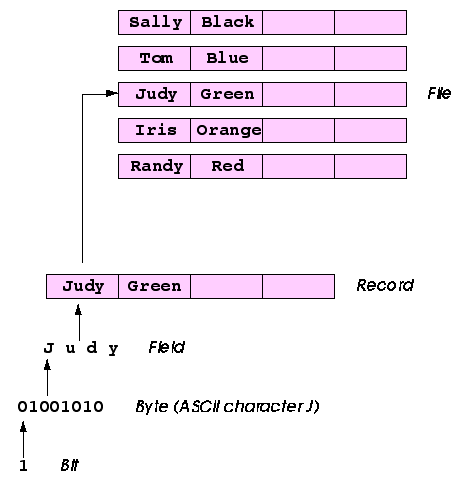
Programmers create programs and data items with characters; computers manipulate and process these characters as patterns of bits

In the same way that characters are composed of bits, fields are composed of characters

A field is a group of characters that conveys some meaning

For example, a field consisting of uppercase and lowercase letters can represent a person’s name

The various kinds of data items processed by computers form a data hierarchy, shown in the diagram below, in which data items become larger and more complex in structure as we progress from bits, to characters, to fields and up to larger data structures:



**Data Hierarchy**

Typically, a record is composed of several fields

In a payroll system, for example, a record for a particular employee might include the following fields:

1. Employee identification number
2. Name
3. Address
4. Hourly pay rate
5. Number of exemptions claimed
6. Year-to-date earnings
7. Amount of taxes withheld

Thus, a record is a group of related fields

In the preceding example, each field is associated with the same employee

A file is a group of related records

A company’s payroll file normally contains one record for each employee

Thus, a payroll file for a small company might contain only 22 records, whereas a payroll file for a large company might contain 100,000 records

It is not unusual for a company to have many files, some containing millions, billions or even trillions of bits of information

To facilitate the retrieval of specific records from a file, at least one field in each record is chosen as a unique record key

A record key identifies a record as belonging to a particular person or entity and distinguishes that record from all other records

In the payroll record described previously, the employee identification number normally would be chosen as the record key

There are many ways of organizing records in a file

The most common type of organization is called a sequential file, in which records typically are stored in order by the record-key field

In a payroll file, records usually are placed in order by employee-identification numbers

The first employee record in the file contains the lowest employee-identification number, and subsequent records contain increasingly higher employee-identification numbers

Most businesses use many different files to store data

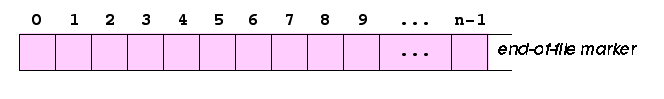
For example, a company might have payroll files, accounts receivable files (listing money due from clients), accounts payable files (listing money due to suppliers), inventory files (listing facts about all the items handled by the business) and many other types of files

Sometimes, a group of related files is called a database

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

Files and Streams

C# views each file as a sequential stream of bytes as shown in the diagram below:



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, C# creates an object, then associates a stream with that object

The runtime environment creates three stream objects upon program execution, which are accessible via properties Console.Out, Console.In and Console.Error, respectively

These objects facilitate communication between a program and a particular file or device

Property Console.In returns the standard input stream object, which enables a program to input data from the keyboard

Property Console.Out returns the standard output stream object, which enables a program to output data to the screen

Property Console.Error returns the standard error stream object, which enables a program to output error messages to the screen

We have been using Console.Out and Console.In in our console applications; Console methods Write and WriteLine use Console.Out to perform output, and methods Read and ReadLine use Console.In to perform input

To perform file processing in C#, namespace **System.IO** must be referenced

This namespace includes definitions for stream classes such as **StreamReader** (for text input from a file), **StreamWriter** (for text output to a file) and **FileStream** (for both input from and output to a file)

Files are opened by creating objects of these stream classes, which inherit from abstract classes **TextReader**, **TextWriter** and **Stream**, respectively

Actually, Console.In and Console.Out are properties of class TextReader and TextWriter, respectively

C# provides class **BinaryFormatter**, which is used in conjunction with a Stream object to perform input and output of objects

Serialization involves converting an object into a format that can be written to a file without losing any of that object’s data

Deserialization consists of reading this format from a file and reconstructing the original object from it

A BinaryFormatter can serialize objects to, and deserialize objects from, a specified Stream

Class System.IO.Stream provides functionality for representing streams as bytes

This class is abstract, so objects of this class cannot be instantiated

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

Later in the booklet, we use FileStream to read data to, and write data from, sequential-access and random-access files

Class MemoryStream enables the transferal of data directly to and from memory - this type of transfer is much faster than are other types of data transfer

(e.g., to and from disk)

Class BufferedStream uses buffering to transfer data to or from a stream

Buffering is an I/O-performance-enhancement technique in which each output operation is directed to a region in memory called a buffer that is large enough to hold the data from many output operations

Then, 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

C# offers many classes for performing input and output

In this booklet, we use several key stream classes to implement a variety of file-processing programs that create, manipulate and destroy sequential-access files and random-access files

Classes File and Directory

Information on computers is stored in files, which are organized in directories

Class **File** is provided for manipulating files, and class **Directory** is provided for manipulating directories

Class File cannot write to or read from files directly; we discuss methods for reading and writing files in subsequent sections

Note that the \ separator character separates directories and files in a path

On UNIX systems, the separator character is /

C# actually processes both characters as identical in a path name

This means that, if we specified the path c:\C\_Sharp/README, which uses one of each separator character, C# still would process the file properly

The table below lists some methods contained in class File for manipulating and determining information about particular files

Class File contains only static methods—you cannot instantiate objects of type File

We use several of these methods in the code example of Fig. [17.5](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.5','codesampleswin');)

|  |  |
| --- | --- |
| **static Method** | **Description** |
| AppendText | Returns a StreamWriter that appends 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 |
| GetCreationTime | Returns a DateTime object representing the time that the file was created |
| GetLastAccessTime | Returns a DateTime object representing the time that the file was last accessed |
| GetLastWriteTime | Returns a DateTime object representing the time that 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 |

**File class methods (partial list)**

Class Directory provides capabilities for manipulating directories

The table below lists some methods that can be used for directory manipulation

We employ several of these methods in the code example of Fig. [17.5](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.5','codesampleswin');)

|  |  |
| --- | --- |
| **static Method** | **Description** |
| CreateDirectory | Creates a directory and returns its associated DirectoryInfo |
| Delete | Deletes the specified directory |
| Exists | Returns true if the specified directory exists; otherwise, it returns false |
| GetLastWriteTime | Returns a DateTime object representing the time that the directory was last modified |
| GetDirectories | Returns a string array representing the names of the subdirectories in the specified directory |
| GetFiles | Returns a string array representing the names of the files in the specified directory |
| GetCreationTime | Returns a DateTime object representing the time that the directory was created |
| GetLastAccessTime | Returns a DateTime object representing the time that the directory was last accessed |
| GetLastWriteTime | Returns a DateTime object representing the time that items were last written to the directory |
| Move | Moves the specified directory to a specified location |

**Directory class methods (partial list)**

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

Much of the information contained in this class also can be accessed via the methods of class Directory

Class FileTestForm created below in Fig. [17.5](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.5','codesampleswin');) uses methods described in the above tables to access file and directory information

Fig. 17.5 Testing classes File and Directory

1 // Fig 17.5: FileTest.cs  
 2 // Using classes File and Directory.  
 3   
 4 using System;  
 5 using System.Drawing;  
 6 using System.Collections;  
 7 using System.ComponentModel;  
 8 using System.Windows.Forms;  
 9 using System.Data;  
 10 using System.IO;  
 11   
 12 // displays contents of files and directories  
 13 public class FileTestForm : System.Windows.Forms.Form  
 14 {  
 15 private System.Windows.Forms.Label directionsLabel;  
 16   
 17 private System.Windows.Forms.TextBox outputTextBox;  
 18 private System.Windows.Forms.TextBox inputTextBox;  
 19   
 20 private System.ComponentModel.Container components = null;  
 21   
 22 [STAThread]  
 23 static void Main()   
 24 {  
 25 Application.Run( new FileTestForm() );  
 26 }  
 27   
 28 // Visual Studio .NET generated code  
 29   
 30 // invoked when user presses key  
 31 private void inputTextBox\_KeyDown(  
 32 object sender, System.Windows.Forms.KeyEventArgs e )  
 33 {  
 34 // determine whether user pressed Enter key  
 35 if ( e.KeyCode == Keys.Enter )  
 36 {  
 37 string fileName; // name of file or directory  
 38   
 39 // get user-specified file or directory  
 40 fileName = inputTextBox.Text;  
 41   
 42 // determine whether fileName is a file  
 43 if ( File.Exists( fileName ) )  
 44 {  
 45 // get file's creation date,   
 46 // modification date, etc.  
 47 outputTextBox.Text = GetInformation(

fileName );  
 48   
 49 // display file contents through StreamReader  
 50 try  
 51 {  
 52 // obtain reader and file contents  
 53 StreamReader stream = new StreamReader( fileName );  
 54 outputTextBox.Text += stream.ReadToEnd();  
 55 }  
 56 // handle exception if StreamReader is unavailable  
 57 catch( IOException )  
 58 {  
 59 MessageBox.Show( "File Error", "File Error",  
 60 MessageBoxButtons.OK, MessageBoxIcon.Error );  
 61 }  
 62 }  
 63   
 64 // determine whether fileName is a directory  
 65 else if ( Directory.Exists( fileName ) )  
 66 {  
 67 // array for directories  
 68 string[] directoryList;  
 69   
 70 // get directory's creation date,   
 71 // modification date, etc.  
 72 outputTextBox.Text = GetInformation( fileName );  
 73   
 74 // obtain file/directory list of specified directory  
 75 directoryList = Directory.GetDirectories( fileName );  
 76   
 77 outputTextBox.Text +=   
 78 "\r\n\r\nDirectory contents:\r\n";  
 79   
 80 // output directoryList contents  
 81 for ( int i = 0; i < directoryList.Length; i++ )  
 82 outputTextBox.Text += directoryList[ i ] + "\r\n";  
 83 }  
 84 else  
 85 {  
 86 // notify user that neither file nor directory exists  
 87 MessageBox.Show( inputTextBox.Text +   
 88 " does not exist", "File Error",   
 89 MessageBoxButtons.OK,

MessageBoxIcon.Error );  
 90 }  
 91 } // end if  
 92   
 93 } // end method inputTextBox\_KeyDown  
 94   
 95 // get information on file or directory  
 96 private string GetInformation( string fileName )  
 97 {  
 98 // output that file or directory exists  
 99 string information = fileName + " exists\r\n\r\n";  
 100   
 101 // output when file or directory was created  
 102 information += "Created: " +  
 103 File.GetCreationTime( fileName ) + "\r\n";  
 104   
 105 // output when file or directory was last modified  
 106 information += "Last modified: " +  
 107 File.GetLastWriteTime( fileName ) + "\r\n";  
 108   
 109 // output when file or directory was last accessed  
 110 information += "Last accessed: " +  
 111 File.GetLastAccessTime( fileName ) + "\r\n" + "\r\n";  
 112   
 113 return information;  
 114   
 115 } // end method GetInformation  
 116   
 117 } // end class FileTestForm

Fig. 17.5 Testing classes File and Directory

code walkthrough

Class FileTestForm contains TextBox inputTextBox (line 18), which enables the user to input a file or directory name

For each key that the user presses in the text box, the program calls method inputTextBox\_KeyDown (lines 31 - 93)

If the user presses the Enter key (line 35), this method displays either file or directory contents, depending on the text the user input in the TextBox

(Note that, if the user does not press the Enter key, this method returns without displaying any content)

Line 43 uses method Exists of class File to determine whether the user-specified text is a name of an existing file

If the user specifies an existing file, line 47 invokes private method GetInformation (lines 96 - 115), which calls methods GetCreationTime (line 103), GetLastWriteTime (line 107) and GetLastAccessTime (line 111) of class File to access file information

When method GetInformation returns, line 53 instantiates a StreamReader for reading text from the file

The StreamReader constructor takes as an argument a string containing the name of the file to open

Line 54 calls method ReadToEnd of the StreamReader to read the file content from the file, then displays the content

If line 43 determines that the user-specified text is not a file, line 65 determines whether it is a directory using method Exists of class Directory

If the user specified an existing directory, line 72 invokes method GetInformation to access the directory information

Line 75 calls method GetDirectories of class Directory to obtain a string array containing the names of subdirectories in the specified directory

Lines 81 - 82 display each element in the string array

Note that, if line 65 determines that the user-specified text is neither a file nor a directory, lines 87 - 89 notify the user (via a MessageBox) that the file or directory does not exist

We now consider another example that uses C#’s file- and directory- manipulation capabilities

Class FileSearchForm created below in Fig. [17.6](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.5','codesampleswin');) uses classes File and Directory in conjunction with classes for performing regular expressions to report the number of files of each file type that exist in the specified directory path

The program also serves as a “clean-up” utility - when the program encounters a file that has the .bak extension (i.e., a backup file), the program displays a MessageBox asking whether that file should be removed, then responds appropriately to the user’s input

Fig. 17.6 Regular expression used to determine file types

1 // Fig 17.6: FileSearch.cs  
 2 // Using regular expressions to determine file types.  
 3   
 4 using System;  
 5 using System.Drawing;  
 6 using System.Collections;  
 7 using System.ComponentModel;  
 8 using System.Windows.Forms;  
 9 using System.Data;  
 10 using System.IO;  
 11 using System.Text.RegularExpressions;  
 12 using System.Collections.Specialized;  
 13   
 14 public class FileSearchForm : System.Windows.Forms.Form  
 15 {  
 16 private System.Windows.Forms.Label directionsLabel;  
 17 private System.Windows.Forms.Label directoryLabel;  
 18   
 19 private System.Windows.Forms.Button searchButton;  
 20   
 21 private System.Windows.Forms.TextBox outputTextBox;  
 22 private System.Windows.Forms.TextBox inputTextBox;  
 23   
 24 private System.ComponentModel.Container components = null;  
 25   
 26 string currentDirectory = Directory.GetCurrentDirectory();  
 27 string[] directoryList; // subdirectories  
 28 string[] fileArray;  
 29   
 30 // store extensions found and number found  
 31 NameValueCollection found = new NameValueCollection();  
 32

33 [STAThread]  
 34 static void Main()   
 35 {  
 36 Application.Run( new FileSearchForm() );  
 37 }  
 38   
 39 // Visual Studio .NET generated code  
 40   
 41 // invoked when user types in text box  
 42 private void inputTextBox\_KeyDown(  
 43 object sender, System.Windows.Forms.KeyEventArgs e )  
 44 {  
 45 // determine whether user pressed Enter  
 46 if ( e.KeyCode == Keys.Enter )  
 47 searchButton\_Click( sender, e );  
 48   
 49 } // end method inputTextBox\_KeyDown  
 50   
 51 // invoked when user clicks "Search Directory" button  
 52 private void searchButton\_Click(  
 53 object sender, System.EventArgs e )  
 54 {  
 55 // check for user input; default is current directory  
 56 if ( inputTextBox.Text != "" )  
 57 {  
 58 // verify that user input is valid directory name  
 59 if ( Directory.Exists( inputTextBox.Text ) )  
 60 {  
 61 currentDirectory = inputTextBox.Text;  
 62   
 63 // reset input text box and update display  
 64 directoryLabel.Text = "Current Directory:" +   
 65 "\r\n" + currentDirectory;  
 66 }  
 67 else  
 68 {  
 69 // show error if user does not specify valid directory  
 70 MessageBox.Show( "Invalid Directory", "Error",  
 71 MessageBoxButtons.OK, MessageBoxIcon.Error );  
 72 }  
 73 }  
 74   
 75 // clear text boxes  
 76 inputTextBox.Clear();  
 77 outputTextBox.Clear();  
 78

79 // search directory  
 80 SearchDirectory( currentDirectory );  
 81   
 82 // summarize and print results  
 83 foreach ( string current in found )  
 84 {  
 85 outputTextBox.Text += "\* Found " +   
 86 found[ current ] + " " + current + " files.\r\n";  
 87 }  
 88   
 89 // clear output for new search  
 90 found.Clear();  
 91   
 92 } // end method searchButton\_Click  
 93   
 94 // search directory using regular expression  
 95 private void SearchDirectory( string currentDirectory )  
 96 {  
 97 // search directory  
 98 try  
 99 {  
 100 string fileName = "";  
 101   
 102 // regular expression for extensions matching pattern  
 103 Regex regularExpression = new Regex(   
 104 "[a-zA-Z0-9]+\\.(?<extension>\\w+)" );  
 105   
 106 // stores regular-expression-match result  
 107 Match matchResult;  
 108   
 109 string fileExtension; // holds file extensions  
 110   
 111 // number of files with given extension in directory  
 112 int extensionCount;  
 113   
 114 // get directories  
 115 directoryList =  
 116 Directory.GetDirectories( currentDirectory );  
 117   
 118 // get list of files in current directory  
 119 fileArray = Directory.GetFiles( currentDirectory );  
 120   
 121 // iterate through list of files  
 122 foreach ( string myFile in fileArray )  
 123 {  
 124 // remove directory path from file name  
 125 fileName = myFile.Substring(  
 126 myFile.LastIndexOf( "\\" ) + 1 );  
 127

128 // obtain result for regular-expression search  
 129 matchResult = regularExpression.Match( fileName );  
 130   
 131 // check for match  
 132 if ( matchResult.Success )  
 133 fileExtension =   
 134 matchResult.Result( "${extension}" );  
 135 else  
 136 fileExtension = "[no extension]";  
 137   
 138 // store value from container  
 139 if ( found[ fileExtension ] == null )  
 140 found.Add( fileExtension, "1" );  
 141 else  
 142 {  
 143 extensionCount = Int32.Parse(  
 144 found[ fileExtension ] ) + 1;  
 145   
 146 found[ fileExtension ] = extensionCount.ToString();  
 147 }  
 148   
 149 // search for backup(.bak) files  
 150 if ( fileExtension == "bak" )  
 151 {  
 152 // prompt user to delete (.bak) file  
 153 DialogResult result =  
 154 MessageBox.Show( "Found backup file " +  
 155 fileName + ". Delete?", "Delete Backup",  
 156 MessageBoxButtons.YesNo,   
 157 MessageBoxIcon.Question );  
 158   
 159 // delete file if user clicked 'yes'  
 160 if ( result == DialogResult.Yes )  
 161 {  
 162 File.Delete( myFile );  
 163   
 164 extensionCount =   
 165 Int32.Parse( found[ "bak" ] ) - 1;  
 166   
 167 found[ "bak" ] = extensionCount.ToString();  
 168 }  
 169 }  
 170 }  
 171   
 172 // recursive call to search files in subdirectory  
 173 foreach ( string myDirectory in directoryList )

174 SearchDirectory( myDirectory );  
 175 }  
 176   
 177 // handle exception if files have unauthorized access  
 178 catch( UnauthorizedAccessException )  
 179 {  
 180 MessageBox.Show( "Some files may not be visible" +  
 181 " due to permission settings", "Warning",  
 182 MessageBoxButtons.OK, MessageBoxIcon.Information );  
 183 }  
 184   
 185 } // end method SearchDirectory  
 186   
 187 } // end class FileSearchForm

Fig. 17.6 Regular expression used to determine file types

code walkthrough

When the user presses the Enter key or clicks the Search Directory button, the program invokes method searchButton\_Click (lines 52 - 92), which searches recursively through the directory path that the user provides

If the user inputs text in the TextBox, line 59 calls method Exists of class Directory to determine whether that text indicates a valid directory

If the user specifies an invalid directory, lines 70 - 71 notify the user of the error

If the user specifies a valid directory, line 80 passes the directory name as an argument to private method SearchDirectory (lines 95 - 185)

This method locates files that match the regular expression defined in lines 103 - 104, which matches any sequence of numbers or letters followed by a period and one or more letters

Notice the substring of format (?<extension>regular-expression) in the argument to the Regex constructor (line 104)

All strings with the substring regular-expression are tagged with the name extension

In this program, we assign to the variable extension any string matching one or more characters

Lines 115 - 116 call method GetDirectories of class Directory to retrieve the names of all subdirectories that belong to the current directory

Line 119 calls method GetFiles of class Directory to store in string array fileArray the names of files in the current directory

The foreach loop in lines 122 - 170 searches for all files with extension bak; it then calls SearchDirectory recursively for each subdirectory in the current directory

Lines 125 - 126 eliminate the directory path, so the program can test only the file name when using the regular expression

Line 129 uses method Match of the Regex object to match the regular expression with the file name, then returns the result to object matchResult of type Match

If the match is successful, lines 133 - 134 use method Result of object matchResult to store the extension string from object matchResult in fileExtension (the string that will contain the current file’s extension)

If the match is unsuccessful, line 136 sets fileExtension to hold a value of

"[no extension]"

Class FileSearchForm uses an instance of class NameValueCollection (declared in line 31) to store each file-extension type and the number of files for each type

A NameValueCollection contains a collection of key/value pairs, each of which is a string, and provides method Add to add a key/value pair

The indexer for this pair can index according to the order that the items were added or according to the entry key

Line 139 uses NameValueCollection found to determine whether this is the first occurrence of the file extension

If so, line 140 adds that extension to found as a key with the value 1

If the extension is in found already, lines 143 - 144 increment the value associated with the extension in found to indicate another occurrence of that file extension

Line 150 determines whether fileExtension equals “bak” - i.e., whether the file is a backup file

If so, lines 153 - 157 prompt the user to indicate whether the file should be removed; if the user clicks Yes (line 160), lines 162 - 167 delete the file and decrement the value for the “bak” file type in found

Lines 173 - 174 call method SearchDirectory for each subdirectory

Using recursion, we ensure that the program performs the same logic for finding bak files on each subdirectory

After each subdirectory has been checked for bak files, method SearchDirectory completes, and lines 83 -87 display the results

Creating a Sequential -Access File

C# imposes no structure on files

Thus, concepts like that of a “record” do not exist in C# files

This means that the programmer must structure files to meet the requirements of applications

In this example, we use text and special characters to organize our own concept of a “record”

The following code examples demonstrate file processing in a bank-account maintenance application

These programs have similar user interfaces, so we created class BankUIForm in

Fig. [17.7](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.7','codesampleswin');) to encapsulate a base-class GUI

(see the screen capture entitled “Base Class for GUIs used in our file – processing applications”, below)

Fig. 17.7 Base class for GUIs in our file-processing applications

1 // Fig 17.7: BankUI.cs  
 2 // A reusable windows form for the examples in this booklet.  
 3   
 4 using System;  
 5 using System.Drawing;  
 6 using System.Collections;  
 7 using System.ComponentModel;  
 8 using System.Windows.Forms;  
 9 using System.Data;  
 10   
 11 public class BankUIForm : System.Windows.Forms.Form  
 12 {  
 13 private System.ComponentModel.Container components = null;  
 14   
 15 public System.Windows.Forms.Label accountLabel;  
 16 public System.Windows.Forms.TextBox accountTextBox;  
 17   
 18 public System.Windows.Forms.Label firstNameLabel;  
 19 public System.Windows.Forms.TextBox firstNameTextBox;  
 20   
 21 public System.Windows.Forms.Label lastNameLabel;  
 22 public System.Windows.Forms.TextBox lastNameTextBox;  
 23   
 24 public System.Windows.Forms.Label balanceLabel;  
 25 public System.Windows.Forms.TextBox balanceTextBox;

26   
 27 // number of TextBoxes on Form'  
 28 protected int TextBoxCount = 4;  
 29   
 30 // enumeration constants specify TextBox indices  
 31 public enum TextBoxIndices  
 32 {  
 33 ACCOUNT,  
 34 FIRST,  
 35 LAST,  
 36 BALANCE  
 37   
 38 } // end enum  
 39   
 40 [STAThread]  
 41 static void Main()   
 42 {  
 43 Application.Run( new BankUIForm() );  
 44 }  
 45   
 46 // Visual Studio .NET generated code  
 47   
 48 // clear all TextBoxes  
 49 public void ClearTextBoxes()  
 50 {  
 51 // iterate through every Control on form  
 52 for ( int i = 0; i < Controls.Count; i++ )  
 53 {  
 54 Control myControl = Controls[ i ]; // get control  
 55   
 56 // determine whether Control is TextBox  
 57 if ( myControl is TextBox )  
 58 {  
 59 // clear Text property (set to empty strng)  
 60 myControl.Text = "";  
 61 }  
 62 }  
 63   
 64 } // end method ClearTextBoxes  
 65   
 66 // set text box values to string array values  
 67 public void SetTextBoxValues( string[] values )  
 68 {  
 69 // determine whether string array has correct length  
 70 if ( values.Length != TextBoxCount )  
 71 {

72 // throw exception if not correct length  
 73 throw( new ArgumentException( "There must be " +  
 74 (TextBoxCount + 1) + " strings in the array" ) );  
 75 }  
 76   
 77 // set array values if array has correct length  
 78 else  
 79 {  
 80 // set array values to text box values  
 81 accountTextBox.Text =   
 82 values[ ( int )TextBoxIndices.ACCOUNT ];  
 83 firstNameTextBox.Text =   
 84 values[ ( int )TextBoxIndices.FIRST ];  
 85 lastNameTextBox.Text =   
 86 values[ ( int )TextBoxIndices.LAST ];  
 87 balanceTextBox.Text =   
 88 values[ ( int )TextBoxIndices.BALANCE ];  
 89 }  
 90   
 91 } // end method SetTextBoxValues  
 92   
 93 // return text box values as string array  
 94 public string[] GetTextBoxValues()  
 95 {  
 96 string[] values = new string[ TextBoxCount ];  
 97   
 98 // copy text box fields to string array  
 99 values[ ( int )TextBoxIndices.ACCOUNT ] =   
 100 accountTextBox.Text;  
 101 values[ ( int )TextBoxIndices.FIRST ] =   
 102 firstNameTextBox.Text;  
 103 values[ ( int )TextBoxIndices.LAST ] =   
 104 lastNameTextBox.Text;  
 105 values[ ( int )TextBoxIndices.BALANCE ] =   
 106 balanceTextBox.Text;  
 107   
 108 return values;  
 109   
 110 } // end method GetTextBoxValues  
 111   
 112 } // end class BankUIForm

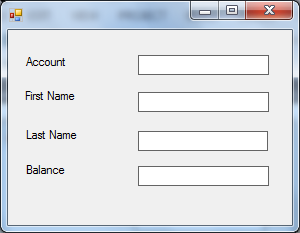
Fig. 17.7 Base class for GUIs in our file-processing applications

code walkthrough

Class BankUIForm contains four Labels (lines 15, 18, 21 and 24) and

four TextBoxes (lines 16, 19, 22 and 25)

Methods ClearTextBoxes (lines 49 - 64), SetTextBoxValues (lines 67–91) and GetTextBoxValues (lines 94 - 110) clear, set the values of, and get the values of the text in the TextBoxes, respectively



**“Base Class for GUIs used in our file – processing applications”**

To reuse class BankUIForm, we compile the GUI into a DLL library by creating a project of type Windows Control Library (the DLL we create is called BankLibrary)

This library, is available from your lecturer

Figure [17.8](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.8','codesampleswin');) contains class Record that Fig. [17.9](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.9','codesampleswin');), Fig. [17.11](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.11','codesampleswin');) and Fig. [17.12](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.12','codesampleswin');) use for reading records from, and writing records to, a file sequentially

This class also belongs to the BankLibrary DLL, so it is located in the same project as is class BankUIForm

Fig. 17.8 Record for sequential-access file-processing applications

1 // Fig. 17.8: Record.cs  
 2 // Serializable class that represents a data record.  
 3   
 4 using System;  
 5   
 6 [Serializable]  
 7 public class Record  
 8 {  
 9 private int account;  
 10 private string firstName;  
 11 private string lastName;  
 12 private double balance;  
 13

14 // default constructor sets members to default values  
 15 public Record() : this( 0, "", "", 0.0 )  
 16 {  
 17 }  
 18   
 19 // overloaded constructor sets members to parameter values  
 20 public Record( int accountValue, string firstNameValue,  
 21 string lastNameValue, double balanceValue )  
 22 {  
 23 Account = accountValue;  
 24 FirstName = firstNameValue;  
 25 LastName = lastNameValue;  
 26 Balance = balanceValue;  
 27   
 28 } // end constructor  
 29   
 30 // property Account  
 31 public int Account  
 32 {  
 33 get  
 34 {  
 35 return account;  
 36 }  
 37   
 38 set  
 39 {  
 40 account = value;  
 41 }  
 42   
 43 } // end property Account  
 44   
 45 // property FirstName  
 46 public string FirstName  
 47 {  
 48 get  
 49 {  
 50 return firstName;  
 51 }  
 52   
 53 set  
 54 {  
 55 firstName = value;  
 56 }  
 57   
 58 } // end property FirstName  
 59   
 60 // property LastName  
 61 public string LastName  
 62 {  
 63 get  
 64 {  
 65 return lastName;  
 66 }

67   
 68 set  
 69 {  
 70 lastName = value;  
 71 }  
 72   
 73 } // end property LastName  
 74   
 75 // property Balance  
 76 public double Balance  
 77 {  
 78 get  
 79 {  
 80 return balance;  
 81 }  
 82   
 83 set  
 84 {  
 85 balance = value;  
 86 }  
 87   
 88 } // end property Balance  
 89   
 90 } // end class Record

Fig. 17.8 Record for sequential-access file-processing applications

code walkthrough

The Serializable attribute (line 6) indicates to the compiler that objects of class Record can be serialized, or represented as sets of bytes - we can read and write these bytes to our streams

Objects that we wish to write to or read from a stream must include this attribute in their class definitions

Class Record contains private data members account, firstName, lastName and balance (lines 9 -12), which collectively represent all information necessary to store record data

The default constructor (lines 15 - 17) sets these members to their default (i.e., empty) values, and the overloaded constructor (lines 20 - 28) sets these members to specified parameter values

Class Record also provides properties Account (lines 31 - 43), FirstName

(lines 46 - 58), LastName (lines 61 - 73) and Balance (lines 76 - 88) for accessing the account number, first name, last name and balance of each customer, respectively

Class CreateFileForm (Fig. [17.9](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.9','codesampleswin');)) uses instances of class Record to create a sequential-access file that might be used in an accounts receivable system - i.e., a program that organizes data regarding money owed by a company’s credit clients

For each client, the program obtains an account number and the client’s first name, last name and balance (i.e., the amount of money that the client owes to the company for previously received goods or services)

The data obtained for each client constitutes a record for that client

In this application, the account number represents the record key - files are created and maintained in account-number order

This program assumes that the user enters records in account-number order

However, a comprehensive accounts receivable system would provide a sorting capability

The user could enter the records in any order, and the records then could be sorted and written to the file in order

(Note that all outputs in this booklet should be read row by row, from left to right in each row)

Fig. 17.9 Create and write to a sequential-access file

1 // Fig 17.9: CreateSequentialAccessFile.cs  
 2 // Creating a sequential-access file.  
 3   
 4 // C# namespaces  
 5 using System;  
 6 using System.Drawing;  
 7 using System.Collections;  
 8 using System.ComponentModel;  
 9 using System.Windows.Forms;  
 10 using System.Data;  
 11 using System.IO;  
 12 using System.Runtime.Serialization.Formatters.Binary;  
 13 using System.Runtime.Serialization;  
 14   
 15 // Deitel namespace  
 16 using BankLibrary;  
 17   
 18 public class CreateFileForm : BankUIForm  
 19 {  
 20 private System.Windows.Forms.Button saveButton;  
 21 private System.Windows.Forms.Button enterButton;  
 22 private System.Windows.Forms.Button exitButton;  
 23   
 24 private System.ComponentModel.Container components = null;  
 25   
 26 // serializes Record in binary format  
 27 private BinaryFormatter formatter = new

BinaryFormatter();  
 28   
 29 // stream through which serializable data is written to file  
 30 private FileStream output;  
 31   
 32 [STAThread]  
 33 static void Main()   
 34 {  
 35 Application.Run( new CreateFileForm() );  
 36 }  
 37   
 38 // Visual Studio .NET generated code  
 39   
 40 // invoked when user clicks Save button  
 41 private void saveButton\_Click(  
 42 object sender, System.EventArgs e )  
 43 {  
 44 // create dialog box enabling user to save file  
 45 SaveFileDialog fileChooser = new SaveFileDialog();  
 46 DialogResult result = fileChooser.ShowDialog();  
 47 string fileName; // name of file to save data  
 48   
 49 // allow user to create file  
 50 fileChooser.CheckFileExists = false;  
 51   
 52 // exit event handler if user clicked "Cancel"  
 53 if ( result == DialogResult.Cancel )  
 54 return;  
 55   
 56 // get specified file name  
 57 fileName = fileChooser.FileName;  
 58   
 59 // show error if user specified invalid file  
 60 if ( fileName == "" || fileName == null )  
 61 MessageBox.Show( "Invalid File Name", "Error",  
 62 MessageBoxButtons.OK, MessageBoxIcon.Error );  
 63 else  
 64 {  
 65 // save file via FileStream if user specified valid file  
 66 try  
 67 {  
 68 // open file with write access  
 69 output = new FileStream( fileName,  
 70 FileMode.OpenOrCreate, FileAccess.Write );  
 71   
 72 // disable Save button and enable Enter button  
 73 saveButton.Enabled = false;  
 74 enterButton.Enabled = true;  
 75 }  
 76   
 77 // handle exception if file does not exist

78 catch ( FileNotFoundException )  
 79 {  
 80 // notify user if file does not exist  
 81 MessageBox.Show( "File Does Not Exist", "Error",  
 82 MessageBoxButtons.OK, MessageBoxIcon.Error );  
 83 }  
 84 }  
 85 } // end method saveButton\_Click  
 86   
 87 // invoke when user clicks Enter button  
 88 private void enterButton\_Click(  
 89 object sender, System.EventArgs e )  
 90 {  
 91 // store TextBox values string array  
 92 string[] values = GetTextBoxValues();  
 93   
 94 // Record containing TextBox values to serialize  
 95 Record record = new Record();  
 96   
 97 // determine whether TextBox account field is empty  
 98 if ( values[ ( int )TextBoxIndices.ACCOUNT ] != "" )  
 99 {  
 100 // store TextBox values in Record and serialize Record  
 101 try  
 102 {  
 103 // get account number value from TextBox  
 104 int accountNumber = Int32.Parse(   
 105 values[ ( int )TextBoxIndices.ACCOUNT ] );  
 106   
 107 // determine whether accountNumber is valid  
 108 if ( accountNumber > 0 )  
 109 {  
 110 // store TextBox fields in Record  
 111 record.Account = accountNumber;  
 112 record.FirstName =  
 113 values[ ( int )TextBoxIndices.FIRST ];  
 114 record.LastName =  
 115 values[ ( int )TextBoxIndices.LAST ];  
 116 record.Balance = Double.Parse( values[  
 117 ( int )TextBoxIndices.BALANCE ] );  
 118   
 119 // write Record to FileStream (serialize object)  
 120 formatter.Serialize( output, record );  
 121 }  
 122 else  
 123 {  
 124 // notify user if invalid account number  
 125 MessageBox.Show( "Invalid Account Number", "Error",

126 MessageBoxButtons.OK, MessageBoxIcon.Error );  
 127 }  
 128 }  
 129   
 130 // notify user if error occurs in serialization  
 131 catch( SerializationException )  
 132 {  
 133 MessageBox.Show( "Error Writing to File", "Error",  
 134 MessageBoxButtons.OK, MessageBoxIcon.Error );  
 135 }  
 136   
 137 // notify user if error occurs regarding parameter format  
 138 catch( FormatException )  
 139 {  
 140 MessageBox.Show( "Invalid Format", "Error",  
 141 MessageBoxButtons.OK, MessageBoxIcon.Error );  
 142 }  
 143 }  
 144   
 145 ClearTextBoxes(); // clear TextBox values  
 146   
 147 } // end method enterButton\_Click  
 148   
 149 // invoked when user clicks Exit button  
 150 private void exitButton\_Click(  
 151 object sender, System.EventArgs e )  
 152 {  
 153 // determine whether file exists  
 154 if ( output != null )  
 155 {  
 156 // close file  
 157 try  
 158 {  
 159 output.Close();  
 160 }  
 161   
 162 // notify user of error closing file  
 163 catch( IOException )  
 164 {  
 165 MessageBox.Show( "Cannot close file", "Error",  
 166 MessageBoxButtons.OK, MessageBoxIcon.Error );  
 167 }  
 168 }  
 169   
 170 Application.Exit();  
 171   
 172 } // end method exitButton\_Click  
 173   
 174 } // end class CreateFileForm

Fig. 17.9 Create and write to a sequential-access file

code walkthrough

Figure [17.9](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.9','codesampleswin');) contains the code for class CreateFileForm, which either creates or opens a file (depending on whether one exists), then allows the user to write bank information to that file

Line 16 imports the BankLibrary namespace; this namespace contains class BankUIForm, from which class CreateFileForm inherits (line 18)

Because of this inheritance relationship, the CreateFileForm GUI is similar to that of class BankUIForm (as shown previously), except that the inherited class also contains buttons Save As, Enter and Exit

When the user clicks the Save As button, the program invokes method saveButton\_Click (lines 41 - 85)

Line 45 instantiates an object of class SaveFileDialog, which belongs to the System.Windows.Forms namespace

Objects of this class are used for selecting files

Line 46 calls method ShowDialog of the SaveFileDialog object to display the SaveFileDialog

When displayed, a SaveFileDialog prevents the user from interacting with any other window in the program until the user closes the SaveFileDialog by clicking either Save or Cancel

Dialogs that behave in this fashion are called modal dialogs

The user selects the appropriate drive, directory and file name, then clicks Save

Method ShowDialog returns an integer specifying which button (Save or Cancel) the user clicked to close the dialog

In this example, the Form property DialogResult receives this integer

Line 53 tests whether the user clicked Cancel by comparing the value returned by property DialogResult to constant DialogResult.Cancel

If the values are equal, method saveButton\_Click returns (line 54)

If the values are unequal (i.e., the user clicked Save, instead of clicking Cancel), line 57 uses property FileName of class SaveFileDialog to obtain the user-selected file

As stated previously, we can open files to perform text manipulation by creating objects of classes FileStream

In this example, we want the file to be opened for output, so lines 69 - 70 instantiate a FileStream object

The FileStream constructor that we use receives three arguments - a string containing the name of the file to be opened, a constant describing how to open the file and a constant describing the file permissions

Line 70 passes constant FileMode.OpenOrCreate to the FileStream constructor as the constructor’s second argument

This constant indicates that the FileStream object should open the file if the file exists or create the file if the file does not exist

C# offers other FileMode constants describing how to open files; we introduce these constants as we use them in code examples

Line 70 passes constant FileAccess.Write to the FileStream constructor as the constructor’s third argument

This constant ensures that the program can perform write-only operations on the FileStream object

C# provides two other constants for this parameter - FileAccess.Read for read-only access and FileAccess.ReadWrite for both read and write access

After the user types information in each TextBox, the user clicks the Enter button, which calls method enterButton\_Click (lines 88 - 147) to save data from the TextBox in the user-specified file

If the user entered a valid account number (i.e., an integer greater than zero), lines 112 - 118 store the TextBox values in an object of type Record

If the user entered invalid data in one of the TextBoxes (such as entering non-numeric characters in the Balance field), the program throws a FormatException

The catch block in lines 138 - 142 handles such an exception by notifying the user (via a MessageBox) of the improper format

If the user entered valid data, line 120 writes the record to the file by invoking method Serialize of the BinaryFormatter object (instantiated in line 27)

Class BinaryFormatter uses methods Serialize and Deserialize to write and read objects into streams, respectively

Method Serialize writes the object’s representation to a file

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 (errors result when the methods attempt to access streams or records that do not exist)

Both methods Serialize and Deserialize require a Stream object (e.g., the FileStream) as a parameter so that the BinaryFormatter can access the correct file; the BinaryFormatter must receive an instance of a class that derives from class Stream, because Stream is abstract

Class BinaryFormatter belongs to the System.Runtime.Serialization.Formatters.Binary namespace

When the user clicks the Exit button, the program invokes method exitButton\_Click (lines 150 - 172) to exit the application

Line 159 closes the FileStream if one has been opened, and line 170 exits the program

In the sample execution for the program in Fig. 17.9, we entered information for five accounts, as shown below

The program does not depict how the data records are rendered in the file

To verify that the file has been created successfully, in the next section, we create a program to read and display the file

|  |  |  |  |
| --- | --- | --- | --- |
| **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 |

**Sample data for the program of Fig. 17.9**

Reading Data from a Sequential - Access File

Data are stored in files so that they can be retrieved for processing when they are needed

The previous section demonstrated how to create a file for use in sequential-access applications

In this section, we discuss how to read (or retrieve) data sequentially from a file

Class ReadSequentialAccessFileForm (see Fig. [17.11](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.11','codesampleswin');) next) reads records from the file created by the program in Fig. [17.9](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.9','codesampleswin');), then displays the contents of each record

Much of the code in this example is similar to that of Fig. [17.9](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.9','codesampleswin');), so we discuss only the unique aspects of the application

Fig. 17.11 Reading sequential-access files

1 // Fig. 17.11: ReadSequentialAccessFile.cs  
 2 // Reading a sequential-access file.  
 3   
 4 // C# namespaces  
 5 using System;  
 6 using System.Drawing;  
 7 using System.Collections;  
 8 using System.ComponentModel;  
 9 using System.Windows.Forms;  
 10 using System.Data;  
 11 using System.IO;  
 12 using System.Runtime.Serialization.Formatters.Binary;  
 13 using System.Runtime.Serialization;  
 14   
 15 // Deitel namespaces  
 16 using BankLibrary;  
 17   
 18 public class ReadSequentialAccessFileForm : BankUIForm  
 19 {  
 20 System.Windows.Forms.Button openButton;  
 21 System.Windows.Forms.Button nextButton;  
 22   
 23 private System.ComponentModel.Container components = null;  
 24   
 25 // stream through which serializable data are read from file  
 26 private FileStream input;  
 27   
 28 // object for deserializing Record in binary

format  
 29 private BinaryFormatter reader = new BinaryFormatter();  
 30   
 31 [STAThread]  
 32 static void Main()   
 33 {  
 34 Application.Run( new ReadSequentialAccessFileForm() );  
 35 }  
 36   
 37 // Visual Studio .NET generated code  
 38   
 39 // invoked when user clicks Open button  
 40 private void openButton\_Click(  
 41 object sender, System.EventArgs e )  
 42 {  
 43 // create dialog box enabling user to open file  
 44 OpenFileDialog fileChooser = new OpenFileDialog();  
 45 DialogResult result = fileChooser.ShowDialog();  
 46 string fileName; // name of file containing data  
 47   
 48 // exit event handler if user clicked Cancel  
 49 if ( result == DialogResult.Cancel )  
 50 return;  
 51   
 52 // get specified file name  
 53 fileName = fileChooser.FileName;   
 54 ClearTextBoxes();  
 55   
 56 // show error if user specified invalid file  
 57 if ( fileName == "" || fileName == null )  
 58 MessageBox.Show( "Invalid File Name", "Error",  
 59 MessageBoxButtons.OK, MessageBoxIcon.Error );  
 60 else  
 61 {  
 62 // create FileStream to obtain read access to file  
 63 input = new FileStream( fileName, FileMode.Open,  
 64 FileAccess.Read );  
 65   
 66 // enable next record button

67 nextButton.Enabled = true;  
 68 }  
 69   
 70 } // end method openButton\_Click  
 71   
 72 // invoked when user clicks Next button  
 73 private void nextButton\_Click(  
 74 object sender, System.EventArgs e )  
 75 {  
 76 // deserialize Record and store data in TextBoxes  
 77 try  
 78 {  
 79 // get next Record available in file  
 80 Record record =  
 81 ( Record )reader.Deserialize( input );  
 82   
 83 // store Record values in temporary string array  
 84 string[] values = new string[] {   
 85 record.Account.ToString(),  
 86 record.FirstName.ToString(),  
 87 record.LastName.ToString(),  
 88 record.Balance.ToString() };  
 89   
 90 // copy string array values to TextBox values  
 91 SetTextBoxValues( values );  
 92 }  
 93   
 94 // handle exception when no Records in file  
 95 catch( SerializationException )  
 96 {\  
 97 // close FileStream if no Records in file  
 98 input.Close();   
 99   
 100 // enable Open Record button  
 101 openButton.Enabled = true;   
 102   
 103 // disable Next Record button  
 104 nextButton.Enabled = false;   
 105   
 106 ClearTextBoxes();  
 107   
 108 // notify user if no Records in file  
 109 MessageBox.Show( "No more records in file", "",  
 110 MessageBoxButtons.OK, MessageBoxIcon.Information );

111 }  
 112   
 113 } // end method nextButton\_Click  
 114   
 115 } // end class ReadSequentialAccessFileForm

Fig. 17.11 Reading sequential-access files

code walkthrough

When the user clicks the Open File button, the program calls method openButton\_Click (lines 40 - 70)

Line 44 instantiates an object of class OpenFileDialog, and line 45 calls the object’s ShowDialog method to display the Open dialog

The behaviour and GUI for the two dialog types are the same (except that Save is replaced by Open)

If the user inputs a valid file name, lines 63 - 64 create a FileStream object and assign it to reference input

We pass constant FileMode.Open as the second argument to the FileStream constructor

This constant indicates that the FileStream should open the file if the file exists or should throw a FileNotFoundException if the file does not exist

(In this example, the FileStream constructor will not throw a FileNotFoundException, because the OpenFileDialog requires the user to enter a name of a file that exists)

Previously (Fig. [17.9](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.9','codesampleswin');)), we wrote text to the file using a FileStream object with write-only access

In this example, we specify read-only access to the file by passing constant FileAccess.Read as the third argument to the FileStream constructor

When the user clicks the Next Record button, the program calls method nextButton\_Click (lines 73 - 113), which reads the next record from the user-specified file

(The user must click Next Record after opening the file to view the first record)

Lines 80 - 81 call method Deserialize of the BinaryFormatter object to read the next record

Method Deserialize reads the data and casts the result to a Record - this cast is necessary because Deserialize returns a reference of type Object

Lines 84 - 91 then display the Record values in the TextBoxes

When method Deserialize attempts to deserialize a record that does not exist in the file (i.e., the program has displayed all file records), the method throws a SerializationException

The catch block (lines 95 - 111) that handles this exception closes the FileStream object (line 98) and notifies the user that there are no more records (lines 109 - 110)

To retrieve data sequentially from a file, programs normally start from the beginning of the file, reading data consecutively until the desired data are found

It sometimes is necessary to process a file sequentially several times (from the beginning of the file) during the execution of a program

A FileStream object can reposition its file-position pointer (which contains the byte number of the next byte to be read from or written to the file) to any position in the file - we show this feature when we introduce random-access file-processing applications later

When a FileStream object is opened, its file-position pointer is set to zero (i.e., the beginning of the file)

We now present a more substantial program that builds on the concepts employed in Fig. [17.11](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.11','codesampleswin');)

Class creditInquiryForm (Fig. [17.12](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.12','codesampleswin');)) is a credit-inquiry program that enables a credit manager to display account information for those customers with

credit balances

(i.e., customers to whom the company owes money),

zero balances

(i.e., customers who do not owe the company money)

and

debit balances

(i.e., customers who owe the company money for previously received goods and services)

Note that line 21 declares a RichTextBox that will display the account information

RichTextBoxes provide more functionality than do regular TextBoxes - for example, RichTextBoxes offer method Find for searching individual strings and method LoadFile for displaying file contents

Class RichTextBox does not inherit from class TextBox; rather, both classes inherit directly from abstract class System.Windows.Forms.TextBoxBase

We use a RichTextBox in this example, because a RichTextBox displays multiple lines of text by default, whereas a regular TextBox displays only one

Alternatively, we could have specified that a TextBox object display multiple lines of text by setting it’s Multiline property to true

The program in Fig. [17.12](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.12','codesampleswin');) displays buttons that enable a credit manager to obtain credit information

The Open File button opens a file for gathering data

The Credit Balances button displays a list of accounts that have credit balances, the Debit Balances button displays a list of accounts that have debit balances, and the Zero Balances button displays a list of accounts that have zero balances

The Done button exits the application

Fig. 17.12 Credit-inquiry program

1 // Fig. 17.12: CreditInquiry.cs  
 2 // Read a file sequentially and display contents based on  
 3 // account type specified by user (credit, debit or zero balances).  
 4   
 5 // C# namespaces  
 6 using System;  
 7 using System.Drawing;  
 8 using System.Collections;  
 9 using System.ComponentModel;  
 10 using System.Windows.Forms;  
 11 using System.Data;  
 12 using System.IO;  
 13 using System.Runtime.Serialization.Formatters.Binary;  
 14 using System.Runtime.Serialization;  
 15   
 16 // Deitel namespaces  
 17 using BankLibrary;  
 18   
 19 public class CreditInquiryForm : System.Windows.Forms.Form  
 20 {  
 21 private System.Windows.Forms.RichTextBox displayTextBox;  
 22   
 23 private System.Windows.Forms.Button doneButton;  
 24 private System.Windows.Forms.Button zeroButton;  
 25 private System.Windows.Forms.Button debitButton;  
 26 private System.Windows.Forms.Button creditButton;  
 27 private System.Windows.Forms.Button openButton;  
 28   
 29 private System.ComponentModel.Container components = null;  
 30   
 31 // stream through which serializable data are read from file  
 32 private FileStream input;  
 33   
 34 // object for deserializing Record in binary format  
 35 BinaryFormatter reader = new BinaryFormatter();  
 36   
 37 // name of file that stores credit, debit and zero balances  
 38 private string fileName;  
 39   
 40 [STAThread]  
 41 static void Main()   
 42 {  
 43 Application.Run( new CreditInquiryForm() );  
 44 }  
 45   
 46 // Visual Studio .NET generated code  
 47

48 // invoked when user clicks Open File button  
 49 private void openButton\_Click(  
 50 object sender, System.EventArgs e )  
 51 {  
 52 // create dialog box enabling user to open file  
 53 OpenFileDialog fileChooser = new OpenFileDialog();  
 54 DialogResult result = fileChooser.ShowDialog();  
 55   
 56 // exit event handler if user clicked Cancel  
 57 if ( result == DialogResult.Cancel )  
 58 return;  
 59   
 60 // get name from user  
 61 fileName = fileChooser.FileName;  
 62   
 63 // show error if user specified invalid file  
 64 if ( fileName == "" || fileName == null )  
 65 MessageBox.Show( "Invalid File Name", "Error",  
 66 MessageBoxButtons.OK, MessageBoxIcon.Error );  
 67 else  
 68 {  
 69 // enable all GUI buttons, except for Open file button  
 70 openButton.Enabled = false;  
 71 creditButton.Enabled = true;  
 72 debitButton.Enabled = true;  
 73 zeroButton.Enabled = true;  
 74 }  
 75   
 76 } // end method openButton\_Click  
 77   
 78 // invoked when user clicks credit balances,  
 79 // debit balances or zero balances button  
 80 private void get\_Click( object sender, System.EventArgs e )  
 81 {  
 82 // convert sender explicitly to object of type button  
 83 Button senderButton = ( Button )sender;  
 84   
 85 // get text from clicked Button, which stores account type  
 86 string accountType = senderButton.Text;  
 87   
 88 // read and display file information  
 89 try  
 90 {  
 91 // close file from previous operation  
 92 if ( input != null )  
 93 input.Close();  
 94   
 95 // create FileStream to obtain read access to file  
 96 input = new FileStream( fileName, FileMode.Open,

97 FileAccess.Read );  
 98   
 99 displayTextBox.Text = "The accounts are:\r\n";  
 100   
 101 // traverse file until end of file  
 102 while ( true )  
 103 {  
 104 // get next Record available in file  
 105 Record record = ( Record )reader.Deserialize( input );  
 106   
 107 // store record's last field in balance  
 108 Double balance = record.Balance;  
 109   
 110 // determine whether to display balance  
 111 if ( ShouldDisplay( balance, accountType ) )  
 112 {  
 113 // display record  
 114 string output = record.Account + "\t" +  
 115 record.FirstName + "\t" + record.LastName +  
 116 new string( ' ', 6 ) + "\t";  
 117   
 118 // display balance with correct monetary format  
 119 output += String.Format(   
 120 "{0:F}", balance ) + "\r\n";  
 121   
 122 // copy output to screen  
 123 displayTextBox.Text += output;   
 124 }  
 125 }  
 126 }  
 127   
 128 // handle exception when file cannot be closed  
 129 catch( IOException )  
 130 {  
 131 MessageBox.Show( "Cannot Close File", "Error",  
 132 MessageBoxButtons.OK, MessageBoxIcon.Error );  
 133 }  
 134   
 135 // handle exception when no more records  
 136 catch( SerializationException )  
 137 {  
 138 // close FileStream if no Records in file  
 139 input.Close();   
 140 }  
 141   
 142 } // end method get\_Click  
 143   
 144 // determine whether to display given record  
 145 private bool ShouldDisplay( double balance, string accountType )  
 146 {  
 147 if ( balance > 0 )

148 {  
 149 // display credit balances  
 150 if ( accountType == "Credit Balances" )  
 151 return true;  
 152 }  
 153   
 154 else if ( balance < 0 )  
 155 {  
 156 // display debit balances  
 157 if ( accountType == "Debit Balances" )  
 158 return true;  
 159 }  
 160   
 161 else // balance == 0  
 162 {  
 163 // display zero balances  
 164 if ( accountType == "Zero Balances" )  
 165 return true;  
 166 }  
 167   
 168 return false;  
 169   
 170 } // end method ShouldDisplay  
 171   
 172 // invoked when user clicks Done button  
 173 private void doneButton\_Click(  
 174 object sender, System.EventArgs e )  
 175 {  
 176 // determine whether file exists  
 177 if ( input != null )  
 178 {  
 179 // close file  
 180 try  
 181 {  
 182 input.Close();  
 183 }  
 184   
 185 // handle exception if FileStream does not exist  
 186 catch( IOException )  
 187 {  
 188 // notify user of error closing file  
 189 MessageBox.Show( "Cannot close file", "Error",  
 190 MessageBoxButtons.OK, MessageBoxIcon.Error);  
 191 }  
 192 }  
 193   
 194 Application.Exit();  
 195   
 196 } // end method doneButton\_Click  
 197   
 198 } // end class CreditInquiryForm

Fig. 17.12 Credit-inquiry program

code walkthrough

When the user clicks the Open File button, the program calls method openButton\_Click (lines 49 - 76)

Line 53 instantiates an object of class OpenFileDialog, and line 54 calls the object’s ShowDialog method to display the Open dialog, in which the user inputs the name of the file to open

When the user clicks Credit Balances, Debit Balances or Zero Balances, the program invokes method get\_Click (lines 80 - 142)

Line 83 casts the sender parameter, which is a reference to the object that sent the event, to a Button object

Line 86 extracts the Button object’s text, which the program uses to determine which GUI Button the user clicked

Lines 96 - 97 create a FileStream object with read-only file access and assign it to reference input

Lines 102 - 125 define a while loop that uses private method ShouldDisplay (lines 145 - 170) to determine whether to display each record in the file

The while loop obtains each record by calling method Deserialize of the FileStream object repeatedly (line 105)

When the file-position pointer reaches the end of file, method Deserialize throws a SerializationException, which the catch block in lines 136–140 handles

Line 139 calls the Close method of FileStream to close the file, and method get\_Click returns

Random-Access Files

So far, we have explained how to create sequential-access files and how to search through such files to locate particular information

However, sequential-access files are inappropriate for so-called “instant-access” applications, in which a particular record of information must be located immediately

Popular instant-access applications include airline-reservation systems, banking systems, point-of-sale systems, automated-teller machines and other kinds of transaction-processing systems requiring rapid access to specific data

The bank at which an individual has an account might have hundreds of thousands or even millions of other customers; however, when that individual uses an automated teller machine, the appropriate account is checked for sufficient funds in seconds

This type of instant access is made possible by random-access files

Individual records of a random-access file can be accessed directly (and quickly), without searching through potentially large numbers of other records, as is necessary with sequential-access files

Random-access files sometimes are called direct-access files

As we discussed earlier in this booklet, C# does not impose structure on files, so applications that use random-access files must implement the random-access capability

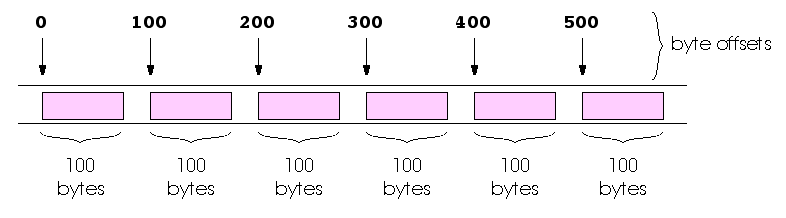
There are a variety of techniques for creating random-access files

Perhaps the simplest involves requiring that all records in a file be of a uniform, fixed length

The use of fixed-length records enables a program to calculate (as a function of the record size and the record key) the exact location of any record in relation to the beginning of the file

We soon demonstrate how this facilitates immediate access to specific records, even in large files

The diagram below illustrates the organization of a random-access file composed of fixed-length records (each record is 100 bytes long)

As an analogy, consider a random-access file as analogous to a railroad train with many cars, some of which are empty and some of which contain contents

**Fig. 17.13 Random-access file with fixed-length records**

Data can be inserted into a random-access file without destroying other data in the file

In addition, previously stored data can be updated or deleted without rewriting the entire file

In the following sections, we explain how to create a random-access file, write data to that file, read data both sequentially and randomly, update data and delete data that is no longer needed

Fig.[17.14](javascript:openPopUpWindow('../../ch17/codesamples/content.htm" \l "Fig17.14','codesampleswin');) contains class RandomAccessRecord, which is used in the random-access file-processing applications in this booklet

This class also belongs to the BankLibrary DLL

i.e., it is part of the project that contains classes BankUIForm and Record

(when adding class RandomAccessRecord to the project containing BankUIForm and Record, remember to rebuild the project)

Fig. 17.14 Record for random-access file-processing applications

1 // Fig. 17.14: RandomAccessRecord.cs  
 2 // Data-record class for random-access applications.  
 3   
 4 using System;  
 5   
 6 public class RandomAccessRecord  
 7 {  
 8 // length of firstName and lastName  
 9 private const int CHAR\_ARRAY\_LENGTH = 15;  
 10   
 11 private const int SIZE\_OF\_CHAR = 2;  
 12 private const int SIZE\_OF\_INT32 = 4;  
 13 private const int SIZE\_OF\_DOUBLE = 8;  
 14   
 15 // length of record  
 16 public const int SIZE = SIZE\_OF\_INT32 +  
 17 2 \* ( SIZE\_OF\_CHAR \* CHAR\_ARRAY\_LENGTH ) + SIZE\_OF\_DOUBLE;  
 18   
 19 // record data  
 20 private int account;  
 21 private char[] firstName = new char[ CHAR\_ARRAY\_LENGTH ];  
 22 private char[] lastName = new char[ CHAR\_ARRAY\_LENGTH ];  
 23 private double balance;  
 24   
 25 // default constructor sets members to default values  
 26 public RandomAccessRecord() : this( 0, "", "", 0.0 )  
 27 {  
 28 }  
 29   
 30 // overloaded counstructor sets members to

parameter values  
 31 public RandomAccessRecord( int accountValue,  
 32 string firstNameValue, string lastNameValue,   
 33 double balanceValue )  
 34 {  
 35 Account = accountValue;  
 36 FirstName = firstNameValue;  
 37 LastName = lastNameValue;  
 38 Balance = balanceValue;  
 39   
 40 } // end constructor  
 41   
 42 // property Account  
 43 public int Account  
 44 {  
 45 get  
 46 {  
 47 return account;  
 48 }  
 49   
 50 set  
 51 {  
 52 account = value;  
 53 }  
 54   
 55 } // end property Account  
 56   
 57 // property FirstName  
 58 public string FirstName  
 59 {  
 60 get  
 61 {  
 62 return new string( firstName );  
 63 }  
 64   
 65 set  
 66 {  
 67 // determine length of string parameter  
 68 int stringSize = value.Length;  
 69   
 70 // firstName string representation  
 71 string firstNameString = value;  
 72   
 73 // append spaces to string parameter if too short  
 74 if ( CHAR\_ARRAY\_LENGTH >= stringSize )  
 75 {  
 76 firstNameString = value +  
 77 new string( ' ', CHAR\_ARRAY\_LENGTH - stringSize );  
 78 }  
 79 else  
 80 {  
 81 // remove characters from string parameter if too long  
 82 firstNameString =

83 value.Substring( 0, CHAR\_ARRAY\_LENGTH );  
 84 }  
 85   
 86 // convert string parameter to char array  
 87 firstName = firstNameString.ToCharArray();  
 88   
 89 } // end set  
 90   
 91 } // end property FirstName  
 92   
 93 // property LastName  
 94 public string LastName  
 95 {  
 96 get  
 97 {  
 98 return new string( lastName );  
 99 }  
 100   
 101 set  
 102 {  
 103 // determine length of string parameter  
 104 int stringSize = value.Length;  
 105   
 106 // lastName string representation  
 107 string lastNameString = value;  
 108   
 109 // append spaces to string parameter if too short  
 110 if ( CHAR\_ARRAY\_LENGTH >= stringSize )  
 111 {  
 112 lastNameString = value +  
 113 new string( ' ', CHAR\_ARRAY\_LENGTH - stringSize );  
 114 }  
 115 else  
 116 {  
 117 // remove characters from string parameter if too long  
 118 lastNameString =   
 119 value.Substring( 0, CHAR\_ARRAY\_LENGTH );  
 120 }  
 121   
 122 // convert string parameter to char array  
 123 lastName = lastNameString.ToCharArray();  
 124   
 125 } // end set  
 126   
 127 } // end property LastName  
 128   
 129 // property Balance  
 130 public double Balance  
 131 {  
 132 get  
 133 {

134 return balance;  
 135 }  
 136   
 137 set  
 138 {  
 139 balance = value;  
 140 }  
 141   
 142 } // end property Balance  
 143   
 144 } // end class RandomAccessRecord

Fig. 17.14 Record for random-access file-processing applications

code walkthrough

Like class Record (refer back to Fig. [17.8](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.8','codesampleswin');)), class RandomAccessRecord contains private data members (lines 20 - 23) for storing record information, two constructors for setting these members to default and parameter-specified values, respectively, and properties for accessing these members

However, class RandomAccessRecord does not contain attribute [Serializable] before its class definition

We do not serialize this class, because C# does not provide a means to obtain an object’s size at runtime

This means that, if we serialize the class, we cannot guarantee a fixed-length record size

Instead of serializing the class, we fix the length of the private data members, then write those data as a byte stream to the file

To fix this length, the set accessors of properties FirstName (lines 58 - 91) and LastName (lines 94 - 127) ensure that members firstName and lastName are char arrays of exactly 15 elements

Each set accessor receives as an argument a string representing the first name and last name, respectively

If the string parameter contains fewer than 15 characters, the property’s set accessor copies the string’s values to the char array, then populates the remainder with spaces

If the string parameter contains more than 15 characters, the set accessor stores only the first 15 characters of the string parameter into the char array

Lines 16 - 17 declare const SIZE, which specifies the record’s length

Each record contains account (4-byte int), firstName and lastName (two 15-element char arrays, where each char occupies two bytes, resulting in a total of 60 bytes) and balance (8-byte double)

In this example, each record (i.e., the four private data members that our programs will read to and write from files) occupies 72 bytes (4 bytes + 60 bytes + 8 bytes)

Creating a Random-Access File

Consider the following problem statement for a credit-processing application

**“Create a transaction-processing program capable of storing a maximum of 100 fixed-length records for a company that can have a maximum of 100 customers**

**Each record consists of an account number (which acts as the record key), a last name, a first name and a balance**

**The program can update an account, create an account and delete an account”**

The next several sections introduce the techniques necessary to create this credit-processing program

We will now discuss the program used to create the random-access file that the programs of Fig. [17.16](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.16','codesampleswin');) and Fig. [17.17](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.17','codesampleswin');) and the transaction-processing application will use to manipulate data

Class CreateRandomAccessFile (Fig. [17.15](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.15','codesampleswin');)) creates a random-access file

Fig. 17.15 Creating files for random-access file-processing applications

1 // Fig. 17.15: CreateRandomAccessFile.cs  
 2 // Creating a random file.  
 3   
 4 // C# namespaces  
 5 using System;  
 6 using System.IO;  
 7 using System.Windows.Forms;  
 8   
 9 // Deitel namespaces  
 10 using BankLibrary;  
 11   
 12 class CreateRandomAccessFile  
 13 {  
 14 // number of records to write to disk  
 15 private const int NUMBER\_OF\_RECORDS = 100;  
 16   
 17 [STAThread]  
 18 static void Main(string[] args)  
 19 {  
 20 // create random file, then save to disk  
 21 CreateRandomAccessFile file = new CreateRandomAccessFile();  
 22 file.SaveFile();  
 23   
 24 } // end method Main  
 25   
 26 // write records to disk  
 27 private void SaveFile()  
 28 {  
 29 // record for writing to disk  
 30 RandomAccessRecord blankRecord = new RandomAccessRecord();  
 31   
 32 // stream through which serializable data are

written to file  
 33 FileStream fileOutput = null;  
 34   
 35 // stream for writing bytes to file  
 36 BinaryWriter binaryOutput = null;  
 37   
 38 // create dialog box enabling user to save file  
 39 SaveFileDialog fileChooser = new SaveFileDialog();  
 40 DialogResult result = fileChooser.ShowDialog();  
 41   
 42 // get file name from user  
 43 string fileName = fileChooser.FileName;  
 44   
 45 // exit event handler if user clicked Cancel  
 46 if ( result == DialogResult.Cancel )  
 47 return;  
 48   
 49 // show error if user specified invalid file  
 50 if ( fileName == "" || fileName == null )  
 51 MessageBox.Show("Invalid File Name", "Error",   
 52 MessageBoxButtons.OK, MessageBoxIcon.Error);  
 53 else  
 54 {  
 55 // write records to file  
 56 try  
 57 {  
 58 // create FileStream to hold records  
 59 fileOutput = new FileStream( fileName,  
 60 FileMode.Create, FileAccess.Write );  
 61   
 62 // set length of file  
 63 fileOutput.SetLength( RandomAccessRecord.SIZE \*  
 64 NUMBER\_OF\_RECORDS );  
 65   
 66 // create object for writing bytes to file  
 67 binaryOutput = new BinaryWriter( fileOutput );  
 68   
 69 // write empty records to file  
 70 for ( int i = 0; i < NUMBER\_OF\_RECORDS; i++ )  
 71 {  
 72 // set file position pointer in file  
 73 fileOutput.Position = i \* RandomAccessRecord.SIZE;  
 74   
 75 // write blank record to file  
 76 binaryOutput.Write( blankRecord.Account );

77 binaryOutput.Write( blankRecord.FirstName );  
 78 binaryOutput.Write( blankRecord.LastName );  
 79 binaryOutput.Write( blankRecord.Balance );  
 80 }  
 81   
 82 // notify user of success  
 83 MessageBox.Show("File Created", "Success",  
 84 MessageBoxButtons.OK, MessageBoxIcon.Information);  
 85 }  
 86   
 87 // handle exception if error occurs during writing  
 88 catch( IOException )  
 89 {  
 90 // notify user of error  
 91 MessageBox.Show( "Cannot write to file", "Error",   
 92 MessageBoxButtons.OK, MessageBoxIcon.Error );  
 93 }  
 94 }  
 95   
 96 // close FileStream  
 97 if ( fileOutput == null )  
 98 fileOutput.Close();  
 99   
 100 // close BinaryWriter  
 101 if ( binaryOutput == null )  
 102 binaryOutput.Close();  
 103   
 104 } // end method SaveFile  
 105 } // end class CreateRandomAccessFile

Fig. 17.15 Creating files for random-access file-processing applications

code walkthrough

Method Main (lines 18 - 24) starts the application, which creates a random-access file by calling user-defined method SaveFile (lines 27 - 104)

Method SaveFile populates a file with 100 copies of the default (i.e., empty) values for private data members account, firstName, lastName and balance of class RandomAccessRecord

Lines 39 - 40 create and display the SaveFileDialog, which enables a user to specify the file to which the program writes data

Using this file, lines 59 - 60 instantiate the FileStream

Note that line 60 passes constant FileMode.Create, which either creates the specified file, if the file does not exist, or overwrites the specified file if it already exists

Lines 63 - 64 sets the FileStream’s length, which is equal to the size of an individual RandomAccessRecord (obtained through constant RandomAccessRecord.SIZE) multiplied by the number of records we want to copy (obtained through constant NUMBER\_OF\_RECORDS in line 15, which we set to value 100)

We now require a means to write bytes to a file

Class BinaryWriter of namespace System.IO provides methods for writing bytes to streams

The BinaryWriter constructor takes as an argument a reference to an instance of class System.IO.Stream, through which the BinaryWriter can write bytes

Class FileStream provides methods for writing streams to files and inherits from class Stream, so we can pass the FileStream object as an argument to the BinaryWriter constructor (line 67)

Now, we can use the BinaryWriter to write bytes directly to the file

Lines 70 - 80 populate the file with 100 copies of the empty record values

(i.e., default values for private data members of class RandomAccessRecord)

Line 73 changes the file-position pointer to specify the location in the file at which to write the next empty record

Now that we are working with a random-access file, we must set the file-pointer explicitly, using the FileStream object’s Position property

This property receives as an argument a long value describing where to position the pointer relative to the beginning of the file - in this example, we set the pointer so that it advances a number of bytes that is equal to the record size (obtained by RandomAccessRecord.SIZE)

Lines 76 - 79 call method Write of the BinaryWriter object to write the data

Method Write is an overloaded method that receives as an argument any primitive data type, then writes that type to a stream of bytes

After the for loop exits, lines 97 - 102 close the FileStream and BinaryWriter objects

Writing Data Randomly to a Random-Access File

Now that we have created a random-access file, we will now create and use class WriteRandomAccessFileForm to write data to that file

Fig. 17.16 Writing records to random-access files

1 // Fig 17.16: WriteRandomAccessFile.cs  
 2 // Write data to a random-access file.  
 3   
 4 // C# namespaces  
 5 using System;  
 6 using System.Drawing;  
 7 using System.Collections;  
 8 using System.ComponentModel;  
 9 using System.Windows.Forms;  
 10 using System.Data;  
 11 using System.IO;  
 12   
 13 // Deitel namespaces  
 14 using BankLibrary;  
 15   
 16 public class WriteRandomAccessFileForm : BankUIForm  
 17 {  
 18 private System.Windows.Forms.Button openButton;  
 19 private System.Windows.Forms.Button enterButton;  
 20   
 21 private System.ComponentModel.Container components = null;  
 22   
 23 // number of RandomAccessRecords to write to disk  
 24 private const int NUMBER\_OF\_RECORDS = 100;  
 25   
 26 // stream through which data are written to file  
 27 private FileStream fileOutput;  
 28   
 29 // stream for writing bytes to file  
 30 private BinaryWriter binaryOutput;  
 31   
 32 [STAThread]  
 33 static void Main()   
 34 {  
 35 Application.Run( new WriteRandomAccessFileForm() );  
 36 }  
 37   
 38 // Visual Studio .NET generated code  
 39   
 40 // invoked when user clicks Open button  
 41 private void openButton\_Click(  
 42 object sender, System.EventArgs e )  
 43 {  
 44 // create dialog box enabling user to open file  
 45 OpenFileDialog fileChooser = new OpenFileDialog();  
 46 DialogResult result = fileChooser.ShowDialog();

47   
 48 // get file name from user  
 49 string fileName = fileChooser.FileName;  
 50   
 51 // exit event handler if user clicked Cancel  
 52 if ( result == DialogResult.Cancel )  
 53 return;  
 54   
 55 // show error if user specified invalid file  
 56 if ( fileName == "" || fileName == null )  
 57 MessageBox.Show("Invalid File Name", "Error",  
 58 MessageBoxButtons.OK, MessageBoxIcon.Error);  
 59 else  
 60 {  
 61 // open file if file already exists  
 62 try  
 63 {  
 64 // create FileStream to hold records  
 65 fileOutput = new FileStream( fileName,   
 66 FileMode.Open, FileAccess.Write );  
 67   
 68 // create object for writing bytes to file  
 69 binaryOutput = new BinaryWriter( fileOutput );  
 70   
 71 // disable Open button and enable Enter button  
 72 openButton.Enabled = false;   
 73 enterButton.Enabled = true;  
 74 }  
 75   
 76 // notify user if file does not exist  
 77 catch( IOException )  
 78 {  
 79 MessageBox.Show("File Does Not Exits", "Error",  
 80 MessageBoxButtons.OK, MessageBoxIcon.Error);  
 81 }  
 82 }  
 83   
 84 } // end method openButton\_Click  
 85   
 86 // invoked when user clicks Enter button  
 87 private void enterButton\_Click(  
 88 object sender, System.EventArgs e )  
 89 {  
 90 // TextBox values string array  
 91 string[] values = GetTextBoxValues();  
 92   
 93 // determine whether TextBox account field is empty  
 94 if ( values[ ( int )TextBoxIndices.ACCOUNT ] != "" )  
 95 {  
 96 // write record to file at appropriate position

97 try  
 98 {  
 99 // get account number value from TextBox  
 100 int accountNumber = Int32.Parse(   
 101 values[ ( int )TextBoxIndices.ACCOUNT ] );  
 102   
 103 // determine whether accountNumber is valid  
 104 if ( accountNumber > 0 &&   
 105 accountNumber <= NUMBER\_OF\_RECORDS )  
 106 {  
 107 // move file position pointer  
 108 fileOutput.Seek( ( accountNumber - 1 ) \*  
 109 RandomAccessRecord.SIZE, SeekOrigin.Begin );  
 110   
 111 // write data to file  
 112 binaryOutput.Write( accountNumber );  
 113 binaryOutput.Write(  
 114 values[ ( int )TextBoxIndices.FIRST ] );  
 115 binaryOutput.Write(  
 116 values[ ( int )TextBoxIndices.LAST ] );  
 117 binaryOutput.Write( Double.Parse( values[  
 118 ( int )TextBoxIndices.BALANCE ] ) );  
 119 }  
 120 else  
 121 {  
 122 // notify user if invalid account number  
 123 MessageBox.Show("Invalid Account Number", "Error",   
 124 MessageBoxButtons.OK, MessageBoxIcon.Error);  
 125 }  
 126 }  
 127   
 128 // handle number-format exception  
 129 catch( FormatException )  
 130 {  
 131 // notify user if error occurs when formatting numbers  
 132 MessageBox.Show("Invalid Balance", "Error",  
 133 MessageBoxButtons.OK, MessageBoxIcon.Error );  
 134 }  
 135 }  
 136   
 137 ClearTextBoxes(); // clear text box values  
 138   
 139 } // end method enterButton\_Click  
 140   
 141 } // end class WriteRandomAccessFileForm

Fig. 17.16 Writing records to random-access files

code walkthrough

When a user clicks the Open File button, the program invokes method openButton\_Click (lines 41 - 84), which displays the OpenFileDialog for specifying the file in which to serialize data (lines 45 - 46); the program then uses the specified file to create a FileStream object with write-only access (lines 65 - 66)

Line 69 uses the FileStream reference to instantiate an object of class BinaryWriter, enabling the program to write bytes to files

Remember, we used the same approach when working with class CreateRandomAccessFile in Fig. [17.15](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.15','codesampleswin');)

The user enters values in the TextBoxes for the account number, first name, last name and balance

When the user clicks the Enter button, the program invokes method enterButton\_Click (lines 87 - 139), which writes the data in the TextBoxes to the file

Line 91 calls method GetTextBoxValues (provided by base class BankUIForm) to retrieve the data

Lines 104 - 105 determine whether the Account Number TextBox holds valid information (i.e., whether the account number is in the 1 - 100 range)

Class WriteRandomAccessFileForm must determine the location in the FileStream at which to insert the data from the TextBoxes

Lines 108 - 109 use method Seek of the FileStream object to locate an exact point in the file

In this case, method Seek sets the position of the file-position pointer for the FileStream object to the byte location as calculated by:

(accountNumber -1) \* RandomAccessRecord.SIZE

Because the account numbers range from 1 to 100, we subtract 1 from the account number when calculating the byte location of the record

For example, our use of method Seek sets the first record’s file-position pointer to byte 0 of the file (the file’s beginning)

The second argument to method Seek is a member of the enumeration SeekOrigin and specifies the location at which the method should begin seeking

We use const SeekOrigin.Begin, because we want the method to seek in relation to the beginning of the file

After the program determines the file location at which to place the record,

lines 112 - 118 write the record to the file using the BinaryWriter (as discussed in a previous section)

Reading Data Sequentially from a Random-Access File

In the previous sections, we created a random-access file and wrote data to that file

Here, in Fig. [17.17](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.17','codesampleswin');), we develop a program that opens the file, reads records from it and displays only the records that contain data (i.e., those records in which the account number is not zero)

This program also provides an additional benefit

Have a think about what this benefit is

- we will reveal it at the end of this section ……

Fig. 17.17 Reading records from random-access files sequentially

1 // Fig 17.17: ReadRandomAccessFile.cs  
 2 // Reads and displays random-access file contents.  
 3   
 4 // C# namespaces  
 5 using System;  
 6 using System.Drawing;  
 7 using System.Collections;  
 8 using System.ComponentModel;  
 9 using System.Windows.Forms;  
 10 using System.Data;  
 11 using System.IO;  
 12   
 13 // Deitel namespaces  
 14 using BankLibrary;  
 15   
 16 public class ReadRandomAccessFileForm : BankUIForm  
 17 {  
 18 private System.Windows.Forms.Button openButton;  
 19 private System.Windows.Forms.Button nextButton;  
 20   
 21 private System.ComponentModel.Container components = null;  
 22   
 23 // stream through which data are read from file  
 24 private FileStream fileInput;  
 25   
 26 // stream for reading bytes from file  
 27 private BinaryReader binaryInput;  
 28   
 29 // index of current record to be displayed  
 30 private int currentRecordIndex;  
 31   
 32 [STAThread]  
 33 static void Main()   
 34 {  
 35 Application.Run( new ReadRandomAccessFileForm() );  
 36 }  
 37   
 38 // Visual Studio .NET generated code  
 39

40 // invoked when user clicks Open button  
 41 private void openButton\_Click(  
 42 object sender, System.EventArgs e )  
 43 {  
 44 // create dialog box enabling user to open file  
 45 OpenFileDialog fileChooser = new OpenFileDialog();  
 46 DialogResult result = fileChooser.ShowDialog();  
 47   
 48 // get file name from user  
 49 string fileName = fileChooser.FileName;  
 50   
 51 // exit eventhandler if user clicked Cancel  
 52 if ( result == DialogResult.Cancel )  
 53 return;  
 54   
 55 // show error if user specified invalid file  
 56 if ( fileName == "" || fileName == null )  
 57 MessageBox.Show( "Invalid File Name", "Error",  
 58 MessageBoxButtons.OK, MessageBoxIcon.Error );  
 59 else  
 60 {  
 61 // create FileStream to obtain read access to file  
 62 fileInput = new FileStream( fileName,   
 63 FileMode.Open, FileAccess.Read );  
 64   
 65 // use FileStream for BinaryWriter to read bytes from file  
 66 binaryInput = new BinaryReader( fileInput );  
 67   
 68 openButton.Enabled = false; // disable Open button  
 69 nextButton.Enabled = true; // enable Next button  
 70   
 71 currentRecordIndex = 0;  
 72 ClearTextBoxes();  
 73 }  
 74   
 75 } // end method openButton\_Click  
 76   
 77 // invoked when user clicks Next button  
 78 private void nextButton\_Click(  
 79 object sender, System.EventArgs e )  
 80 {  
 81 // record to store file data  
 82 RandomAccessRecord record = new RandomAccessRecord();  
 83   
 84 // read record and store data in TextBoxes

85 try  
 86 {  
 87 string[] values; // for storing TextBox values  
 88   
 89 // get next record available in file  
 90 while( record.Account == 0 )  
 91 {  
 92 // set file position pointer to next record in file  
 93 fileInput.Seek(   
 94 currentRecordIndex \* RandomAccessRecord.SIZE, 0 );  
 95   
 96 currentRecordIndex += 1;  
 97   
 98 // read data from record  
 99 record.Account = binaryInput.ReadInt32();  
 100 record.FirstName = binaryInput.ReadString();  
 101 record.LastName = binaryInput.ReadString();  
 102 record.Balance = binaryInput.ReadDouble();  
 103 }  
 104   
 105 // store record values in temporary string array  
 106 values = new string[] {  
 107 record.Account.ToString(),  
 108 record.FirstName,  
 109 record.LastName,  
 110 record.Balance.ToString() };  
 111   
 112 // copy string array values to TextBox values  
 113 SetTextBoxValues( values );  
 114 }  
 115   
 116 // handle exception when no records in file  
 117 catch( IOException )  
 118 {  
 119 // close streams if no records in file  
 120 fileInput.Close();  
 121 binaryInput.Close();  
 122   
 123 openButton.Enabled = true; // enable Open button  
 124 nextButton.Enabled = false; // disable Next button  
 125 ClearTextBoxes();  
 126   
 127 // notify user if no records in file  
 128 MessageBox.Show("No more records in file", "",

129 MessageBoxButtons.OK, MessageBoxIcon.Information);  
 130 }  
 131   
 132 } // end method nextButton\_Click  
 133   
 134 } // end class ReadRandomAccessFileForm

Fig. 17.17 Reading records from random-access files sequentially

code walkthrough

When the user clicks the Open File button, class ReadRandomAccessFileForm invokes method openButton\_Click (lines 41 - 75), which displays the OpenFileDialog for specifying the file from which to read data

Lines 62 - 63 instantiate a FileStream object that opens a file with read-only access

Line 66 creates an instance of class BinaryReader, which reads bytes from a stream

We pass the FileStream object as an argument to the BinaryReader constructor, thus enabling the BinaryReader to read bytes from the file

When the user clicks the Next button, the program calls method nextButton\_Click (lines 78 - 132), which reads the next record from the file

Line 82 instantiates a RandomAccessRecord for storing the record data from the file

Lines 90 - 114 provide a while loop that reads from the file until it reaches a record that has a non-zero account number (0 is the initial value for the account number)

Lines 93 - 94 call method Seek of the FileStream object, which moves the file-position pointer to the appropriate place in the file where the record must be read

To accomplish this, method Seek uses int currentRecordIndex, which stores the number of records that have been read

Lines 99 - 102 use the BinaryReader object to store the file data in the RandomAccessRecord object

Recall that class BinaryWriter provides overloaded Write methods for writing data

However, class BinaryReader does not provide overloaded Read methods to read data

This means that we must use method ReadInt32 to read an int, method ReadString to read a string and method ReadDouble to read a double

Note that the order of these method invocations must correspond to the order in which the BinaryWriter object wrote each data type

When the BinaryReader reads a valid account number (i.e., a non-zero value), the loop terminates, and lines 106 - 113 display the record values in the TextBoxes

When the program has displayed all records, method Seek throws an IOException (because method Seek tries to position the file-position pointer to a location that is beyond the end-of-file marker)

The catch block (lines 117 - 130) handles this exception by closing the FileStream and BinaryReader objects (lines 120 - 121) and notifying the user that no more records exist (lines 128 - 129)

And what about that additional benefit we promised ?

If you examine the GUI as the program executes, they will notice that the program displays the records in ascending order by account number !

This is a simple consequence of using our direct-access techniques to store these records in the file

Sorting with direct-access techniques is much faster than sorting with the bubble sort as presented in the “Arrays” booklet

We achieve this improved speed by making the file large enough to hold every possible record that a user might create

Of course, this means that the file could be sparsely occupied most of the time, resulting in a waste of storage

Here is yet another example of the space/time trade-off -

by using large amounts of space, we are able to develop a faster sorting algorithm

Case Study: A Transaction-Processing Program

We now develop a substantial transaction-processing program

(Figs. 17.18 through to 17.23)

using a random-access file to achieve “instant - access” processing

The program maintains a bank’s account information

Users of this program can add new accounts, update existing accounts and delete accounts that are no longer needed

First, we discuss the transaction-processing behaviour (i.e., the class that enables the addition, updating and removal of accounts)

We then discuss the GUI, which contains windows that display the account information and enable the user to invoke the application’s transaction-processing behaviour

Fig. 17.18 Record-transaction class for the transaction-processor case study

1 // Fig. 17.18: Transaction.cs  
 2 // Handles record transactions.  
 3   
 4 // C# namespaces  
 5 using System;  
 6 using System.IO;  
 7 using System.Windows.Forms;  
 8   
 9 // Deitel namespaces  
 10 using BankLibrary;  
 11   
 12 public class Transaction  
 13 {  
 14 // number of records to write to disk  
 15 private const int NUMBER\_OF\_RECORDS = 100;  
 16   
 17 // stream through which data move to and from file  
 18 private FileStream file;  
 19   
 20 // stream for reading bytes from file  
 21 private BinaryReader binaryInput;  
 22   
 23 // stream for writing bytes to file  
 24 private BinaryWriter binaryOutput;  
 25   
 26 // create/open file containing empty records  
 27 public void OpenFile( string fileName )  
 28 {  
 29 // write empty records to file  
 30 try  
 31 {  
 32 // create FileStream from new file or existing file  
 33 file = new FileStream( fileName, FileMode.OpenOrCreate );  
 34   
 35 // use FileStream for BinaryWriter to read bytes from file  
 36 binaryInput = new BinaryReader( file );  
 37   
 38 // use FileStream for BinaryWriter to write bytes to file  
 39 binaryOutput = new BinaryWriter( file );  
 40   
 41 // determine whether file has just been created  
 42 if ( file.Length == 0 )  
 43 {  
 44 // record to be written to file  
 45 RandomAccessRecord blankRecord =   
 46 new RandomAccessRecord();  
 47   
 48 // new record can hold NUMBER\_OF\_RECORDS

records  
 49 file.SetLength( RandomAccessRecord.SIZE \*  
 50 NUMBER\_OF\_RECORDS );  
 51   
 52 // write blank records to file  
 53 for ( int i = 0; i < NUMBER\_OF\_RECORDS; i++ )  
 54 {  
 55 // move file-position pointer to next position  
 56 file.Position = i \* RandomAccessRecord.SIZE;  
 57   
 58 // write blank record to file  
 59 binaryOutput.Write( blankRecord.Account );  
 60 binaryOutput.Write( blankRecord.FirstName );  
 61 binaryOutput.Write( blankRecord.LastName );  
 62 binaryOutput.Write( blankRecord.Balance );  
 63 }  
 64 }  
 65 }  
 66   
 67 // notify user of error during writing of blank records  
 68 catch( IOException )  
 69 {   
 70 MessageBox.Show("Cannot create file", "Error",  
 71 MessageBoxButtons.OK, MessageBoxIcon.Error);  
 72 }  
 73   
 74 } // end method OpenFile  
 75   
 76 // retrieve record depending on whether account is valid  
 77 public RandomAccessRecord GetRecord( string accountValue )  
 78 {  
 79 // store file data associated with account in record  
 80 try  
 81 {  
 82 // record to store file data  
 83 RandomAccessRecord record = new RandomAccessRecord();  
 84   
 85 // get value from TextBox's account field  
 86 int accountNumber = Int32.Parse( accountValue );  
 87

88 // if account is invalid, do not read data  
 89 if ( accountNumber < 1 ||   
 90 accountNumber > NUMBER\_OF\_RECORDS )  
 91 {  
 92 // set record's account field with account number  
 93 record.Account = accountNumber;  
 94 }  
 95   
 96 // get data from file if account is valid  
 97 else  
 98 {  
 99 // locate position in file where record exists  
 100 file.Seek( ( accountNumber - 1 ) \*  
 101 RandomAccessRecord.SIZE, 0 );  
 102   
 103 // read data from record  
 104 record.Account = binaryInput.ReadInt32();  
 105 record.FirstName = binaryInput.ReadString();  
 106 record.LastName = binaryInput.ReadString();  
 107 record.Balance = binaryInput.ReadDouble();  
 108 }  
 109   
 110 return record;  
 111 }  
 112   
 113 // notify user of error during reading  
 114 catch( IOException )  
 115 {  
 116 MessageBox.Show( "Cannot read file", "Error",  
 117 MessageBoxButtons.OK, MessageBoxIcon.Error );  
 118 }  
 119   
 120 return null;  
 121   
 122 } // end method GetRecord;  
 123   
 124 // add record to file at position determined by accountNumber  
 125 public bool AddRecord(   
 126 RandomAccessRecord record, int accountNumber )  
 127 {  
 128 // write record to file  
 129 try  
 130 {  
 131 // move file position pointer to appropriate position  
 132 file.Seek( ( accountNumber - 1 ) \*   
 133 RandomAccessRecord.SIZE, 0 );

134   
 135 // write data to file  
 136 binaryOutput.Write(record.Account);  
 137 binaryOutput.Write(record.FirstName);  
 138 binaryOutput.Write(record.LastName);  
 139 binaryOutput.Write(record.Balance);  
 140 }  
 141   
 142 // notify user if error occurs during writing  
 143 catch( IOException )  
 144 {  
 145 MessageBox.Show( "Error Writing To File", "Error",  
 146 MessageBoxButtons.OK, MessageBoxIcon.Error );  
 147   
 148 return false; // failure  
 149 }  
 150   
 151 return true; // success  
 152   
 153 } // end method AddRecord  
 154   
 155 } // end class Transaction

Fig. 17.19 TransactionProcessorForm class runs the transaction-processor application

1 // Fig. 17.19: TransactionProcessor.cs  
 2 // MDI parent for transaction-processor application.  
 3   
 4 using System;  
 5 using System.Drawing;  
 6 using System.Collections;  
 7 using System.ComponentModel;  
 8 using System.Windows.Forms;  
 9 using System.Data;  
 10   
 11 public class TransactionProcessorForm   
 12 : System.Windows.Forms.Form  
 13 {  
 14 private System.ComponentModel.Container components = null;  
 15 private System.Windows.Forms.MdiClient MdiClient1;  
 16   
 17 // reference to StartDialog  
 18 private StartDialogForm startDialog;  
 19   
 20 // constructor  
 21 public TransactionProcessorForm()  
 22 {  
 23 // required for Windows Form Designer support  
 24 InitializeComponent();  
 25   
 26 startDialog = new StartDialogForm();  
 27 startDialog.MdiParent = this;  
 28 startDialog.Show();  
 29 }  
 30   
 31 [STAThread]  
 32 static void Main()   
 33 {  
 34 Application.Run( new TransactionProcessorForm() );  
 35 }  
 36   
 37 // Visual Studio .NET generated code  
 38   
 39 } // end class TransactionProcessorForm

Fig. 17.20 StartDialogForm class enables users to access dialog boxes associated with various transactions

1 // Fig. 17.20: StartDialog.cs  
 2 // Initial dialog box displayed to user. Provides buttons for   
 3 // creating/opening file and for adding, updating and removing  
 4 // records from file.  
 5   
 6 // C# namespaces  
 7 using System;  
 8 using System.Drawing;  
 9 using System.Collections;  
 10 using System.ComponentModel;  
 11 using System.Windows.Forms;  
 12   
 13 // Deitel namespaces  
 14 using BankLibrary;  
 15   
 16 public delegate void MyDelegate();  
 17   
 18 public class StartDialogForm : System.Windows.Forms.Form  
 19 {  
 20 private System.Windows.Forms.Button updateButton;  
 21 private System.Windows.Forms.Button newButton;  
 22 private System.Windows.Forms.Button deleteButton;  
 23 private System.Windows.Forms.Button openButton;  
 24   
 25 private System.ComponentModel.Container components = null;  
 26   
 27 // reference to dialog box for adding record  
 28 private NewDialogForm newDialog;  
 29   
 30 // reference to dialog box for updating record  
 31 private UpdateDialogForm updateDialog;  
 32   
 33 // reference to dialog box for removing record  
 34 private DeleteDialogForm deleteDialog;  
 35   
 36 // reference to object that handles transactions  
 37 private Transaction transactionProxy;  
 38   
 39 // Visual Studio .NET generated code  
 40   
 41 // invoked when user clicks New/Open File button  
 42 private void openButton\_Click(  
 43 object sender, System.EventArgs e )  
 44 {  
 45 // create dialog box enabling user to create or open file  
 46 OpenFileDialog fileChooser = new OpenFileDialog();  
 47 DialogResult result;

48 string fileName;  
 49   
 50 // enable user to create file if file does not exist  
 51 fileChooser.Title = "Create File / Open File";  
 52 fileChooser.CheckFileExists = false;  
 53   
 54 // show dialog box to user  
 55 result = fileChooser.ShowDialog();  
 56   
 57 // exit event handler if user clicked Cancel  
 58 if ( result == DialogResult.Cancel )  
 59 return;  
 60   
 61 // get file name from user  
 62 fileName = fileChooser.FileName;  
 63   
 64 // show error if user specified invalid file  
 65 if ( fileName == "" || fileName == null )  
 66 MessageBox.Show( "Invalid File Name", "Error",  
 67 MessageBoxButtons.OK, MessageBoxIcon.Error );  
 68   
 69 // open or create file if user specified valid file  
 70 else  
 71 {  
 72 // create Transaction with specified file  
 73 transactionProxy = new Transaction();  
 74 transactionProxy.OpenFile( fileName );  
 75   
 76 // enable GUI buttons except for New/Open File button  
 77 newButton.Enabled = true;  
 78 updateButton.Enabled = true;  
 79 deleteButton.Enabled = true;  
 80 openButton.Enabled = false;  
 81   
 82 // instantiate dialog box for creating records  
 83 newDialog = new NewDialogForm( transactionProxy,  
 84 new MyDelegate( ShowStartDialog ) );  
 85   
 86 // instantiate dialog box for updating records  
 87 updateDialog = new UpdateDialogForm( transactionProxy,  
 88 new MyDelegate( ShowStartDialog ) );  
 89   
 90 // instantiate dialog box for removing records  
 91 deleteDialog = new DeleteDialogForm( transactionProxy,  
 92 new MyDelegate( ShowStartDialog ) );

93   
 94 // set StartDialog as MdiParent for dialog boxes  
 95 newDialog.MdiParent = this.MdiParent;  
 96 updateDialog.MdiParent = this.MdiParent;  
 97 deleteDialog.MdiParent = this.MdiParent;  
 98 }  
 99   
 100 } // end method openButton\_Click  
 101   
 102 // invoked when user clicks New Record button  
 103 private void newButton\_Click(  
 104 object sender, System.EventArgs e )  
 105 {  
 106 Hide(); // hide StartDialog  
 107 newDialog.Show(); // show NewDialog  
 108   
 109 } // end method newButton\_Click  
 110   
 111 private void updateButton\_Click(  
 112 object sender, System.EventArgs e )  
 113 {  
 114 Hide(); // hide StartDialog  
 115 updateDialog.Show(); // show UpdateDialog  
 116   
 117 } // end method updateButton\_Click  
 118   
 119 private void deleteButton\_Click(  
 120 object sender, System.EventArgs e )  
 121 {  
 122 Hide(); // hide StartDialog  
 123 deleteDialog.Show(); // show DeleteDialog  
 124   
 125 } // end method deleteButton\_Click  
 126   
 127 protected void ShowStartDialog()  
 128 {  
 129 Show();  
 130 }  
 131   
 132 } // end class StartDialogForm

Fig. 17.21 UpdateDialogForm class enables users to update records in transaction-processor case study

1 // Fig. 17.21: UpdateDialog.cs  
 2 // Enables user to update records in file.  
 3   
 4 // C# namespaces  
 5 using System;  
 6 using System.Drawing;  
 7 using System.Collections;  
 8 using System.ComponentModel;  
 9 using System.Windows.Forms;  
 10   
 11 // Deitel namespaces  
 12 using BankLibrary;  
 13   
 14 public class UpdateDialogForm : BankUIForm  
 15 {  
 16 private System.Windows.Forms.Label transactionLabel;  
 17 private System.Windows.Forms.TextBox transactionTextBox;  
 18   
 19 private System.Windows.Forms.Button saveButton;  
 20 private System.Windows.Forms.Button cancelButton;  
 21   
 22 private System.ComponentModel.Container components = null;  
 23   
 24 // reference to object that handles transactions  
 25 private Transaction transactionProxy;  
 26   
 27 // delegate for method that displays previous window  
 28 private MyDelegate showPreviousWindow;  
 29   
 30 // initialize components and set members to parameter values  
 31 public UpdateDialogForm(  
 32 Transaction transactionProxyValue,   
 33 MyDelegate delegateValue )  
 34 {  
 35 InitializeComponent();  
 36 showPreviousWindow = delegateValue;  
 37   
 38 // instantiate object that handles transactions  
 39 transactionProxy = transactionProxyValue;  
 40 }  
 41   
 42 // Visual Studio .NET generated code  
 43   
 44 // invoked when user enters text in account TextBox  
 45 private void accountTextBox\_KeyDown(

46 object sender, System.Windows.Forms.KeyEventArgs e )  
 47 {  
 48 // determine whether user pressed Enter key  
 49 if ( e.KeyCode == Keys.Enter )  
 50 {  
 51 // retrieve record associated with account from file  
 52 RandomAccessRecord record =  
 53 transactionProxy.GetRecord( GetTextBoxValues()  
 54 [ ( int )TextBoxIndices.ACCOUNT ] );  
 55   
 56 // return if record does not exist  
 57 if ( record == null )  
 58 return;  
 59   
 60 // determine whether record is empty  
 61 if ( record.Account != 0 )  
 62 {  
 63 // store record values in string array  
 64 string[] values = {  
 65 record.Account.ToString(),  
 66 record.FirstName.ToString(),  
 67 record.LastName.ToString(),  
 68 record.Balance.ToString() };  
 69   
 70 // copy string array value to TextBox values  
 71 SetTextBoxValues( values );  
 72 transactionTextBox.Text = "[Charge or Payment]";  
 73   
 74 }  
 75 else  
 76 {  
 77 // notify user if record does not exist  
 78 MessageBox.Show(  
 79 "Record Does Not Exist", "Error",  
 80 MessageBoxButtons.OK, MessageBoxIcon.Error );  
 81 }  
 82 }  
 83   
 84 } // end method accountTextBox\_KeyDown  
 85   
 86 // invoked when user enters text in transaction TextBox  
 87 private void transactionTextBox\_KeyDown(  
 88 object sender, System.Windows.Forms.KeyEventArgs e )  
 89 {  
 90 // determine whether user pressed Enter key  
 91 if ( e.KeyCode == Keys.Enter )  
 92 {  
 93 // calculate balance using transaction

TextBox value  
 94 try  
 95 {  
 96 // retrieve record associated with account from file  
 97 RandomAccessRecord record =   
 98 transactionProxy.GetRecord( GetTextBoxValues()  
 99 [ ( int )TextBoxIndices.ACCOUNT ] );  
 100   
 101 // get transaction TextBox value  
 102 double transactionValue =   
 103 Double.Parse( transactionTextBox.Text );  
 104   
 105 // calculate new balance (old balance + transaction)  
 106 double newBalance =   
 107 record.Balance + transactionValue;  
 108   
 109 // store record values in string array  
 110 string[] values = {  
 111 record.Account.ToString(),  
 112 record.FirstName.ToString(),  
 113 record.LastName.ToString(),  
 114 newBalance.ToString() };  
 115   
 116 // copy string array value to TextBox values  
 117 SetTextBoxValues( values );  
 118   
 119 // clear transaction TextBox  
 120 transactionTextBox.Text = "";  
 121 }  
 122   
 123 // notify user if error occurs in parameter mismatch  
 124 catch( FormatException )  
 125 {  
 126 MessageBox.Show(   
 127 "Invalid Transaction", "Error",   
 128 MessageBoxButtons.OK, MessageBoxIcon.Error );  
 129 }  
 130 }  
 131   
 132 } // end method transactionTextBox\_KeyDown  
 133   
 134 // invoked when user clicks Save button  
 135 private void saveButton\_Click(  
 136 object sender, System.EventArgs e )  
 137 {  
 138 RandomAccessRecord record =   
 139 transactionProxy.GetRecord( GetTextBoxValues()  
 140 [ ( int )TextBoxIndices.ACCOUNT ] );

141   
 142 // if record exists, update in file  
 143 if ( record != null )  
 144 UpdateRecord( record );  
 145   
 146 Hide();  
 147 ClearTextBoxes();  
 148 showPreviousWindow();  
 149   
 150 } // end method saveButton\_Click  
 151   
 152 // invoked when user clicks Cancel button  
 153 private void cancelButton\_Click(  
 154 object sender, System.EventArgs e )  
 155 {  
 156 Hide();  
 157 ClearTextBoxes();  
 158 showPreviousWindow();  
 159   
 160 } // end method cancelButton\_Click  
 161   
 162 // update record in file at position specified by accountNumber  
 163 public void UpdateRecord( RandomAccessRecord record )  
 164 {  
 165 // store TextBox values in record and write record to file  
 166 try  
 167 {  
 168 int accountNumber = record.Account;  
 169 string[] values = GetTextBoxValues();  
 170   
 171 // store values in record  
 172 record.Account = accountNumber;  
 173 record.FirstName =   
 174 values[ ( int )TextBoxIndices.FIRST ];  
 175 record.LastName =   
 176 values[ ( int )TextBoxIndices.LAST ];  
 177 record.Balance =   
 178 Double.Parse(   
 179 values[ ( int )TextBoxIndices.BALANCE ] );  
 180   
 181 // add record to file  
 182 if ( transactionProxy.AddRecord(  
 183 record, accountNumber ) == false )  
 184   
 185 return; // if error  
 186 }  
 187   
 188 // notify user if error occurs in parameter mismatch  
 189 catch( FormatException )  
 190 {  
 191 MessageBox.Show( "Invalid Balance",

"Error",  
 192 MessageBoxButtons.OK, MessageBoxIcon.Error );  
 193   
 194 return;  
 195 }  
 196   
 197 MessageBox.Show( "Record Updated", "Success",  
 198 MessageBoxButtons.OK,   
 199 MessageBoxIcon.Information );  
 200   
 201 } // end method UpdateRecord  
 202   
 203 } // end class UpdateDialogForm

Fig. 17.22 NewDialogForm class enables users to create records in transaction-processor case study

1 // Fig. 17.22: NewDialog.cs  
 2 // Enables user to insert new record into file.  
 3   
 4 // C# namespaces  
 5 using System;  
 6 using System.Drawing;  
 7 using System.Collections;  
 8 using System.ComponentModel;  
 9 using System.Windows.Forms;  
 10   
 11 // Deitel namespaces  
 12 using BankLibrary;  
 13   
 14 public class NewDialogForm : BankUIForm  
 15 {  
 16 private System.Windows.Forms.Button saveButton;  
 17 private System.Windows.Forms.Button cancelButton;  
 18   
 19 private System.ComponentModel.Container components = null;  
 20   
 21 // reference to object that handles transactions  
 22 private Transaction transactionProxy;  
 23   
 24 // delegate for method that displays previous window  
 25 public MyDelegate showPreviousWindow;  
 26   
 27 // constructor  
 28 public NewDialogForm( Transaction transactionProxyValue,  
 29 MyDelegate delegateValue )  
 30 {  
 31 InitializeComponent();  
 32 showPreviousWindow = delegateValue;  
 33   
 34 // instantiate object that handles transactions  
 35 transactionProxy = transactionProxyValue;  
 36 }  
 37   
 38 // Visual Studio .NET generated code  
 39   
 40 // invoked when user clicks Cancel button  
 41 private void cancelButton\_Click(  
 42 object sender, System.EventArgs e )  
 43 {  
 44 Hide();  
 45 ClearTextBoxes();  
 46 showPreviousWindow();  
 47   
 48 } // end method cancelButton\_Click  
 49

50 // invoked when user clicks Save As button  
 51 private void saveButton\_Click(  
 52 object sender, System.EventArgs e )  
 53 {  
 54 RandomAccessRecord record =  
 55 transactionProxy.GetRecord( GetTextBoxValues()  
 56 [ ( int )TextBoxIndices.ACCOUNT ] );  
 57   
 58 // if record exists, add it to file  
 59 if ( record != null )  
 60 InsertRecord( record );  
 61   
 62 Hide();  
 63 ClearTextBoxes();  
 64 showPreviousWindow();  
 65   
 66 } // end method saveButton\_Click  
 67   
 68 // insert record in file at position specified by accountNumber  
 69 private void InsertRecord( RandomAccessRecord record )  
 70 {  
 71 //store TextBox values in string array  
 72 string[] textBoxValues = GetTextBoxValues();  
 73   
 74 // store TextBox account field  
 75 int accountNumber = Int32.Parse(  
 76 textBoxValues[ ( int )TextBoxIndices.ACCOUNT ] );  
 77   
 78 // notify user and return if record account is not empty  
 79 if ( record.Account != 0 )  
 80 {  
 81 MessageBox.Show(  
 82 "Record Already Exists or Invalid Number", "Error",  
 83 MessageBoxButtons.OK, MessageBoxIcon.Error);  
 84   
 85 return;  
 86 }  
 87   
 88 // store values in record  
 89 record.Account = accountNumber;  
 90 record.FirstName =   
 91 textBoxValues[ ( int )TextBoxIndices.FIRST];  
 92 record.LastName =   
 93 textBoxValues[ ( int )TextBoxIndices.LAST];  
 94 record.Balance = Double.Parse(   
 95 textBoxValues[ ( int )TextBoxIndices.BALANCE ] );  
 96

97 // add record to file  
 98 try  
 99 {  
 100 if ( transactionProxy.AddRecord(  
 101 record, accountNumber ) == false )  
 102   
 103 return; // if error  
 104 }  
 105   
 106 // notify user if error occurs in parameter mismatch  
 107 catch( FormatException )  
 108 {  
 109 MessageBox.Show( "Invalid Balance", "Error",  
 110 MessageBoxButtons.OK, MessageBoxIcon.Error );  
 111 }  
 112   
 113 MessageBox.Show( "Record Created", "Success",  
 114 MessageBoxButtons.OK, MessageBoxIcon.Information );  
 115   
 116 } // end method InsertRecord  
 117   
 118 } // end class NewDialogForm

Fig. 17.23 DeleteDialogForm class enables users to remove records from files in transaction-processor case study

1 // Fig. 17.23: DeleteDialog.cs  
 2 // Enables user to delete records in file.  
 3   
 4 // C# namespaces  
 5 using System;  
 6 using System.Drawing;  
 7 using System.Collections;  
 8 using System.ComponentModel;  
 9 using System.Windows.Forms;  
 10   
 11 // Deitel namespaces  
 12 using BankLibrary;  
 13   
 14 public class DeleteDialogForm : System.Windows.Forms.Form  
 15 {  
 16 private System.Windows.Forms.Label accountLabel;  
 17 private System.Windows.Forms.TextBox accountTextBox;  
 18   
 19 private System.Windows.Forms.Button deleteButton;  
 20 private System.Windows.Forms.Button cancelButton;  
 21   
 22 private System.ComponentModel.Container components = null;  
 23   
 24 // reference to object that handles transactions  
 25 private Transaction transactionProxy;  
 26   
 27 // delegate for method that displays previous window  
 28 private MyDelegate showPreviousWindow;  
 29   
 30 // initialize components and set members to parameter values  
 31 public DeleteDialogForm( Transaction transactionProxyValue,   
 32 MyDelegate delegateValue)  
 33 {  
 34 InitializeComponent();  
 35 showPreviousWindow = delegateValue;  
 36   
 37 // instantiate object that handles transactions  
 38 transactionProxy = transactionProxyValue;  
 39 }  
 40   
 41 // Visual Studio .NET generated code  
 42   
 43 // invoked when user clicks Delete Record button  
 44 private void deleteButton\_Click(  
 45 object sender, System.EventArgs e)  
 46 {

47 RandomAccessRecord record =  
 48 transactionProxy.GetRecord( accountTextBox.Text );  
 49   
 50 // if record exists, delete it in file  
 51 if ( record != null )  
 52 DeleteRecord( record );  
 53   
 54 this.Hide();  
 55 showPreviousWindow();  
 56   
 57 } // end method deleteButton\_Click  
 58   
 59 // invoked when user clicks Cancel button  
 60 private void cancelButton\_Click(  
 61 object sender, System.EventArgs e)  
 62 {  
 63 this.Hide();  
 64 showPreviousWindow();  
 65   
 66 } // end method cancelButton\_Click  
 67   
 68 // delete record in file at position specified by accountNumber  
 69 public void DeleteRecord( RandomAccessRecord record )  
 70 {  
 71 int accountNumber = record.Account;  
 72   
 73 // display error message if record does not exist  
 74 if ( record.Account == 0 )  
 75 {  
 76 MessageBox.Show( "Record Does Not Exist", "Error",  
 77 MessageBoxButtons.OK, MessageBoxIcon.Error );  
 78 accountTextBox.Clear();  
 79   
 80 return;  
 81 }  
 82   
 83 // create blank record  
 84 record = new RandomAccessRecord();  
 85   
 86 // write over file record with empty record  
 87 if ( transactionProxy.AddRecord(  
 88 record, accountNumber ) == true )  
 89   
 90 // notify user of successful deletion  
 91 MessageBox.Show( "Record Deleted", "Success",  
 92 MessageBoxButtons.OK, MessageBoxIcon.Information );  
 93 else  
 94

95 // notify user of failure  
 96 MessageBox.Show(   
 97 "Record could not be deleted", "Error",  
 98 MessageBoxButtons.OK, MessageBoxIcon.Error );  
 99   
 100 accountTextBox.Clear();  
 101   
 102 } // end method DeleteRecord  
 103   
 104 } // end class DeleteDialogForm

Case Study discussion and Figs. 17.18 through to 17.23 code walkthroughs

Transaction-Processing Behaviour

In this case study, we create class Transaction (Fig. [17.18](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.18','codesampleswin');)), which acts as a proxy to handle all transaction processing

Rather than providing the transaction-processing behaviour themselves, the objects in this application use an instance of Transaction to provide the necessary functionality

By using a proxy, we can encapsulate transaction-processing behaviour in only one class, enabling various other classes in our application to reuse this behaviour

Furthermore, if we decide to modify this behaviour, we modify only the proxy

(i.e., class Transaction), instead of having to modify the behaviour of each class that uses the proxy

Class Transaction contains methods OpenFile, GetRecord and AddRecord

Method OpenFile (Fig. 17.18, lines 27 - 74) uses constant FileMode.OpenOrCreate (line 33) to create a FileStream object from either an existing file or one not yet created

Lines 36 - 39 use this FileStream to create BinaryReader and BinaryWriter objects for reading and writing bytes to the file, respectively

If the file is new, lines 42 - 64 populate the FileStream object with empty records

Remember, we used these techniques when we created a random – access file

Method GetRecord (lines 77 - 122) returns the record associated with the account-number parameter

Line 83 instantiates a RandomAccessRecord object that will store the file data

If the account parameter is valid, lines 100 - 101 call method Seek of the FileStream object, which uses the parameter to determine the position of the specified record in the file

Lines 104 - 107 then call methods ReadInt32, ReadString and ReadDouble of the BinaryReader object to store the file data in the RandomAccessRecord object

Line 110 returns the RandomAccessRecord object

Remember, we used these techniques when reading data sequentially from a random – access file

Method AddRecord (lines 125 - 153) inserts a record into the file

Lines 132 - 133 call method Seek of the FileStream object, which uses the account-number parameter to locate the position at which to insert the record in the file

Lines 136 - 139 call the overloaded Write methods of the BinaryWriter object to write the RandomAccessRecord object’s data to the file

Remember, we used these techniques when we writing data randomly to

a random – access file

Note that, if an error occurs when adding the record (i.e., either the FileStream or the BinaryWriter throws an IOException), lines 145 - 146 notify the user of the error and return false (failure)

Transaction-Processor GUI

The GUI for this program uses a multiple-document interface

Class TransactionProcessorForm (Fig. [17.19](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.19','codesampleswin');)) is the parent window, and contains corresponding child windows

StartDialogForm (Fig. [17.20](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.20','codesampleswin');)),

NewDialogForm (Fig. [17.22](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.22','codesampleswin');)),

UpdateDialogForm (Fig. [17.21](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.21','codesampleswin');))

and DeleteDialogForm (Fig. [17.23](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.23','codesampleswin');))

StartDialogForm allows the user to open a file containing account information and provides access to the NewDialogForm, UpdateDialogForm and DeleteDialogForm internal frames

These frames allow users to update, create and delete records, respectively

Initially, TransactionProcessorForm displays the StartDialogForm object; this window provides the user with various options

It contains four buttons, which enable the user to create or open a file, create a record, update an existing record or delete an existing record

Before the user can modify records, the user must either create or open a file

When the user clicks the New/Open File button, the program calls method openButton\_Click (Fig. 17.20, lines 42 - 100), which opens a file that the application uses for modifying records

Lines 46 - 62 display the OpenFileDialog for specifying the file from which to read data, then use this file to create the FileStream object

Note that line 52 sets property CheckFileExists of the OpenFileDialog object to false - this enables the user to create a file if the specified file does not exist

If this property were true (its default value), the dialog would notify the user that the specified file does not exist, thus preventing the user from creating a file

If the user specifies a file name, line 73 instantiates an object of class Transaction

(of Fig. [17.18](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.18','codesampleswin');)), which acts as the proxy for creating, reading records from and writing records to random-access files

Line 74 calls Transaction’s method OpenFile, which either creates or opens the specified file, depending on whether the file exists

Class StartDialogForm also creates internal windows that enable the user to create, update and delete records

We do not use the default constructor created by Visual Studio .NET for these classes; instead, we use an overloaded constructor that takes as arguments the Transaction object and a delegate object that references method ShowStartDialog

(lines 127 - 130)

Each child window uses the second delegate parameter to display the StartDialogForm GUI when the user closes a child window

Lines 83–92 instantiate objects of classes UpdateDialogForm, NewDialogForm and DeleteDialogForm, which serve as the child windows

When the user clicks the New Record button in the Start Dialog, the program invokes method newButton\_Click of class StartDialogForm (Fig. [17.20](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.20','codesampleswin');), lines 103 - 109), which displays the NewDialogForm internal frame (Fig. [17.22](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.22','codesampleswin');))

Class NewDialogForm enables the user to create records in the file that StartDialogForm opened (or created)

Line 25 of Fig. [17.22](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.22','codesampleswin');) defines MyDelegate as a delegate to a method that does not return a value and has no parameters; method ShowStartDialog of class StartDialogForm (Fig. [17.20](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.20','codesampleswin');), lines 127 -130) conforms to these requirements

Class NewDialogForm receives a MyDelegate object, which references this method as a parameter - therefore, NewDialogForm can invoke this method to display the start window when the user exits the NewDialogForm

Classes UpdateDialogForm and DeleteDialogForm also receive MyDelegate references as arguments, enabling them to display StartDialogForm after completing their tasks

After the user enters data in the TextBoxes and clicks the Save Record button, the program invokes method saveButton\_Click (Fig. 17.22, lines 51 - 66) to write the record to disk

Lines 54 - 56 call method GetRecord of the Transaction object, which should return an empty RandomAccessRecord

If method GetRecord returns a RandomAccessRecord that contains data, the user is attempting to overwrite that RandomAccessRecord with a new one

Line 60 calls private method InsertRecord (lines 69 - 116)

If the RandomAccessRecord is empty, method InsertRecord calls method AddRecord of the Transaction object (lines 100 - 101), which adds the newly created RandomAccessRecord to the file

If the user is attempting to overwrite an existing record, lines 81 - 83 notify the user that the record already exists and return from the method

When the user clicks the Update Record button in the Start Dialog, the program invokes method updateButton\_Click of class StartDialogForm

(Fig. [17.20](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.20','codesampleswin');), lines 111 - 117), which displays the UpdateDialogForm internal frame (Fig. [17.21](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.21','codesampleswin');))

Class UpdateDialogForm enables the user to update existing records in the file

To update a record, the user must enter the account number associated with that record

When the user presses Enter, UpdateDialogForm calls method accountTextBox\_KeyDown (Fig. [17.21](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.21','codesampleswin');), lines 45 - 84) to display the record contents

This method calls method GetRecord of the Transaction object (lines 52 - 54) to retrieve the specified RandomAccessRecord

If the record is not empty, lines 64 - 72 populate the TextBoxes with the RandomAccessRecord values

The Transaction TextBox initially contains the string **Charge or Payment**

The user should select this text, type the transaction amount (a positive value for a charge or a negative value for a payment), then press Enter

The program calls method transactionTextBox\_KeyDown (lines 87 - 132) to add the user-specified transaction amount to the current balance

The user clicks the Save Changes button to write the altered contents of the TextBoxes to the file

(Note that pressing Save Changes does not update the Balance field—the user must press Enter to update this field before pressing Save Changes)

When the user clicks Save Changes, the program invokes method saveButton\_Click (lines 135 - 150), which calls private method UpdateRecord (lines 163 - 201)

This method calls method AddRecord of the Transaction object (lines 182 - 183) to store the TextBox values in a RandomAccessRecord and overwrite the existing file record with the RandomAccessRecord containing the new data

When the user clicks the Delete Record button of the Start Dialog, the program invokes method deleteButton\_Click of class StartDialogForm (Fig. [17.20](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.20','codesampleswin');), lines

119 - 125), which displays the DeleteDialogForm internal frame (Fig. [17.23](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.23','codesampleswin');))

Class DeleteDialogForm enables the user to remove existing records from the file

To remove a record, the user must enter the account number associated with that record

When the user clicks the Delete Record button (now, from the DeleteDialogForm internal frame), DeleteDialogForm calls method deleteButton\_Click (Fig. [17.23](javascript:openPopUpWindow('../../ch17/codesamples/content.htm#Fig17.23','codesampleswin');), lines 44 - 57)

This method calls method DeleteRecord (lines 69 - 102), which ensures that the record to be deleted exists, then calls method AddRecord of the Transaction object (lines 87 - 88) to overwrite the file record with an empty one

In this booklet, we demonstrated how to read data from files and write data to files via both sequential-access and random-access file-processing techniques

Using class BinaryFormatter, we serialized and deserialized objects to and from streams; we then employed FileStream, BinaryWriter and BinaryReader to transfer the objects’ byte representation to and from files

In the Extensible Markup Language booklet, we discuss XML, a widely supported technology for describing data

Using XML, we can describe any type of data, such as mathematical formulas, music and financial reports

Booklet Summary

Files and Streams: A Summary

* All data items processed by a computer ultimately are reduced to combinations of zeros and ones
* The smallest data items that computers support are called bits and can assume either the value 0 or the value 1
* Digits, letters and special symbols are referred to as characters. The set of all characters used to write programs and represent data items on a particular computer is called that computer’s character set. Every character in a computer’s character set is represented as a pattern of 1s and 0s (characters in C# are Unicode characters, which are composed of 2 bytes)
* At least one field in a record is chosen as a record key, which identifies that record as belonging to a particular person or entity and distinguishes that record from all other records in the file
* A file is a group of related records
* Files are used for long-term retention of large amounts of data and can store those data even after the program that created the data terminates
* Data maintained in files often are called persistent data
* Class File enables programs to obtain information about a file
* Class Directory enables programs to obtain information about a directory
* Class FileStream provides method Seek for repositioning the file-position pointer (the byte number of the next byte in the file to be read or written) to any position in the file
* The most common type of file organization is the sequential file, in which records typically are stored in order by the record-key field
* When a file is opened, an object is created, and a stream is associated with the object
* C# imposes no structure on files. This means that concepts like that of a “record” do not exist in C#. The programmer must structure each file appropriately to meet the requirements of an application
* A collection of programs designed to create and manage databases is called a database management system (DBMS)
* C# views each file as a sequential stream of bytes
* Each file ends in some machine-dependent form of end-of-file marker
* Objects of classes OpenFileDialog and SaveFileDialog are used for selecting files to open and save, respectively. Method ShowDialog of these classes displays that dialog
* When displayed, both an OpenFileDialog and a SaveFileDialog prevent the user from interacting with any other program window until the dialog is closed. Dialogs that behave in this fashion are called modal dialogs
* Streams provide communication channels between files and programs
* To perform file processing in C#, the namespace System.IO must be referenced. This namespace includes definitions for stream classes such as StreamReader, StreamWriter and FileStream. Files are opened by instantiating objects of these classes
* To retrieve data sequentially from a file, programs normally start from the beginning of the file, reading all data consecutively until the desired data are found
* With a sequential-access file, each successive input/output request reads or writes the next consecutive set of data in the file
* Instant data access is possible with random-access files. A program can access individual records of a random-access file directly (and quickly) without searching through other records. Random-access files sometimes are called direct-access files
* With a random-access file, each successive input/output request can be directed to any part of the file, which can be any distance from the part of the file referenced in the previous request
* Programmers can use members of the FileAccess enumeration to control users’ access to files
* Only classes with the Serializable attribute can be serialized to and deserialized from files
* There are a variety of techniques for creating random-access files. Perhaps the simplest involves requiring that all records in a file be of the same fixed length
* The use of fixed-length records makes it easy for a program to calculate (as a function of the record size and the record key) the exact location of any record in relation to the beginning of the file
* Data can be inserted into a random-access file without destroying other data in the file. Users can also update or delete previously stored data without rewriting the entire file
* BinaryFormatter uses methods Serialize and Deserialize to write and to read objects, respectively. Method Serialize writes the object’s representation to a stream. Method Deserialize reads this representation from a stream and reconstructs the original object
* Methods Serialize and Deserialize each require a Stream object as a parameter, enabling the BinaryFormatter to access the correct file
* Class BinaryReader and BinaryWriter provide methods for reading and writing bytes to streams, respectively. The BinaryReader and BinaryWriter constructors receive as arguments references to instances of class System.IO.Stream
* Class FileStream inherits from class Stream, so we can pass the FileStream object as an argument to either the BinaryReader or BinaryWriter constructor to create an object that can transfer bytes directly to or from a file
* Random-access file-processing programs rarely write a single field to a file. Normally, they write one object at a time
* Sorting with direct-access techniques is fast. This speed is achieved by making the file large enough to hold every possible record that might be created. Of course, this means that the file could be sparsely occupied most of the time, possibly wasting memory

Self-Review Exercises

Self-Review Exercises

Part 1

State whether each of the following is **true** or **false**

If **false**, explain why

1. Creating instances of classes File and Directory is impossible
2. Typically, a sequential file stores records in order by the record-key field
3. Class StreamReader inherits from class Stream
4. Any class can be serialized to a file
5. Searching a random-access file sequentially to find a specific record is unnecessary
6. Method Seek of class FileStream always seeks relative to the beginning of a file
7. C# provides class Record to store records for random-access file-processing applications
8. Banking systems, point-of-sale systems and automated-teller machines are types of transaction-processing systems
9. Classes StreamReader and StreamWriter are used with sequential-access files
10. Instantiating objects of type Stream is impossible

Part 2

Fill in the blanks for each of the following statements

1. Ultimately, all data items processed by a computer are reduced to combinations of \_\_\_\_\_\_\_\_ and \_\_\_\_\_\_\_\_
2. The smallest data item a computer can process is called a \_\_\_\_\_\_\_\_
3. A \_\_\_\_\_\_\_\_ is a group of related records
4. Digits, letters and special symbols are collectively referred to as \_\_\_\_\_\_\_\_
5. A group of related files is called a \_\_\_\_\_\_\_\_
6. StreamReader method \_\_\_\_\_\_\_\_ reads a line of text from a file
7. StreamWriter method \_\_\_\_\_\_\_\_ writes a line of text to a file
8. Method Serialize of class BinaryFormatter takes a(n) \_\_\_\_\_\_\_\_ and a(n) \_\_\_\_\_\_\_\_ as arguments
9. The \_\_\_\_\_\_\_\_ namespace contains most of C#’s file-processing classes
10. The \_\_\_\_\_\_\_\_ namespace contains the BinaryFormatter class