# Files

Source : [BOOK] How to programming C#

# Data Hierarchy

**Bits**

The smallest data item that computers support is called a bit (short for “binary digit”—a digit that can assume one of two values).Each bit can assume either the value 0 or the value 1.

**Characters**

Programming with data in the low-level form of bits is cumbersome. It’s 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–Z and a–z) and special symbols (i.e., $, @, %, &, *, (, ), -, +, ", :, ?, / and many others). Digits, letters and special symbols are referred to as characters.

# Data Hierarchy

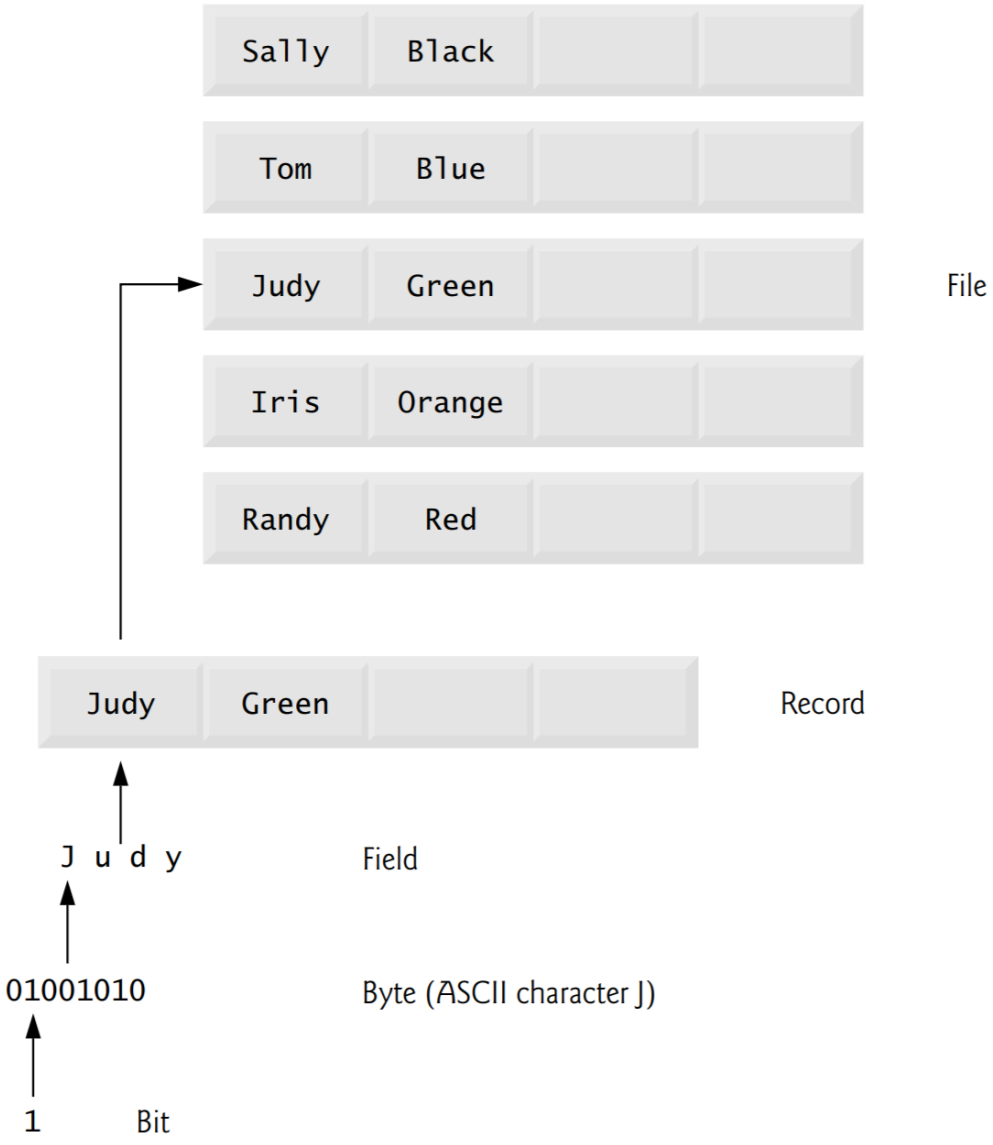
Continue

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 0s and 1s, every character in a computer's character set is represented as a pattern of 0s and 1s. Bytes are composed of eight bits. C# uses the Unicode® character set. Programmers create programs and data items with characters; computers manipulate and process these characters as patterns of bits

**Fields**

Just as characters are composed of bits, fields are composed of characters. A field is a group of characters that conveys meaning. For example, a field consisting of uppercase and lowercase letters can represent a person's name.

# Data hierarchy



# Record Key

To facilitate the retrieval of specific records from a file, at least one field in each record is chosen as a record key, which identifies a record as belonging to a particular person or entity and distinguishes that record from all others. For example, in a payroll record, the employee identification number normally would be the record key.

# Sequential Files

A common organization of records in a file is called a sequential file, in which records typically are stored in order by a record-key field. In a payroll file, records usually are placed in order by employee identification number. The first employee record in the file contains the lowest employee identification number, and subsequent records contain increasingly higher ones.

# Files and Streams

C# views each file as a sequential stream of bytes.

Each file ends with an end-of-file marker.

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


**Fig. 17.2** | C#'s view of an *n*-byte file.

# Files and Streams

Continue

There are many file-processing classes in the Framework Class Library. The System.IO namespace includes 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).

# Creating a Sequential-Access Text File

C# imposes no structure on files. Thus, the concept of a “record” does not exist in C# files. This means that you must structure files to meet the requirements of your apps. The next few examples use text and special characters to organize our own concept of a “record.”

```
// Fig. 17.7: BankUIForm.cs
// A reusable Windows Form for the examples in this chapter.
using System;
using System.Windows.Forms;

namespace BankLibrary
{
   public partial class BankUIForm : Form
   {
      protected int TextBoxCount = 4; // number of TextBoxes on Form

      // enumeration constants specify TextBox indices
      public enum TextBoxIndices
      {
         ACCOUNT,
         FIRST,
         LAST,
         BALANCE
      } // end enum
```

```
// parameterless constructor
      public BankUIForm()
      {
         InitializeComponent();
      } // end constructor

      // clear all TextBoxes
      public void ClearTextBoxes()
      {
         // iterate through every Control on form
         foreach ( Control guiControl in Controls )
         {
            // determine whether Control is TextBox
            if ( guiControl is TextBox )
            {
               // clear TextBox
               ( ( TextBox ) guiControl ).Clear();
            } // end if
         } // end for
      } // end method ClearTextBoxes
```

```
// set text box values to string array values
public void SetTextBoxValues( string[] values )
{
   // determine whether string array has correct length
   if ( values.Length != TextBoxCount )
   {
      // throw exception if not correct length
      throw ( new ArgumentException( "There must be " +
         ( TextBoxCount ) + " strings in the array" ) );
   } // end if
   // set array values if array has correct length
   else
   {
      // set array values to TextBox values
      accountTextBox.Text =
         values[ ( int ) TextBoxIndices.ACCOUNT ];
      firstNameTextBox.Text =
         values[ ( int ) TextBoxIndices.FIRST ];
      lastNameTextBox.Text = values[ ( int ) TextBoxIndices.LAST ];
      balanceTextBox.Text =
         values[ ( int ) TextBoxIndices.BALANCE ];
   } // end else
} // end method SetTextBoxValues
```

```
// return TextBox values as string array
public string[] GetTextBoxValues()
{
   string[] values = new string[ TextBoxCount ];

   // copy TextBox fields to string array
   values[ ( int ) TextBoxIndices.ACCOUNT ] = accountTextBox.Text;
   values[ ( int ) TextBoxIndices.FIRST ] = firstNameTextBox.Text;
   values[ ( int ) TextBoxIndices.LAST ] = lastNameTextBox.Text;
   values[ ( int ) TextBoxIndices.BALANCE ] = balanceTextBox.Text;

   return values;
} // end method GetTextBoxValues
```


**Fig. 17.7** | Base class for GUIs in our file-processing apps. (Part 3 of 3.)

```
// Fig. 17.8: Record.cs
// Class that represents a data record.
namespace BankLibrary
{
   public class Record
   {
      public int Account { get; set; }
      public string FirstName { get; set; }
      public string LastName { get; set; }
      public decimal Balance { get; set; }
      public Record() : this(0, string.Empty, string.Empty, 0M) { }
      public Record( int accountValue, string firstNameValue,string lastNameValue,
                     decimal balanceValue )
      {
         Account = accountValue;
         FirstName = firstNameValue;
         LastName = lastNameValue;
         Balance = balanceValue;
      }
   }
}
```

```
// Fig. 17.9: CreateFileForm.cs
// Creating a sequential-access file.
using System;
using System.Windows.Forms;
using System.IO;
using BankLibrary;

namespace CreateFile
{
   public partial class CreateFileForm : BankUIForm
   {
      private StreamWriter fileWriter;
      public CreateFileForm()
      {
         InitializeComponent();
      }
```

```
private void saveButton_Click( object sender, EventArgs e )
{
        DialogResult result; // result of SaveFileDialog
        string fileName; // name of file containing data
        SaveFileDialog fileChooser = new SaveFileDialog();
        fileChooser.CheckFileExists = false; // let user create file
         result = fileChooser.ShowDialog();
         fileName = fileChooser.FileName; // name of file to save data
        if ( result == DialogResult.OK )
         {
            if ( fileName == string.Empty )
               MessageBox.Show( "Invalid File Name", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
            else
            {
               try
               {
                  FileStream output = new FileStream( fileName,FileMode.OpenOrCreate, FileAccess.Write );
                  fileWriter = new StreamWriter( output );
                  saveButton.Enabled = false;
                  enterButton.Enabled = true;
               }
               catch ( IOException )
               {
                  MessageBox.Show( "Error opening file", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error );
               }
            }
         }
      }
```

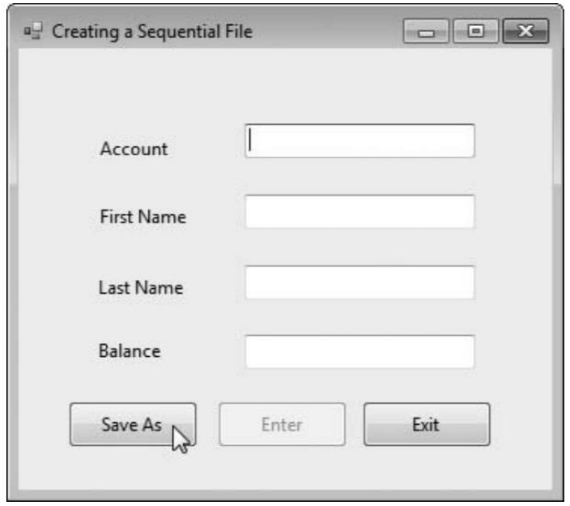
```
private void enterButton_Click( object sender, EventArgs e )
{
   // store TextBox values string array
   string[] values = GetTextBoxValues();

   // Record containing TextBox values to output
   Record record = new Record();

   // determine whether TextBox account field is empty
   if ( values[ ( int ) TextBoxIndices.ACCOUNT ] != string.Empty )
   {
      // store TextBox values in Record and output it
      try
      {
         // get account number value from TextBox
         int accountNumber = Int32.Parse(
            values[ ( int ) TextBoxIndices.ACCOUNT ] );
```

```
      // determine whether accountNumber is valid
      if ( accountNumber > 0 )
      {
         // store TextBox fields in Record
         record.Account = accountNumber;
         record.FirstName = values[ ( int )
            TextBoxIndices.FIRST ];
         record.LastName = values[ ( int )
            TextBoxIndices.LAST ];
         record.Balance = Decimal.Parse(
            values[ ( int ) TextBoxIndices.BALANCE ] );

         // write Record to file, fields separated by commas
         fileWriter.WriteLine(
            record.Account + "," + record.FirstName + "," +
            record.LastName + "," + record.Balance );
      } // end if
      else
      {
         // notify user if invalid account number
         MessageBox.Show( "Invalid Account Number", "Error",
            MessageBoxButtons.OK, MessageBoxIcon.Error );
      } // end else
   } // end try
```

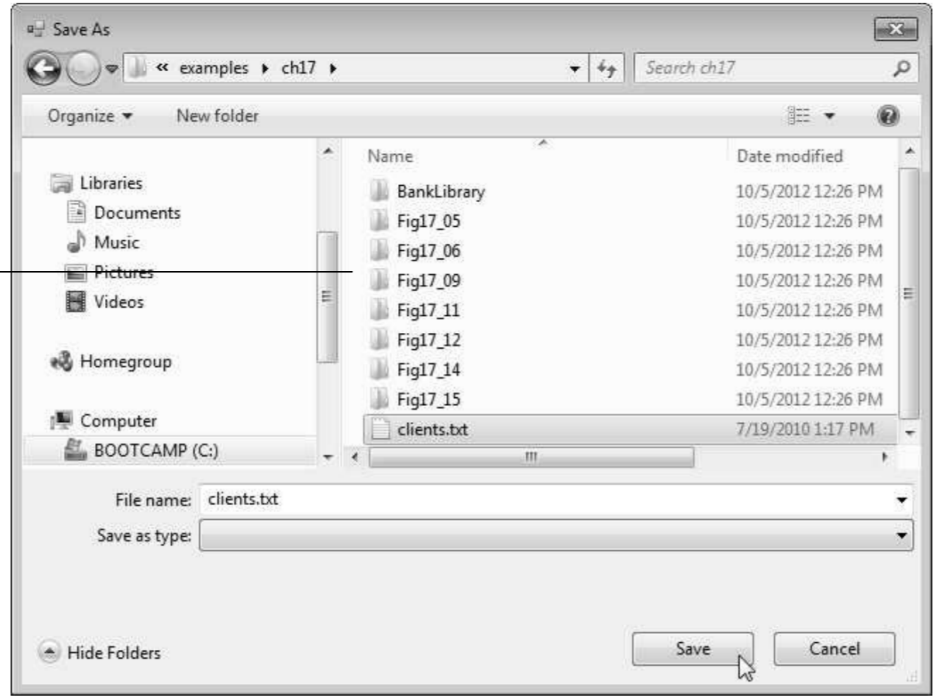
```
      // notify user if error occurs during write operation
      catch ( IOException )
      {
         MessageBox.Show( "Error Writing to File", "Error",
            MessageBoxButtons.OK, MessageBoxIcon.Error );
      } // end catch
      // notify user if error occurs regarding parameter format
      catch ( FormatException )
      {
         MessageBox.Show( "Invalid Format", "Error",
            MessageBoxButtons.OK, MessageBoxIcon.Error );
      } // end catch
   } // end if

   ClearTextBoxes(); // clear TextBox values
} // end method enterButton_Click
```

```
private void exitButton_Click( object sender, EventArgs e )
      {
         // determine whether file exists
         if ( fileWriter != null )
         {
            try
            {
               // close StreamWriter and underlying file
               fileWriter?.Close();
            } // end try
            // notify user of error closing file
            catch ( IOException )
            {
               MessageBox.Show( "Cannot close file", "Error",
                  MessageBoxButtons.OK, MessageBoxIcon.Error );
            } // end catch
         } // end if

         Application.Exit();
      } // end method exitButton_Click
```

a) BankUI graphical user interface with three additional controls

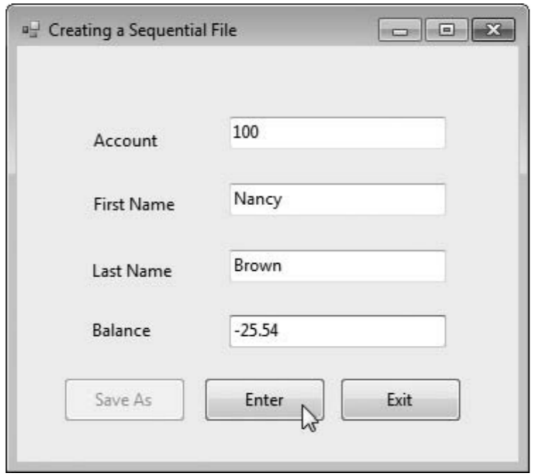



b) Save File dialog




Files and directories

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



# Reading Data from a Sequential-Access Text File

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

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

```
// Fig. 17.11: ReadSequentialAccessFileForm.cs
// Reading a sequential-access file.
using System;
using System.Windows.Forms;
using System.IO;
using BankLibrary;

namespace ReadSequentialAccessFile
{
   public partial class ReadSequentialAccessFileForm : BankUIForm
   {
      private StreamReader fileReader;
      public ReadSequentialAccessFileForm()
      {
         InitializeComponent();
      }
```

```
private void openButton_Click( object sender, EventArgs e )
      {
         // create and show dialog box enabling user to open file
         DialogResult result; // result of OpenFileDialog
         string fileName; // name of file containing data

         OpenFileDialog fileChooser = new OpenFileDialog();
         result = fileChooser.ShowDialog();
         fileName = fileChooser.FileName; // get specified name

         // ensure that user clicked "OK"
         if ( result == DialogResult.OK )
         {
            ClearTextBoxes();

            // show error if user specified invalid file
            if ( fileName == string.Empty )
               MessageBox.Show( "Invalid File Name", "Error",
                  MessageBoxButtons.OK, MessageBoxIcon.Error );
```

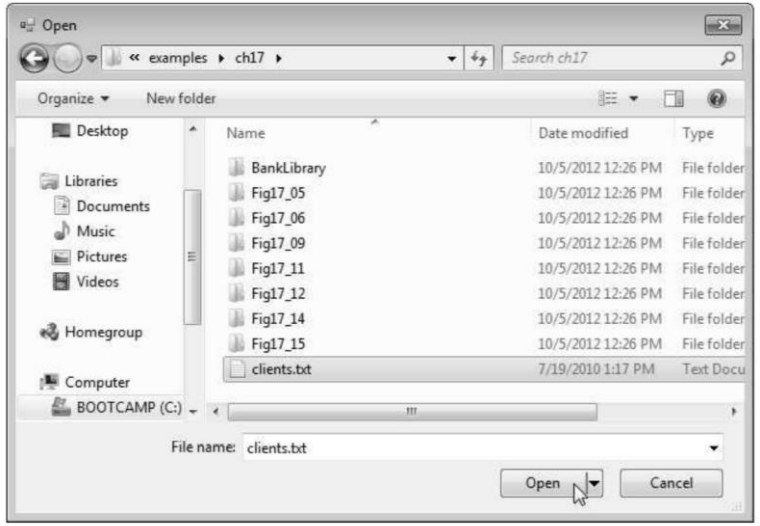
```
        else
        {
            try
            {
                // create FileStream to obtain read access to file
                FileStream input = new FileStream(
                    fileName, FileMode.Open, FileAccess.Read );

                // set file from where data is read
                fileReader = new StreamReader( input );

                openButton.Enabled = false; // disable Open File button
                nextButton.Enabled = true; // enable Next Record button
            } // end try
            catch ( IOException )
            {
                MessageBox.Show( "Error reading from file",
                    "File Error", MessageBoxButtons.OK,
                    MessageBoxIcon.Error );
            } // end catch
        } // end else
    } // end if
} // end method openButton_Click
```

```
private void nextButton_Click( object sender, EventArgs e )
{
   try
   {
      // get next record available in file
      string inputRecord = fileReader.ReadLine();
      string[] inputFields; // will store individual pieces of data

      if ( inputRecord != null )
      {
         inputFields = inputRecord.Split( ',' );

         Record record = new Record(
            Convert.ToInt32( inputFields[ 0 ] ), inputFields[ 1 ],
            inputFields[ 2 ],
            Convert.ToDecimal( inputFields[ 3 ] ) );

         // copy string array values to TextBox values
         SetTextBoxValues( inputFields );
      } // end if
```

```
        else
        {
            // close StreamReader and underlying file
            fileReader.Close();
            openButton.Enabled = true; // enable Open File button
            nextButton.Enabled = false; // disable Next Record button
            ClearTextBoxes();

            // notify user if no Records in file
            MessageBox.Show( "No more records in file", String.Empty,
                MessageBoxButtons.OK, MessageBoxIcon.Information );
        } // end else
    } // end try
    catch ( IOException )
    {
        MessageBox.Show( "Error Reading from File", "Error",
            MessageBoxButtons.OK, MessageBoxIcon.Error );
    } // end catch
} // end method nextButton_Click
```

a) BankUI graphical user interface with an Open File button


b) `OpenFileDialog` window




c) Reading account 100


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

# Classes File and Directory

Information is stored in files, which are organized in directories (also called folders). Classes File and Directory enable programs to manipulate files and directories on disk. Class File can determine information about files and can be used to open files for reading or writing. We discuss techniques for writing to and reading from files in subsequent sections.

| static Method | Description |
|---|---|
| AppendText | Returns a StreamWriter that appends text to an existing file or creates a file if one does not exist. |
| Copy | Copies a file to a new file. |
| Create | Creates a file and returns its associated FileStream. |
| CreateText | Creates a text file and returns its associated StreamWriter. |
| Delete | Deletes the specified file. |
| Exists | Returns true if the specified file exists and false otherwise. |
| GetCreationTime | Returns a DateTime object representing when the file was created. |
| GetLastAccessTime | Returns a DateTime object representing when the file was last accessed. |
| GetLastWriteTime | Returns a DateTime object representing when the file was last modified. |
| Move | Moves the specified file to a specified location. |
| Open | Returns a FileStream associated with the specified file and equipped with the specified read/write permissions. |
| OpenRead | Returns a read-only FileStream associated with the specified file. |
| OpenText | Returns a StreamReader associated with the specified file. |
| OpenWrite | Returns a write FileStream associated with the specified file. |

**Fig. 17.3** | File class static methods (partial list).

# Case Study: Credit Inquiry Program

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

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. When a FileStream object is opened, its file-position pointer is set to byte position 0 (i.e., the beginning of the file)

Credit-inquiry program enables a credit manager

# Case Study: Credit Inquiry Program

Continue

to search for and 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). We use a RichTextBox in the program to display the account information.

```
// Fig. 17.12: CreditInquiryForm.cs
// Read a file sequentially and display contents based on
// account type specified by user ( credit, debit or zero balances ).
using System;
using System.Windows.Forms;
using System.IO;
using BankLibrary;

namespace CreditInquiry
{
   public partial class CreditInquiryForm : Form
   {
      private FileStream input; // maintains the connection to the file
      private StreamReader fileReader; // reads data from text file

      // name of file that stores credit, debit and zero balances
      private string fileName;

      // parameterless constructor
      public CreditInquiryForm()
      {
         InitializeComponent();
      } // end constructor
```

```
// invoked when user clicks Open File button
      private void openButton_Click( object sender, EventArgs e )
      {

         DialogResult result;
         OpenFileDialog fileChooser = new OpenFileDialog() )
         result = fileChooser.ShowDialog();
         fileName = fileChooser.FileName;

         // exit event handler if user clicked Cancel
         if ( result == DialogResult.OK )
         {
            // show error if user specified invalid file
            if ( fileName == string.Empty )
               MessageBox.Show( "Invalid File Name", "Error",
                  MessageBoxButtons.OK, MessageBoxIcon.Error );
```

```
        else
        {
           // create FileStream to obtain read access to file
           input = new FileStream( fileName,
              FileMode.Open, FileAccess.Read );

           // set file from where data is read
           fileReader = new StreamReader( input );

           // enable all GUI buttons, except for Open File button
           openButton.Enabled = false;
           creditButton.Enabled = true;
           debitButton.Enabled = true;
           zeroButton.Enabled = true;
        } // end else
     } // end if
  } // end method openButton_Click
```

```
// invoked when user clicks credit balances,
// debit balances or zero balances button
private void getBalances_Click( object sender, System.EventArgs e )
{
   // convert sender explicitly to object of type button
   Button senderButton = ( Button ) sender;

   // get text from clicked Button, which stores account type
   string accountType = senderButton.Text;

   // read and display file information
   try
   {
      // go back to the beginning of the file
      input.Seek( 0, SeekOrigin.Begin );

      displayTextBox.Text = "The accounts are:\r\n";
```

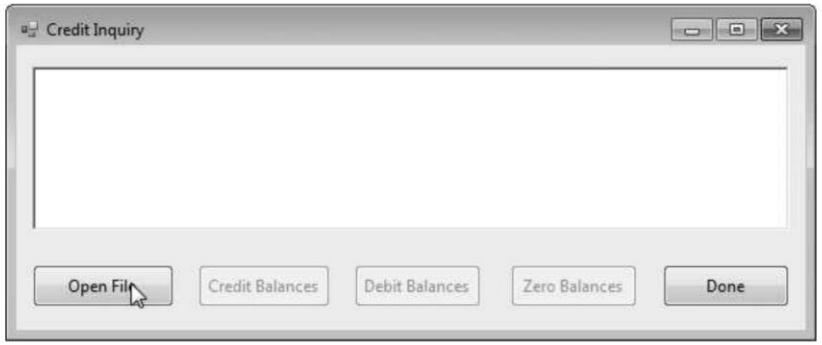
```
// traverse file until end of file
while ( true )
{
   string[] inputFields; // stores individual pieces of data
   Record record; // store each Record as file is read
   decimal balance; // store each Record's balance

   // get next Record available in file
   string inputRecord = fileReader.ReadLine();

   // when at the end of file, exit method
   if ( inputRecord == null )
      return;

   inputFields = inputRecord.Split( ',' ); // parse input

   // create Record from input
   record = new Record(
      Convert.ToInt32( inputFields[ 0 ] ), inputFields[ 1 ],
      inputFields[ 2 ], Convert.ToDecimal( inputFields[ 3 ] ) );

   // store record's last field in balance
   balance = record.Balance;
```

```
            // determine whether to display balance
            if ( ShouldDisplay( balance, accountType ) )
            {
               // display record
               string output = record.Account + "\t" +
                  record.FirstName + "\t" + record.LastName + "\t";

               // display balance with correct monetary format
               output += String.Format( "{0:F}", balance ) + "\n";

               // copy output to screen
               displayTextBox.AppendText( output );
            } // end if
         } // end while
      } // end try
      // handle exception when file cannot be read
      catch ( IOException )
      {
         MessageBox.Show( "Cannot Read File", "Error",
            MessageBoxButtons.OK, MessageBoxIcon.Error );
      } // end catch
   } // end method getBalances_Click
```

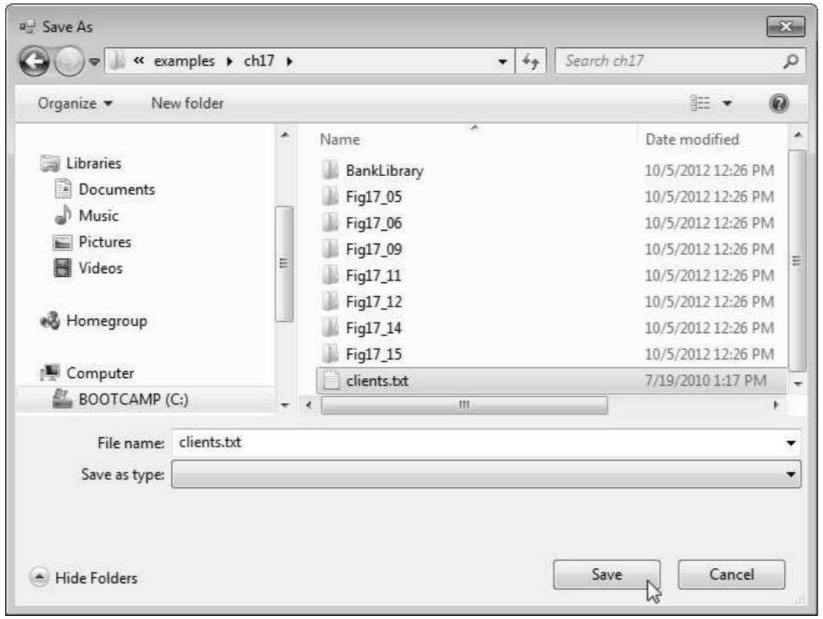
```
// determine whether to display given record
      private bool ShouldDisplay( decimal balance, string accountType )
      {
         if ( balance > 0M )
         {
            if ( accountType == "Credit Balances" )
               return true;
         } // end if
         else if ( balance < 0M )
         {
            if ( accountType == "Debit Balances" )
               return true;
         } // end else if
         else // balance == 0
         {
            if ( accountType == "Zero Balances" )
               return true;
         } // end else

         return false;
      } // end method ShouldDisplay
```

```
// invoked when user clicks Done button
      private void doneButton_Click( object sender, EventArgs e )
      {
         if ( input != null )
         {
            // close file and StreamReader
            try
            {
               // close StreamReader and underlying file
               fileReader.Close();
            } // end try
            // handle exception if FileStream does not exist
            catch ( IOException )
            {
               // notify user of error closing file
               MessageBox.Show( "Cannot close file", "Error",
                  MessageBoxButtons.OK, MessageBoxIcon.Error );
            } // end catch
         } // end if

         Application.Exit();
      } // end method doneButton_Click
```

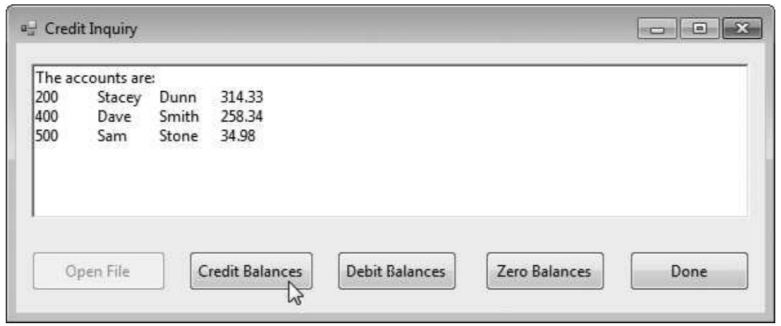
a) GUI when the app first executes



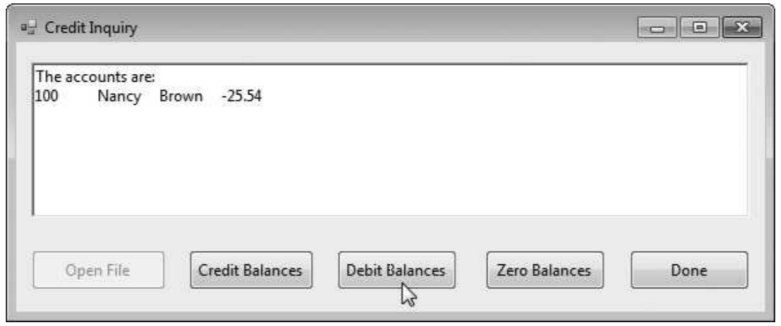

b) Opening the `clients.txt` file



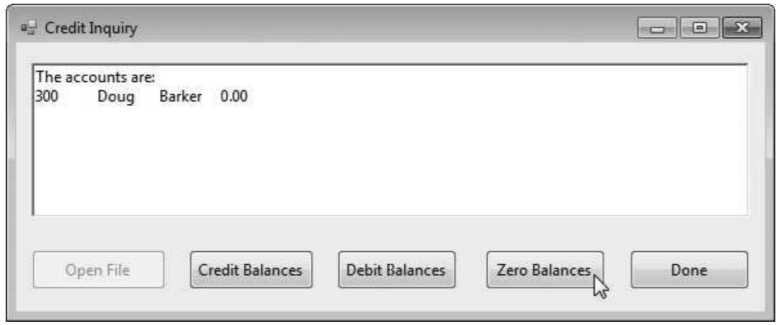

c) Displaying accounts with credit balances




d) Displaying accounts with debit balances




e) Displaying accounts with zero balances



# Serialization

C# provides such a mechanism, called object serialization. A serialized object is an object represented as a sequence of bytes that includes the object's data, as well as information about the object's type and the types of data stored in the object. After a serialized object has been written to a file, it can be read from the file and deserialized—that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

# Serialization

Continue

Class BinaryFormatter (namespace System.Runtime.Serialization.Formatters.Binary) enables entire objects to be written to or read from a stream.

BinaryFormatter method Serialize writes an object's representation to a file. BinaryFormatter method Deserialize reads this representation from a file and reconstructs the original object.

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

Both methods require a Stream object (e.g., the FileStream) as a parameter so that the BinaryFormatter can access the correct stream.

Object serialization is performed with byte-based streams and binary files are not human readable.

```
// Fig. 17.13: RecordSerializable.cs
// Serializable class that represents a data record.
using System;

namespace BankLibrary
{
   [Serializable]
   public class RecordSerializable
   {
      public int Account { get; set; }
      public string FirstName { get; set; }
      public string LastName { get; set; }
      public decimal Balance { get; set; }
      public RecordSerializable() : this( 0, string.Empty, string.Empty, 0M ){}
      public RecordSerializable( int accountValue, string firstNameValue,string lastNameValue,
                                 decimal balanceValue )
      {
         Account = accountValue;
         FirstName = firstNameValue;
         LastName = lastNameValue;
         Balance = balanceValue;
      } // end constructor
   } // end class RecordSerializable
} // end namespace BankLibrary
```

# Creating a Sequential-Access File Using Object Serialization

In a class that's marked with the [Serializable] attribute or that implements interface ISerializable, you must ensure that every instance variable of the class is also serializable.

All simple-type variables and strings are serializable.

For variables of reference types, you must check the class declaration (and possibly its base classes) to ensure that the type is serializable. By default, array objects are serializable.

```
// Fig 17.14: CreateFileForm.cs
// Creating a sequential-access file using serialization.
using System;
using System.Windows.Forms;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization;
using BankLibrary;

namespace CreateFile
{
   public partial class CreateFileForm : BankUIForm
   {
      // object for serializing RecordSerializables in binary format
      private BinaryFormatter formatter = new BinaryFormatter();
      private FileStream output; // stream for writing to a file

      // parameterless constructor
      public CreateFileForm()
      {
         InitializeComponent();
      } // end constructor
```

```
// handler for saveButton_Click
private void saveButton_Click( object sender, EventArgs e )
{
   // create and show dialog box enabling user to save file
   DialogResult result;
   string fileName; // name of file to save data

   SaveFileDialog fileChooser = new SaveFileDialog();
   fileChooser.CheckFileExists = false; // let user create file
   // retrieve the result of the dialog box
   result = fileChooser.ShowDialog();
   fileName = fileChooser.FileName; // get specified file name
```

```
// ensure that user clicked "OK"
         if ( result == DialogResult.OK )
         {
               if ( fileName == string.Empty )
                  MessageBox.Show( "Invalid File Name", "Error",MessageBoxButtons.OK, MessageBoxIcon.Error );
            else
            {
               try
               {
                  // open file with write access
                  output = new FileStream( fileName,
                     FileMode.OpenOrCreate, FileAccess.Write );

                  // disable Save button and enable Enter button
                  saveButton.Enabled = false;
                  enterButton.Enabled = true;
               } // end try
               // handle exception if there is a problem opening the file
               catch ( IOException )
               {
                  // notify user if file could not be opened
                  MessageBox.Show( "Error opening file", "Error",
                     MessageBoxButtons.OK, MessageBoxIcon.Error );
               } // end catch
            } // end else
         } // end if
      } // end method saveButton_Click
```

```
// handler for enterButton Click
      private void enterButton_Click( object sender, EventArgs e )
      {
         // store TextBox values string array
         string[] values = GetTextBoxValues();

         // RecordSerializable containing TextBox values to serialize
         RecordSerializable record = new RecordSerializable();

         // determine whether TextBox account field is empty
         if ( values[ ( int ) TextBoxIndices.ACCOUNT ] != string.Empty )
         {
            // store TextBox values in RecordSerializable and serialize it
            try
            {
               // get account number value from TextBox
               int accountNumber = Int32.Parse(
                  values[ ( int ) TextBoxIndices.ACCOUNT ] );
```

```
      // determine whether accountNumber is valid
      if ( accountNumber > 0 )
      {
         // store TextBox fields in RecordSerializable
         record.Account = accountNumber;
         record.FirstName = values[ ( int )
            TextBoxIndices.FIRST ];
         record.LastName = values[ ( int )
            TextBoxIndices.LAST ];
         record.Balance = Decimal.Parse( values[
            ( int ) TextBoxIndices.BALANCE ] );

         // write Record to FileStream ( serialize object )
         formatter.Serialize( output, record );
      } // end if
      else
      {
         // notify user if invalid account number
         MessageBox.Show( "Invalid Account Number", "Error",
            MessageBoxButtons.OK, MessageBoxIcon.Error );
      } // end else
   } // end try
```

```
      // notify user if error occurs in serialization
      catch ( SerializationException )
      {
         MessageBox.Show( "Error Writing to File", "Error",
            MessageBoxButtons.OK, MessageBoxIcon.Error );
      } // end catch
   } // end if

   ClearTextBoxes(); // clear TextBox values
} // end method enterButton_Click
```

```
 // handler for exitButton Click
private void exitButton_Click( object sender, EventArgs e )
{
   // determine whether file exists
   if ( output != null )
   {
      // close file
      try
      {
         output.Close(); // close FileStream
      } // end try
      // notify user of error closing file
      catch ( IOException )
      {
         MessageBox.Show( "Cannot close file", "Error",
            MessageBoxButtons.OK, MessageBoxIcon.Error );
      } // end catch
   } // end if

   Application.Exit();
} // end method exitButton_Click
```

# Reading and Deserializing Data from a Binary File

The preceding section showed how to create a sequential-access file using object serialization. In this section, we discuss how to read serialized objects sequentially from a file.

Figure 17.15 reads and displays the contents of the clients.ser file created by the program in Fig. 17.14. The sample screen captures are identical to those of Fig. 17.11, so they are not shown here. Line 15 creates the BinaryFormatter that will be used to read objects. The program opens the file for input by creating a FileStream object (lines 49–50). The name of the file to open is specified as the first argument to the FileStream constructor.

# Reading and Deserializing Data from a Binary File

Continue

The program reads objects from a file in event handler nextButton_Click. We use method Deserialize (of the BinaryFormatter created in line 15) to read the data (lines 65–66). Note that we cast the result of Deserialize to type RecordSerializable this cast is necessary, because Deserialize returns a reference of type object and we need to access properties that belong to class RecordSerializable. If an error occurs during deserialization or the end of the file is reached, a SerializationException is thrown, and the FileStream object is closed.

```
// Fig. 17.15: ReadSequentialAccessFileForm.cs
// Reading a sequential-access file using deserialization.
using System;
using System.Windows.Forms;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization;
using BankLibrary;

namespace ReadSequentialAccessFile
{
   public partial class ReadSequentialAccessFileForm : BankUIForm
   {
      private BinaryFormatter reader = new BinaryFormatter();
      private FileStream input; // stream for reading from a file
      public ReadSequentialAccessFileForm()
      {
         InitializeComponent();
      } // end constructor
```

```
// invoked when user clicks the Open button
      private void openButton_Click( object sender, EventArgs e )
      {
         // create and show dialog box enabling user to open file
         DialogResult result; // result of OpenFileDialog
         string fileName; // name of file containing data

         OpenFileDialog fileChooser = new OpenFileDialog();
         result = fileChooser.ShowDialog();
         fileName = fileChooser.FileName; // get specified name
```

```
// ensure that user clicked "OK"
  if ( result == DialogResult.OK )
  {
     ClearTextBoxes();

     // show error if user specified invalid file
     if ( fileName == string.Empty )
        MessageBox.Show( "Invalid File Name", "Error",
           MessageBoxButtons.OK, MessageBoxIcon.Error );
     else
     {
        // create FileStream to obtain read access to file
        input = new FileStream(
           fileName, FileMode.Open, FileAccess.Read );

        openButton.Enabled = false; // disable Open File button
        nextButton.Enabled = true;  // enable Next Record button
     } // end else
  } // end if
} // end method openButton_Click
```

```
 // invoked when user clicks Next button
private void nextButton_Click( object sender, EventArgs e )
{
   // deserialize RecordSerializable and store data in TextBoxes
   try
   {
      // get next RecordSerializable available in file
      RecordSerializable record =
         ( RecordSerializable ) reader.Deserialize( input );

      // store RecordSerializable values in temporary string array
      string[] values = new string[] {
         record.Account.ToString(),
         record.FirstName.ToString(),
         record.LastName.ToString(),
         record.Balance.ToString()
      };

      // copy string array values to TextBox values
      SetTextBoxValues( values );
   } // end try
```

```
    // handle exception when there are no RecordSerializables in file
     catch ( SerializationException )
     {
        input.Close(); // close FileStream
        openButton.Enabled = true; // enable Open File button
        nextButton.Enabled = false; // disable Next Record button

        ClearTextBoxes();

        // notify user if no RecordSerializables in file
        MessageBox.Show( "No more records in file", string.Empty,
           MessageBoxButtons.OK, MessageBoxIcon.Information );
     } // end catch
  } // end method nextButton_Click
```