

Chapter 9 introduced sequential file operations. The files you worked with in that chapter were text files, which means that data is stored in the files as text. Even the numbers that you store in a text file are converted to text. When you read an item from a text file, it is read as a string. Sometimes text files are convenient to work with because you can open them in programs such as Notepad.

Visual Basic also lets you create binary files. Data stored in a binary file is stored in its raw binary format. Binary files are sometimes smaller than text files, and the data that is read from them does not have to be converted before math or other operations are performed on them. Although you might be able to open a binary file in Notepad, you cannot read its contents because the file is not formatted as text.

To open a binary file, first you create a `FileStream` object, and then you create a `BinaryWriter` or `BinaryReader` object. For example, the following code creates a new binary file named *myfile.dat*. A `BinaryWriter` object can then be used to write binary data to the file.

```
Dim outputFile As System.IO.BinaryWriter
Dim fs As System.IO.FileStream
fs = System.IO.File.Open("myfile.dat", System.IO.FileMode.Create)
outputFile = New System.IO.BinaryWriter(fs)
```

The `System.IO.File.Open` method in this code takes two arguments. The first argument is the pathname of the file. The second argument specifies the mode in which the file will be opened. Valid modes are `FileMode.Append`, `FileMode.Create`, `FileMode.CreateNew`, `FileMode.Open`, `FileMode.OpenOrCreate`, and `FileMode.Truncate`. You can view detailed descriptions of each of these modes by searching for *FileMode Enumeration* in the *Visual Basic Help* system. To avoid having to repeat the `System.IO` namespace before each `FileMode`, use the following `Imports` statement: `Imports System.IO`

The `BinaryWriter` class has a `Write` method that writes data in binary form to a file. The following code shows how an integer is written to a file. Assume that `age` is an integer variable.

```
age = CInt(TextBox("Enter your age."))
outputFile.Write(age)
```

To close a binary file, use the `Close` method, as shown in the following statement:

```
outputFile.Close()
```

Reading data from a binary file is also straightforward. The following code shows how a binary file may be opened, an integer value read from it, and then closed.

```
Dim inputFile As System.IO.BinaryReader
Dim fs As System.IO.FileStream
Dim age As Integer
fs = System.IO.File.Open("myfile.dat", SystemIO.FileMode.Open)
inputFile = New System.IO.BinaryReader(fs)
age = inputFile.ReadInt32()
inputFile.Close()
```

Table C-1 lists some of the `BinaryReader` methods for reading data from a file.

Table C-1 Some of the `BinaryReader` methods for reading data

Method	Description
<code>PeekChar</code>	Reads the next available item as a character from the file and does not advance the read position. Returns <code>-1</code> if there are no more items in the file.
<code>Read</code>	Reads the next character from the file and advances the read position to the next item in the file.
<code>ReadBoolean</code>	Reads a Boolean value from the file and advances the read position to the next item in the file.
<code>ReadByte</code>	Reads a Byte value from the file and advances the read position to the next item in the file.
<code>ReadBytes</code>	Reads an array of Byte values from the file and advances the read position to the next item in the file.
<code>ReadChar</code>	Reads a Char value from the file and advances the read position to the next item in the file.
<code>ReadChars</code>	Reads an array of Char values from the file and advances the read position to the next item in the file.
<code>ReadDecimal</code>	Reads a Decimal value from the file and advances the read position to the next item in the file.
<code>ReadDouble</code>	Reads a Double value from the file and advances the read position to the next item in the file.
<code>ReadInt16</code>	Reads a Short value from the file and advances the read position to the next item in the file.
<code>ReadInt32</code>	Reads an Integer value from the file and advances the read position to the next item in the file.
<code>ReadInt64</code>	Reads a Long value from the file and advances the read position to the next item in the file.
<code>ReadSingle</code>	Reads a Single value from the file and advances the read position to the next item in the file.

Random-Access Files

Random-access files provide an efficient way to store and retrieve data so that records may be modified selectively. Sequential files may only be processed in sequential order, hence the name sequential. Processing begins at the first item and continues in sequence to the last item, unless the process is halted and the file closed. A random-access file, on the other hand, does not have to be processed in sequence. You can select any item in the file, read it, and modify it in place.

Fields, Records, and Structures

In Chapter 10 you learned that a field is an individual piece of data pertaining to a single item, and a record is a collection of related fields. For example, a set of fields might be a person's name, age, address, and phone number. Together, those fields that pertain to one person make up a record.

Random-access files are designed to hold data organized in records. Each record in a random-access file is identified by a unique integer. The first record is record 1, the second record is record 2, and so on. The records in a random-access file must be the same length, for a very good reason: The physical position of any record can be found by multiplying the record number by the record length.

Structures provide a convenient way to organize information into records. For example, the following structure declaration could be used to create a record.

```
Structure PersonData
    Public Name As String
    Public Age As Integer
    Public Address As String
    Public City As String
    Public State As String
    Public PostalCode As String
    Public Telephone as String
End Structure
```

A problem arises when you store strings in a random-access file because strings are variable lengths. To declare a string with a fixed length, prefix its declaration with the following attribute.

```
<VBFixedString(Length)>
```

Length is the fixed length of the string. The following structure declaration uses this attribute and it is suitable for holding data that will be written to a random-access file.

```
Structure PersonData
    <VBFixedString(25)> Public Name As String
    Public Age As Integer
    <VBFixedString(25)> Public Address As String
    <VBFixedString(15)> Public City As String
    <VBFixedString(2)> Public State As String
    <VBFixedString(10)> Public PostalCode As String
    <VBFixedString(15)> Public Telephone as String
End Structure
```

Opening a Random-Access File

The `FileOpen` method is used to open a random-access file. The method's general format is

```
FileOpen (FileNumber, FileName, OpenMode.Random, _
    AccessMode, OpenShare.Default, RecordLength)
```

FileNumber is an integer that you use to identify the file. *FileName* is the name of the file, which can include the file's path. *AccessMode* specifies how the file will be accessed, and can be one of these values: `OpenAccess.Default`, `OpenAccess.Read`, `OpenAccess.ReadWrite`, or `OpenAccess.Write`. Note that a random-access file can be opened for both reading and writing with `OpenAccess.ReadWrite`. *RecordLength* is the size of the records that will be stored in the file.

You can determine the size of a record with the `Len` function. The general format is

```
Len(Object)
```

Object is any variable or object. The function returns the size of *Object*.

For example, the following code declares `person` as a `PersonData` structure variable. It passes `person` to a Sub procedure, `GetData`, which stores data in `person`. Then it opens a random-access file named `CustomerData.dat`. The `Len` function is used to determine the length of the `person` variable.

```
Dim person As PersonData
GetData(person)
FileOpen(1, "CustomerData.dat", OpenMode.Random, _
    OpenAccess.ReadWrite, OpenShare.Default, Len(person))
```

Writing Records to a Random-Access File

You use the `FilePut` method to write a record to a random-access file. The general format is

```
FilePut(FileNumber, Item, RecordNumber)
```

FileNumber is the number of the file. This is the same number you specified as the *FileNumber* argument with the `FileOpen` method. *Item* is the record to be written. *RecordNumber* is the number of the record to write. For example, the following statement writes the `person` variable to file number 1 as record number 1.

```
FilePut(1, person, 1)
```

Reading Records from a Random-Access File

You use the `FileGet` method to read a record from a random-access file. The general format is

```
FileGet (FileNumber, Item, RecordNumber)
```

FileNumber is the number of the file. This is the same number you specified as the *FileNumber* argument with the `FileOpen` method. *Item* is a variable to hold the record being read. *RecordNumber* is the number of the record to read. For example, the following statement reads record number 5 and stores it in the `person` variable.

```
FileGet(1, person, 5)
```

Testing for the End of a Random-Access File

The `EOF` function determines if the end of a random-access file has been reached. The general format is

```
EOF(FileNumber)
```

FileNumber is the number of the file to test. The function returns `True` if the end of the file has already been reached, or `False` otherwise. For example, the following loop reads

each record from a file into the `person` variable, and passes `person` to a function named `DisplayData`.

```
recNum = 1
Do Until EOF(1)
    FileGet(1, person, recNum)
    DisplayData(person)
    recNum += 1
Loop
```

Closing a Random-Access File

Use the `FileClose` method to close a random-access file. The general format is

```
FileClose(FileNumber)
```

FileNumber is the file number associated with the file.