Bonus Reference

VB.NET Functions and Statements

THIS BONUS REFERENCE DESCRIBES the functions and statements that are supported by Visual Basic .NET, grouped by category. When you're searching for the statement to open a file, you probably want to locate all file I/O commands in one place. This is exactly how this reference is organized. Moreover, by grouping all related functions and statements in one place, I can present examples that combine more than one function or statement.

The majority of the functions are the same as in VB6. One difference is that many of the VB6 statements are implemented as functions in VB.NET. Moreover, many VB6 functions have an equivalent method in a Framework class. VB programmers are so accustomed to the old functions that they will not consider the alternatives—at least for a while. The Len() function of VB6 returns the length of a string. In VB.NET you can retrieve the length of a string with the Length method of a string variable. If *strVar* is declared as string variable, you can retrieve its length by calling the Length method:

```
Dim strVar As String = "a short string"
Console.WriteLine("The string contains " & strVar.Length & " characters")
```

Or you can call the Len() function passing the name of the string as argument:

```
Dim strVar As String = "a short string"
Console.WriteLine("The string contains " & Len(strVar) & " characters")
```

Most of the built-in functions are VB6 functions, and they accept optional arguments. VB.NET uses overloaded forms of the same function, and this is an important difference you have to keep in mind as you work with the built-in functions. If you omit an optional argument, you must still insert the comma to indicate that an argument is missing. Optional arguments are enclosed in square brackets. The Mid() function, for example, extracts a number of characters from a string, and its syntax is

```
newString = Mid(string[, start][, length])
```

The starting location of the characters to be extracted is specified by the *start* argument, and the number of characters to be extracted is *length*. If you omit the *start* argument, the extraction starts with the first character in the string. If you omit the *length* argument, all the characters from the

specified position to the end of the string are extracted. The only mandatory argument is the first one, which is the string from which the characters will be extracted, and this argument can't be omitted.

The methods of the various classes are discussed in detail in the book. This bonus reference contains all the functions supported by VB.NET, and these functions are listed by category in Table 1. Items in the table that are not followed by parentheses are statements and are also described in this reference.

TABLE 1: VB.NET FUNCTIONS BY TYPE

TYPE	FUNCTIONS
Input/Output	InputBox(), MsgBox()
File and Folder Manipulation	ChDir(), ChDrive(), CurDir(), Dir(), FileCopy(), FileDateTime(), FileLen, GetAttr(), Kill, MkDir(), Rename(), RmDir(), SetAttr()
Data Type Identification	IsArray(), IsDate(), IsDBNull(), IsNothing() IsNumeric(), IsReference, TypeName(), VarType()
Variable Type Conversion	CBool(), CByte(), CChar(), CDate(), CDbl(), CDec(), CInt(), CLng(), CObj(), CShort(), CSng(), CStr(), CType()
String Manipulation	Asc(), AscW(), Chr(), ChrW(), Filter(), InStr(), InStrRev(), Join(), LCase(), Left(), Len(), LTrim(), Mid(), Mid, Replace(), Right(), RTrim(), Space(), Split(), StrComp(), StrConv(), StrDup(), StrReverse(), Trim(), UCase()
Data Formatting	Format(), FormatCurrency(), FormatDateTime(), FormatNumber(), FormatPercent(), LSet(), RSet(), Str(), Val()
Math	Abs(), Atan(), Cos(), Exp(), Fix(), Hex(), Int(), Log(), Oct(), Pow(), Round(), Sin(), Sqrt(), Tan()
Date and Time	DateAdd(), DateDiff(), DatePart(), DateSerial(), DateValue(), Day(), Hour(), Minute(), Month(), MonthName(), Now(), Second(), TimeSerial(), TimeValue(), Weekday(), WeekdayName(), Year()
Financial	DDB(), FV(), IPmt(), IRR(), MIRR(), NPer(), NPV(), Pmt(), PPmt(), PV(), Rate(), SLN(), SYD()
File I/O	EOF(), FileAttr(), FileClose(), FileOpen(), FileGet(), FilePut(), FreeFile(), Input(), LineInput(), Loc(), Lock(), LOF(), Print(), PrintLine(), Reset(), Seek(), Unlock(), Width(), Write(), WriteLine()
Random Numbers	Rnd(), Randomize
Graphics	QBColor(), RGB()
Registry	DeleteSetting(), GetAllSettings(), GetSetting(), SaveSetting()
Application Collaboration	AppActivate(), Shell()
Miscellaneous	Beep, CallByName(), Choose(), Environ(), Iif(), Option, Switch()

These functions and statements are described in the following sections, along with examples. The entries are not discussed alphabetically within each category. I start with the simpler ones so that I can present examples that combine more than one function and/or statement.

Input/Output

Visual Basic provides two basic functions for displaying (or requesting) information to the user: MsgBox() and InputBox(). Windows applications should communicate with the user via nicely designed forms, but the MsgBox() and InputBox() functions are still around and quite useful.

InputBox(prompt[, title][, default][, xpos][, ypos])

The InputBox() function displays a dialog box with a prompt and a TextBox control and waits for the user to enter some text and click the OK or Cancel button. The arguments of the InputBox() function are shown in Table 2.

TABLE 2: ARGUMENTS OF THE INPUTBOX() FUNCTION

ARGUMENT	WHAT IT IS	DESCRIPTION
prompt	The prompt that appears in the dialog box	If necessary, the prompt is broken into multiple lines automatically. To control line breaks from within your code, use a carriage return character or a linefeed character (vbCr, vbLf).
title	The title of the dialog box	If you omit this argument, the application's name is displayed as the title.
default	The default input (if any)	If you anticipate the user's response, use this argument to display it when the dialog box is first opened.
xpos, ypos	The coordinates of the top-left corner of the dialog box	Expressed in twips.

The simplest format of the InputBox() function is as follows:

```
SSN = InputBox("Please enter your social security number")
```

The string that the user enters in the dialog box is assigned to the variable *SSN*. The return value is always a string, even if the user enters numeric information. When prompting for input with the InputBox() function, always check the value returned by the function. At the very least, check for a blank string. Use the IsNumeric() function if you expect the user to enter a number, use the IsDate() function if you expect the user to enter a date, and so on.

```
BDay = InputBox("Please enter your birth date")
If IsDate(BDay) Then
   MsgBox("Preparing your horoscope")
Else
   MsgBox("Please try again with a valid birth date")
End If
```

MsgBox(prompt[, buttons][, title])

The MsgBox() function displays a dialog box with a message and waits for the user to close it by clicking a button. The message is the first argument (*prompt*). The simplest form of the MsgBox() function is as follows:

```
MsgBox("Your computer is running out of memory!")
```

This function displays a message in a dialog box that has an OK button. The MsgBox() function can display other buttons and/or an icon in the dialog box and return a numeric value, depending on which button was clicked. Table 3 summarizes the values for the *buttons* argument.

Table 3: The MsgBoxStyle Enumeration

Constant	Value	Description
Button Values		
OKOnly	0	Displays OK button only.
OKCancel	1	Displays OK and Cancel buttons.
AbortRetryIgnore	2	Displays Abort, Retry, and Ignore buttons.
YesNoCancel	3	Displays Yes, No, and Cancel buttons.
YesNo	4	Displays Yes and No buttons.
RetryCancel	5	Displays Retry and Cancel buttons.
Icon Values		
Critical	16	Displays Critical Message icon.
Question	32	Displays Warning Query icon.
Exclamation	48	Displays Warning Message icon.
Information	64	Displays Information Message icon.
Default Button		
DefaultButton1	0	First button is default.
DefaultButton2	256	Second button is default.
DefaultButton3	512	Third button is default.
DefaultButton4	768	Fourth button is default.
Modality		
ApplicationModal	0	The user must respond to the message box before switching to any of the Forms of the current application.
SystemModal	4096	All applications are suspended until the user responds to the message box.

Button values determine which buttons appear in the dialog box. Notice that you can't choose which individual buttons to display; you can only choose groups of buttons.

Icon values determine an optional icon you can display in the dialog box. These are the common icons used throughout the Windows user interface to notify the user about an unusual or exceptional event.

Default button values determine which button is the default one; pressing Enter activates this button. The constants `ApplicationModal` and `SystemModal` determine whether the message box is modal.

To combine any of these settings into a single value, simply add their values.

Finally, the MsgBox() function returns an integer, which indicates the button pressed, according to Table 4.

TABLE 4: THE MSGBOXRESULT ENUMERATION

CONSTANT	VALUE
`OK`	1
`Cancel`	2
`Abort`	3
`Retry`	4
`Ignore`	5
`Yes`	6
`No`	7

To display a dialog box with the OK and Cancel buttons and the Warning Message icon, add the values `MsgBoxStyle.Exclamation` and `MsgBoxStyle.OKCancel` as follows:

```
cont = MsgBox("This operation may take several minutes", _
              MsgBoxStyle.Exclamation + MsgBoxStyle.OKCancel)
```

The value returned by the MsgBox() function is a member of the MsgBoxResult enumeration, which is shown in Table 4. Your program continues with the operation if the value of *cont* is `MsgBoxResult.OK`.

To display a dialog box with the Yes and No buttons and the Critical Message icon, add the values 4 and 16 as follows:

```
cont = MsgBox("Incomplete data. Would you like to retry?", _
              MsgBoxStyle.YesNo + MsgBoxStyle.Critical)
If cont = MsgBoxResult.Yes Then     ' user clicked Yes
   { prompt again }
Else                                ' user clicked No
   { exit procedure }
Endif
```

File and Folder Manipulation

The following Visual Basic functions manipulate files and folders (move and rename files, create new folders and delete existing ones, and so on). The functions discussed in this section do not manipulate the contents of the files. Most of them are equivalent to the members of the File and Directory objects, discussed in Chapter 13. They are also equivalent to the basic DOS commands for manipulating files and folders.

GetAttr(pathname)

This function returns an integer (a member of the FileAttribute enumeration) representing the attributes of a file, directory, or folder, according to Table 5.

TABLE 5: THE FILEATTRIBUTE ENUMERATION

CONSTANT	VALUE	ATTRIBUTE
Normal	0	Normal
ReadOnly	1	Read-only
Hidden	2	Hidden
System	4	System
Volume	8	Volume label
Directory	16	Directory or folder
Archive	32	File has changed since last backup

To determine which attributes are set, use the AND operator to perform a bitwise comparison of the value returned by the GetAttr() function and the value of one or more attributes. If the result is not zero, that attribute is set for the named file. For example, to find out if a file is read-only, use a statement such as the following:

```
Result = GetAttr(FName) And FileAttribute.ReadOnly
```

If the file *Fname* has its read-only attribute set, *Result* will be 1. If not, *Result* will be 0, regardless of the values of any other attributes. To find out whether a file has its archive attribute set, then the statement

```
Result = GetAttr(FName) And FileAttribute.Archive
```

will assign the value 32 to the *Result* variable.

If the file has both its archive and read-only attributes set, the GetAttr() function will return the value 33. However, you must AND this value with the appropriate constant to find out whether a certain attribute is set.

SetAttr(pathname, attributes)

The SetAttr() function sets the attributes of a file. The argument *pathname* is the path name of an existing file and attributes is a numeric value that specifies one or more attributes. To set an attribute without affecting any existing attributes, use a statement like

```
SetAttr(pathname, GetAttr(file_name) Or new_attribute)
```

The *new_attribute* argument can have any of the values shown in Table 5 in the preceding section.

To change multiple attributes, combine the corresponding values with the logical OR operator. Notice that the statement

```
SetAttr(file_name, new_attribute)
```

will turn on a specific attribute, but it will clear all other attributes. If a file is read-only and hidden, its Attributes property is 3 (1 + 2 according to Table 5). If you attempt to turn on the Archive attribute by setting its Attributes property to 32, the other two attributes will be cleared. By combining the new attribute (32) and the existing attributes with the OR operator, the file will be read-only, hidden, and archive.

To remove a specific attribute, first find out whether this attribute is already set, and then subtract its value from the value returned by the GetAttr() function. To remove the Hidden attribute, use a structure such as the following:

```
If GetAttr(file_name) And FileAttribute.Hidden Then
   SetAttr(file_name, GetAttr(file_name) – FileAttribute.Hidden)
End If
```

You can also use the MsgBox() function to prompt the user to change the read-only attribute:

```
If GetAttr(file_name) And FileAttribute.ReadOnly Then
   reply = MsgBox("This is a read-only file. Delete it anyway?", _
                  MsgBoxStyle.YesNo)
   If reply = MsgBoxResult.Yes Then
      SetAttr(file_name, GetAttr(file_name) - FileAttribute.ReadOnly)
      Kill(file_name)
   End If
Else
   Kill(file_name)
End If
```

You can also use the XOR operator to reset an attribute. The call to the SetAttr() function can also be written as follows (the two methods of resetting an attribute are equivalent):

```
SetAttr(file_name, GetAttr(file_name) Xor FileAttribute.ReadOnly)
```

Kill(pathname)

The Kill() function deletes the specified file permanently from the hard disk. The argument *pathname* specifies one or more file names to be deleted. The Kill() function supports the use of

multiple-character (*) and single-character (?) wildcards to specify multiple files. If the specified file does not exist, a runtime error is generated. The Kill() function is frequently used as follows:

```
Try
   Kill("C:\RESUME.OLD")
Catch exc As Exception
End Try
```

The error handler prevents the runtime error that would occur if the specified file doesn't exist or can't be deleted, and the program continues with the execution of the following statement.

The Kill() function does not move the specified file to the Recycle Bin; it permanently deletes the file from the disk. If you move the file to the Recycle Bin with the FileCopy() function, the file will appear in the Recycle Bin's window, but you won't be able to restore it.

FileDateTime(pathname)

This function returns the date and time when a file was created or last modified. The following statement:

```
Console.WriteLine(FileDateTime("myDocument.txt"))
```

returns a date/time value such as "21/11/01 14:13:02 PM".

FileLen(pathname)

The FileLen() function returns a long integer value indicating the file's length in bytes. The file whose length you want to find out is passed as an argument to the function. The statement

```
MsgBox("The file contains" & FileLen(".\docs\myDocument.txt") & " bytes")
```

displays the length of the specified file in a message box.

The FileLen() function is different from the LOF() function, which returns the length of a file that has already been opened. See the description of the LOF() function in the section "File I/O."

MkDir(path)

The MkDir() function creates a new folder (directory). The *path* argument can be the full path of the new folder, or just a folder name, in which case a new folder is created under the current folder. The statement:

```
MkDir("C:\Users\New User")
```

will create the `New User` folder under `C:\Users`, but only if the parent folder exists already. If `C:\Users` doesn't already exist, you must call the MkDir() function twice, to create two folders, as shown next:

```
MkDir("C:\Users")
MkDir("C:\Users\New User")
```

Alternatively, you can switch to the parent folder and then create the subfolder:

```
ChDrive("C:\")
ChDir("C:\")
```

```
MkDir("Users")
ChDir("Users")
MkDir("New User")
```

You should also use the appropriate error-trapping code, because if a folder you attempt to create exists already, a runtime error will occur.

RmDir(path)

The RmDir() function deletes a folder (directory), specified by the path argument. The argument can't contain wildcard characters (in other words, you can't remove multiple folders with a single call to the RmDir() function). Moreover, the folder must be empty; if not, a runtime error will occur. To remove a folder containing files, use the Kill() function to delete the files first. In addition, you must remove all subfolders of a given folder before you can remove the parent folder.

The statement:

```
RmDir("C:\Users")
```

will generate an error message. You must first remove the subfolder, then the parent folder:

```
RmDir("C:\Users\New User")
RmDir("C:\Users")
```

ChDir(path)

The ChDir() function changes the current folder (directory). If your application opens many disk files, you can either specify the path name of each file, or switch to the folder where the files reside and use their names only.

To switch to the folder `C:\Windows`, use the statement:

```
ChDir("C:\Windows")
```

If the argument of the ChDir() function doesn't include a drive name, then ChDir() will attempt to switch to the specified folder on the current drive. If no such folder exists, then the current folder won't change and a runtime error will be raised.

The ChDir() function changes the current folder but not the current drive. For example, if the current drive is C:, the following statement changes the *current folder* to another folder on drive D:, but C: remains the *current drive:*

```
ChDir "D:\TMP"
```

To change the current drive, use the ChDrive() function, described next.

You can also use relative folder names. The statement

```
ChDir("..")
```

takes you to the parent folder of the current folder, while the statement

```
ChDir("..\MyFiles")
```

takes you to the `MyFiles` folder of the parent folder (both the current folder and `MyFiles` are subfolders of the same folder).

ChDrive(drive)

The ChDrive() function changes the current drive. The *drive* argument must be the name of an existing drive. If the drive argument is a multiple-character string, ChDrive() uses only the first letter.

CurDir([drive])

The CurDir() function, when called without an argument, returns the name of the current folder in the current drive. To find out the current folder on another drive, supply the drive's name as argument. If you're in a folder of the C: drive, the function

```
CDir = CurDir()
```

returns the current folder on the current drive. To find out the current folder on drive D:, call the function CurDir() as follows:

```
DDir = CurDir("D")
```

Dir([pathname[, attributes]])

The Dir() function accepts two optional arguments and returns a string with the name of a file or folder that matches the specified pathname or file attribute(s).

If you specify the first argument, which supports wildcard characters, Dir() will return the name of the file or folder that matches the specification. If no file or folder matched the specification, an empty string is returned (""). The second argument is a numeric value, which specifies one or more attributes, from the enumeration shown in Table 5 earlier in this reference. If the second argument is omitted, only normal files (files without attributes) are returned.

A common use of the Dir() function is to check whether a specific file or folder exists. The statement:

```
OCXFile = Dir("C:\WINDOWS\SYSTEM\MSCOMCTL.OCX")
```

will return "MSCOMCTL.OCX" if the specified file exists, an empty string otherwise.

To find out how many DLL files exist in your `WinNT\System` folder, you must specify a wildcard specification and call the Dir() function repeatedly:

```
Dim DLLFile As String
Dim DLLFiles As Integer
DLLFile = Dir("C:\WINNT\SYSTEM\*.DLL")
If DLLFile <> "" Then DLLFiles = DLLFiles + 1
While DLLFile <> ""
   DLLFile = Dir()
   DLLFiles = DLLFiles + 1
End While
MsgBox(DLLFiles)
```

The Dir() function is called for the first time with an argument. If a DLL file is found, its name is returned. Then the function is called repeatedly, this time without arguments. Each time it returns the name of the next DLL file, until all DLL files that match the original specification are exhausted. After the loop, the variable *DLLFiles* contains the number of DLL files in the `\WinNT\System` folder. (There should be two dozen DLL files there, even on a bare-bones system.)

If you want to find out whether there are any *hidden* DLL files in the `\WinNT\System` folder, supply the attributes arguments to the Dir() function:

```
HiddenFile = Dir("C:\WINNT\SYSTEM\*.DLL", FileAttribute.Hidden)
```

You can also combine multiple attributes by adding the corresponding constants.

To list all the subfolders in a given folder, you must specify the `FileAttribute.Directory` attribute, which returns folder names as well as the names of the normal files. To check whether an entry is a folder or a file, you must also examine its attributes with the GetAttr() function. The following loop counts the subfolders of the System folder.

```
Dim path, FName As String
Dim totFolders As Integer
path = "C:\"
FName = Dir(path, FileAttribute.Directory)
While FName <> ""
   If (GetAttr(path & FName) And FileAttribute.Directory) = _
         FileAttribute.Directory Then
      Console.WriteLine(FName)
      totFolders = totFolders + 1
   End If
   FName = Dir()
End While
Console.WriteLine("Found " & totFolders & " folders")
```

If you run this code segment, you'll see a list of folder names followed by the total number of folders under the specified folder.

FileCopy(source_file, dest_file)

The FileCopy() function copies a file to a new location on the hard disk. *source_file* is the name of the file to be copied. If the file is in the current folder, then you can specify its name only. Otherwise, you must specify the file's path name. The *dest_file* argument specifies the target file name and may include a folder and drive name. Notice that the file can't be copied if an application is using it at the time.

To copy the file `C:\VBMaterial\Examples\Files.txt` to `D:\MasteringVB\Files.txt`, use the following statements:

```
source = "C:\VBMaterial\Examples\Files.txt"
destination = "D:\MasteringVB\Files.txt"
FileCopy(source, destination)
```

The FileCopy() function does not allow wildcard characters. In other words, you can't use this function to copy multiple files at once.

Rename(oldpath, newpath)

The Rename() function renames a disk file or folder. The existing file's or folder's name is specified by the *oldpath* argument, and the new name is specified with the *newpath* argument. The path specified by the *newpath* argument should not exist already. The statement

```
Rename("C:\Users", "C:\All Users")
```

will rename the folder `C:\Users` to `C:\All Users`. The folder will be renamed even if it contains subfolders and/or files. If you attempt to rename two nested folders at once with a statement like the following one:

```
Rename("C:\Users\New User", "C:\All Users\User1")
```

a runtime error will be generated. Rename them one at a time (it doesn't make any difference which one is renamed first).

The Rename() function can rename a file and move it to a different directory or folder, if necessary. However, it can't move a folder (with or without its subfolders and files). If the folder `D:\New User` folder exists, the following statement will move the file `UserProfile.cps` to the folder `New User` on the D: drive and rename it as well:

```
Rename("C:\AllUsers\User1\Profile1.cps", "D:\New User\UserProfile.cps")
```

If the folder `D:\New User` does not exist, it will not be created automatically. You must first create it, then move the file there. The Rename() function cannot create a new folder.

Notice that the Rename() function can not act on an open file. You must first close it, then rename it. Like most file- and folder-manipulation statements of Visual Basic, the Rename function's arguments don't recognize wildcards.

Data Type Identification

The functions in this section merely identify a data type. To *change* data types, use the functions in the following section, "Variable Type Conversion."

IsArray(variable)

This function returns True if its argument is an array. If the variable *names* has been defined as

```
Dim names(100)
```

then the function

```
IsArray(names)
```

returns True.

IsDate(expression)

This function returns True if *expression* is a valid date. Use the IsDate() function to validate user data. Dates can be specified in various formats, and validating them without the help of the IsDate() function would be a task on its own.

```
BDate = InputBox("Please enter your birth date")
If IsDate(BDate) Then
   MsgBox "Date accepted"
End If
```

IsDBNull(expression)

This function returns True if *expression* is DBNull. A DBNull value is a nonvalid value and is different from the Nothing value. The DBNull value represents missing or nonexistent data.

IsNothing(expression)

This function returns a Boolean (True/False) value indicating whether *expression* represents an object variable that hasn't been instantiated yet. Let's say you've declared an object variable with the following statements:

```
Dim obj As Object
```

After this declaration, the expression `IsNothing(obj)` is True. Later in your code, you assign an object to the *obj* variable with the following statement:

```
obj = New Rectangle(10, 100, 12, 12)
```

After the execution of this statement, *obj* is no longer Nothing. You can set it back to Nothing explicitly to release the object it references (the Rectangle object):

```
obj = Nothing
```

You can also compare an object variable against the Nothing value with the following statement:

```
If IsNothing(obj) Then
   { code to initialize variable }
End If
{ code to process variable }
```

You can also use the Is keyword to compare an object variable against the Nothing value:

```
If obj Is Nothing Then
```

IsNumeric(expression)

This function returns True if *expression* is a valid number. Use this function to check the validity of strings containing numeric data as follows:

```
age = InputBox("Please enter your age")
If Not IsNumeric(age) Then
   MsgBox("Please try again, this time with a valid number")
End If
```

IsReference(expression)

This function returns a Boolean (True/False) value indicating whether *expression* represents an object variable. To find out the type of object, use the TypeName() or VarType() functions, which are described next.

TypeName(variable_name)

This function returns a string that identifies the variable's type. The variable whose type you're examining with the TypeName function may have been declared implicitly or explicitly. Suppose you declare the following variables

```
Dim name As Integer
Dim a
```

The following statements produce the results shown in bold:

```
Console.WriteLine(TypeName(name))
   Integer
Console.WriteLine(TypeName(a))
   Nothing
a = "I'm a string"
Console.WriteLine(TypeName(a))
   String
```

VarType(variable)

The VarType() function returns a member of the VariantType enumeration indicating the type of a variable, according to Table 6.

Table 6: The VariantType Enumeration

Constant	Description
Array	Array
Boolean	Boolean
Byte	Byte
Char	Character
Currency	Currency
DataObject	A data-access object
Date	Date
Decimal	Decimal
Double	Double-precision floating-point number
Empty	Empty (uninitialized)
Error	Error
Integer	Integer
Long	Long integer
Null	Null (no valid data)
Object	Automation object
Short	Short integer
Single	Single-precision floating-point number
String	String
UserDefinedType	User-defined type (structure). Each member of the structure has its own type.
Variant	Variant (used only with arrays of Variants)

Variable Type Conversion

These functions convert their numeric argument to the corresponding type. With the introduction of the Variant data type, these functions are of little use. You can use them to document your code and show that the result of an operation should be of the particular type, but keep in mind that all operands in an arithmetic operation are first converted to double-precision numbers for the greatest possible accuracy. Table 7 lists the variable type conversion functions.

TABLE 7: VARIABLE TYPE CONVERSION FUNCTIONS

FUNCTION	CONVERTS ITS ARGUMENT TO
CBool(*expression*)	Boolean (True/False)
CByte(*expression*)	Byte
CChar(*expression*)	Char
CDate(*expression*)	Date
CDec(*expression*)	Decimal
CDbl(*expression*)	Double
CInt(*expression*)	Integer
CLng(*expression*)	Long
CObj(*expression*)	Object
CShort(*expression*)	Short
CSng(*expression*)	Single
CStr(*expression*)	String
CType(*expression, type*)	The type specified

CType(varName, typeName)

This function converts the variable (or expression) specified by the first argument to the type specified by the second argument. The following statements convert the integer value 1,000 and the string "1000" to Double values:

```
CType(1000, System.Double)
CType("1000", System.Double)
```

String Manipulation

The following functions manipulate strings. Visual Basic .NET provides an impressive array of functions for string manipulation, as the average application spends most of its time operating on strings, not numbers. This group contains a single statement, the Mid statement, which happens to have the same name as the Mid() function. All the string-manipulation functions of Visual Basic

have an equivalent method (or property) in the `System.String` class, as well as in the StringBuilder class, which were described in Chapter 12.

Asc(character), AscW(string)

The Asc() function returns the character code corresponding to the *character* argument, and it works on all systems, regardless of whether they support Unicode characters. If you specify a string as argument to the Asc() function, it will return the character code of the first character in the string.

The AscW() function returns the Unicode character code except on platforms that do not support Unicode, in which case, the behavior is identical to that of the Asc() function.

If you call either function with a string instead of a character, the character code of the string's first character is returned.

Chr(number), ChrW(number)

The Chr() function is the inverse of the Asc() function and returns the character associated with the specified character code. Use this function to print characters that don't appear on the keyboard (such as line feeds or special symbols).

The ChrW() function returns a string containing the Unicode character except on platforms that don't support Unicode, in which case, the behavior is identical to that of the Chr() function.

LCase(string), UCase(string)

The LCase() function accepts a string as an argument and converts it to lowercase; the Ucase() function accepts a string as an argument and converts it to uppercase. After the following statements are executed:

```
Title = "Mastering Visual Basic"
LTitle = LCase(Title)
UTitle = UCase(Title)
```

the variable *LTitle* contains the string "mastering visual basic", and the variable *UTitle* contains the string "MASTERING VISUAL BASIC".

InStr([startPos,] string1, string2[, compare])

The InStr() function returns the position of *string2* within *string1*. The first argument, which is optional, determines where in *string1* the search begins. If the *startPos* argument is omitted, the search begins at the first character of *string1*. If you execute the following statements:

```
str1 = "The quick brown fox jumped over the lazy dog"
str2 = "the"
Pos = Instr(str1, str2)
```

the variable *Pos* will have the value 33. If you search for the string "he" by setting:

```
str2 = "he"
```

the *Pos* variable's value will be 2. If the search begins at the third character in the string, the first instance of the string "he" after the third character in the original string will be located:

```
Pos = InStr(3, str1, str2)
```

This time the *Pos* variable will be 34.

The search is by default case-sensitive. To locate "the", "The", or "THE" in the string, specify the last, optional argument, whose value is `CompareMethod.Binary` (default) for a case-sensitive search and `CompareMethod.Text` for a case-insensitive search (Table 8).

TABLE 8: THE COMPAREMETHOD ENUMERATION

VALUE	DESCRIPTION
`Binary`	Performs a binary (case-sensitive) comparison
`Text`	Performs a textual (case-insensitive) comparison

The following statement locates the first occurrence of "the" in the string, regardless of case:

```
str1 = "The quick brown fox jumped over the lazy dog"
str2 = "the"
Pos = InStr(1, str1, str2, CompareMethod.Text)
```

The value of *Pos* will be 1. If you set the last argument to `CompareMethod.Binary`, the *Pos* variable becomes 33.

InStrRev(string1, string2[, start][, compare])

This function returns the position of one string within another as does the InStr() function, but it starts from the end of the string (hence InStrRev = "in string reverse"). The *string1* argument is the string being searched, and the *string2* argument is the string being searched for. The other two arguments are optional. The *start* argument is the starting position for the search. If it is omitted, the search begins at the last character. Notice that in this function, the starting location of the search is the third argument. The *compare* argument indicates the kind of comparison to be used in locating the substrings, and its value is one of the members of the CompareMethod enumeration, listed in Table 8 earlier. If *compare* is omitted, a binary comparison is performed.

StrComp(string1, string2[, compare])

This function compares two strings and returns a value indicating the result, according to Table 9.

TABLE 9: VALUES RETURNED BY THE STRCOMP() FUNCTION

VALUE	DESCRIPTION
–1	*string1* is less than *string2*.
0	*string1* is equal to *string2*.
1	*string1* is greater than *string2*.
Null	*string1* and/or *string2* is Null.

The last argument of the StrComp() function determines whether the comparison will be case-sensitive. If *compare* is `CompareMethod.Binary` (or omitted), the comparison is case-sensitive. If it's `CompareMethod.Text`, the comparison is case-insensitive.

The following function:

```
StrComp("Sybex", "SYBEX")
```

returns 1 ("Sybex" is greater than "SYBEX", because the lowercase *y* character is after the uppercase *Y* in the ASCII sequence). The function

```
StrComp("Sybex", "SYBEX", CompareMethod.Text)
```

returns 0.

Left(string, number)

This function returns a number of characters from the beginning of a string. It accepts two arguments: the string and the number of characters to extract. If the string *date1* starts with the month name, the following Left() function can extract the month's abbreviation from the string, as follows:

```
date1 = "December 25, 1995"
MonthName = Left(date1, 3)
```

The value of the *MonthName* variable after the execution of the statements is "Dec".

Right(string, number)

This function is similar to the Left() function, except that it returns a number of characters from the end of a string. The following statements

```
date1 = "December 25, 1995"
Yr = Right(date1, 4)
```

assign to the *Yr* variable the value "1995".

Mid(string, start, [length])

The Mid() function returns a section of a string of *length* characters, starting at position *start*. The following function:

```
Mid("09 February, 1957", 4, 8)
```

extracts the name of the month from the specified string.

If you omit the *length* argument, the Mid() function returns all the characters from the starting position to the end of the string. If the specified length exceeds the number of characters in the string after the start position, the remaining string from the start location is returned.

Mid(string, start[, length]) = new_string

In addition to the Mid() function, there's a Mid statement, which does something similar. Instead of extracting a few characters from a string, the Mid statement replaces a specified number of characters in a String variable (specified with the first argument, *string*) with another string (the argument

new_string). The location and count of characters to be replaced are specified with the arguments *start* and *length*. The last argument is optional. If you omit it, all the characters from the starting character to the end of the string will be replaced.

If you're writing code that performs operations with long strings, you should use the StringBuilder class of the .NET Framework. StringBuilder variables are text variables, but the compiler can handle them much more efficiently than strings. The StringBuilder class is discussed in detail in Chapter 12.

Len(string)

The Len() function returns the length of a string. After the following statements execute:

```
Name = InputBox("Enter your first name")
NameLen = Len(Name)
```

the variable *NameLen* contains the length of the string entered by the user in the input box.

The Len() function is frequently used as a first test for invalid input, as in the following lines:

```
If Len(Name) = 0 Then
   MsgBox ("NAME field can't be empty")
Else
   MsgBox ("Thank you for registering with us")
EndIf
```

The Len() function can accept any base type as argument, and it returns the number of bytes required to store the variable. The expression

```
Len(12.01)
```

will return 8 (by default, floating-point values are stored in Double variables). The expression

```
Len(12)
```

will return 4, because integers are stored in 4 bytes.

LTrim(string), RTrim(string), Trim(string)

These functions trim the spaces in front of, after, and on both sides of a string, respectively. They are frequently used in validating user input. Let's say you want to make sure that the *EMail* variable isn't empty and you use the following If structure:

```
If EMail <> "" Then
   MsgBox ("Applications without an e-mail address won't be processed")
End If
```

The preceding won't, however, catch a string that only has spaces. To detect empty strings, use the Trim() function instead:

```
If Trim(EMail) = "" Then
   MsgBox ("Invalid Entry!")
End If
```

Space(number)

This function returns a string consisting of the specified number of spaces. The *number* argument is the number of spaces you want in the string. This function is useful for formatting output and clearing data in fixed-length strings.

StrDup(number, character)

This function returns a string of *number* characters, all of which are *character*. The following function:

```
StrDup(12, "*")
```

returns the string "**********". Use the StrDup() function to create long patterns of special symbols. The StrDup() function replaces the String() function of VB6.

StrConv(string, conversion)

This function returns a string variable converted as specified by the *conversion* argument, whose values as shown in Table 10.

TABLE 10: THE VBSTRCONV ENUMERATION

CONSTANT	CONVERTS
UpperCase	The string to uppercase characters
LowerCase	he string to lowercase characters
ProperCase	The first letter of every word in *string* to uppercase
Wide	Narrow (single-byte) characters in *string* to wide (double-byte) characters*
Narrow	Wide (double-byte) characters in *string* to narrow (single-byte) characters*
Katakana	Hiragana characters in *string* to Katakana characters*
Hiragana	Katakana characters in *string* to Hiragana characters*
TraditionalChinese	Simplified Chineese to traditional Chinese*
SimplifiedChinese	Traditional Chinese to simplified Chinese*

**Applies to Far East locales.*

To perform multiple conversions, add the corresponding values. To convert a string to lowercase and to Unicode format, use a statement such as the following:

```
newString = StrConv(txt, vbStrConv.LowerCase + vbStrConv.Unicode)
```

StrReverse(string)

This function reverses the character order of its argument. Its syntax is:

```
StrReverse(string)
```

where *string* is a string variable or expression that will be reversed.

Filter(inputStrings, value[, include][, compare])

This function returns an array containing part of a string array, based on specified filter criteria. The *inputStrings* argument is a one-dimensional array of the strings to be searched, and the *value* argument is the string to search for. The last two arguments are optional. *include* indicates whether the function should contain substrings that include or exclude the specified value. If True, the Filter() function returns the subset of the array that contains *value* as a substring. If False, the Filter() function returns the subset of the array that does not contain *value* as a substring. The *compare* argument indicates the kind of string comparison to be used and can be either of the values (`CompareMethod.Binary` or `CompareMethod.Text`) in Table 8, shown previously in this reference.

The array returned by the Filter() function contains only enough elements to store the number of matched items. To use the Filter() function, you must declare an array without specifying its dimensions, which will accept the selected strings. Let's say you have declared the Names array as follows:

```
Dim selNames() As String
Dim Names() As String = {"Abe", "John", "John", "Ruth", "Pat"}
```

You can find out if the name stored in the variable *myName* is in the *Names* array by calling the Filter() function as follows:

```
selNames = Filter(Names, myName)
```

If the name stored in the variable *myName* isn't part of the *Names* array, *selNames* is an array with no elements (its Length property is 0). If the name stored in the variable *myName* is "Abe," the upper bound of the array *selNames* will be 0, and the element `selNames(0)` will be "Abe." If the value of the *myName* variable is "John," the upper bound of the *selNames* array will be 1, and the elements `selNames(0)` and `selNames(1)` will have the value "John."

You can also create an array that contains all the elements in the original, except for a specific value. The array *selNames* created with the statement

```
selNames = Filter(Names, "Ruth", False)
```

will have 4 elements, which are all the elements of the array *Names* except for "Ruth."

Replace(expression, find, replacewith[, start][, count][, compare])

This function returns a string in which a specified substring has been replaced with another substring, a specified number of times. The *expression* argument is a string on which the Replace function acts. The *find* argument is the substring to be replaced, and *replacewith* is the replacement string. The remaining arguments are optional. The *start* argument is the character position where the search begins. If it is omitted, the search starts at the first character. The *count* argument is the number of replacements to be performed. If it is omitted, all possible replacements will take place. Finally, the *compare* argument specifies the kind of comparison to be performed. The values of the *compare* argument are the members of the CompareMethod enumeration, described in Table 8.

Join(list[, delimiter])

This function returns a string created by joining a number of substrings contained in an array. The *list* argument is a one-dimensional array containing the strings to be joined, and the optional *delimiter* argument is a character used to separate the substrings in the returned string. If it is omitted, the space character (" ") is used. If *delimiter* is a zero-length string, all items in the list are concatenated with no delimiters.

Split(expression[, delimiter][, count][, compare])

This function is the counterpart of the Join() function. It returns a one-dimensional array containing a number of substrings. The *expression* argument is a string that contains the original string that will be broken into smaller strings, and the optional *delimiter* argument is a character delimiting the substrings in the original string. If *delimiter* is omitted, the space character (" ") is assumed to be the delimiter. If *delimiter* is a zero-length string, a single-element array containing the entire expression string is returned. The *count* argument is also optional, and it determines the number of substrings to be returned. If it's –1, all substrings are returned. The last argument, *compare*, is also optional and indicates the kind of comparison to use when evaluating substrings. Its value can be one of the CompareMethod enumeration's members (Table 8).

Let's say you have declared a string variable with the following path name:

```
path = "c:\win\desktop\DotNet\Examples\Controls"
```

The Split() function can extract the path's components and assign them to the `parts` array, if called as follows:

```
parts = Split("c:\win\desktop\DotNet\Examples\Controls", "\")
```

To display the parts of the path, set up a loop such as the following:

```
For i = 0 To parts.GetUpperBound(0)
   Console.WriteLine(parts(i))
Next
```

Data Formatting

In addition to the ToString method exposed by all VB.NET variables, all VB6 formatting functions are supported by VB.NET.

Format(expression[, format[, firstdayofweek[, firstweekofyear]]])

This function returns a string containing an expression formatted according to instructions contained in a format expression. The *expression* variable is the number, string, or date to be converted, and *format* is a string that tells Visual Basic how to format the value. The string "hh:mm.ss", for example, displays the expression as a time string (if the first argument is a date expression).

The Format() function is used to prepare numbers, dates, and strings for display. `Atan(1)*4` calculates pi with double precision; if you attempt to display the following expression:

```
Console.WriteLine(Math.Atan(1)*4)
```

the number 3.14159265358979 is displayed. If this value must appear in a text control, chances are good that it will overflow the available space.

You can control the number of decimal digits to be displayed with the following call to the Format() function:

```
Console.WriteLine(Format(Math.Atan(1)*4, "##.####"))
```

This statement displays the result 3.1416. If you are doing financial calculations and the result turns out to be 13,454.332345201, it would best to display it as a proper dollar amount, with a statement such as the following:

```
amount = 13454.332345201
Console.WriteLine(Format(amount, "$###,###.##"))
```

These statements display the value $13,454.33.

The *firstdayofweek* and *firstweekofyear* arguments are used only in formatting dates. The *firstdayofweek* argument determines the week's first day and can have one of the values in Table 11. Similarly, *firstweekofyear* determines the first week of the year, and it can have one of the values in Table 12.

Table 11: The DayOfWeek Enumeration

Constant	Value	Description
System	0	Use NLS API setting
Sunday	1	Sunday (default)
Monday	2	Monday
Tuesday	3	Tuesday
Wednesday	4	Wednesday
Thursday	5	Thursday
Friday	6	Friday
Saturday	7	Saturday

Table 12: The WeekOfYear Enumeration

Constant	Value	Description
System	0	Uses NLS API setting
FirstJan1	1	Year starts with the week of January 1
FirstFourDays	2	Year starts with the week that has at least four days
FirstFullWeek	3	Year starts with the first full week

There are many formatting strings for all three types of variables: numeric, string, and date and time. Tables 13 through 15 show them.

Table 13: User-Defined Time and Date Formatting

Character	Description
:	Time separator. In some locales, other characters may be used to represent the time separator. The time separator separates hours, minutes, and seconds when time values are formatted.
/	Date separator. In some locales, other characters may be used to represent the date separator. The date separator separates the day, month, and year when date values are formatted.
d	Displays day as a number (1–31).
dd	Displays day as a number with a leading zero (01–31).
ddd	Displays day as an abbreviation (Sun–Sat).
dddd	Displays day as a full name (Sunday–Saturday).
w	Displays day of the week as a number (1 for Sunday through 7 for Saturday).
ww	Displays week of the year as a number (1–54).
M	Displays month as a number (1–12). If M immediately follows h or hh, the minute rather than the month is displayed.
MM	Displays month as a number with a leading zero (01–12). If M immediately follows h or hh, the minute rather than the month is displayed.
MMM	Displays month as an abbreviation (Jan–Dec).
MMMM	Displays month as a full month name (January–December).
q	Displays quarter of the year as a number (1–4).
y	Displays day of the year as a number (1–366).
yy	Displays year as a 2-digit number (00–99).
yyyy	Displays year as a 4-digit number (0100–9999).
h	Displays hours as a number (0–12).
hh	Displays hours with leading zeros (00–12).
H	Displays hours as a number in 24-hour format (0–24)
HH	Displays hours with a leading zero as a number in 24-hour format (00–24)
m	Displays minutes without leading zeros (0–59).
mm	Displays minutes with leading zeros (00–59).
s	Displays seconds without leading zeros (0–59).

Continued on next page

Table 13: User-Defined Time and Date Formatting *(continued)*

Character	Description
ss	Displays seconds with leading zeros (00–59).
AM/PM	Uses the 12-hour format and displays the indication AM/PM.
am/pm	Uses the 12-hour format and displays the indication am/pm.
A/P	Uses the 12-hour format and displays the indication A/P
a/p	Uses the 12-hour format and displays the indication a/p.
AMPM	Uses the 12-hour format and displays the AM/PM string literal as defined by the system. Use the Regional Settings applet in the Control Panel to set this literal for your system.

Table 14: User-Defined Number Formatting

Character	Description	Explanation
None		Displays the number with no formatting.
0	Digit placeholder	Displays a digit or a zero. If the expression has a digit in the position where the 0 appears in the format string, display it; otherwise, display a zero in that position. If the number has fewer digits than there are zeros in the format expression, leading or trailing zeros are displayed. If the number has more digits to the right of the decimal separator than there are zeros to the right of the decimal separator in the format expression, round the number to as many decimal places as there are zeros. If the number has more digits to the left of the decimal separator than there are zeros to the left of the decimal separator in the format expression, display the extra digits without modification.
#	Digit placeholder	Displays a digit or nothing. If the expression has a digit in the position where the # appears in the format string, display it; otherwise, display nothing in that position. This symbol works like the 0 digit placeholder, except that leading and trailing zeros aren't displayed if the number has the same or fewer digits than there are # characters on either side of the decimal separator in the format expression.
.	Decimal placeholder	The decimal placeholder determines how many digits are displayed to the left and right of the decimal separator. If the format expression contains only number signs to the left of this symbol, numbers smaller than 1 begin with a decimal separator. To display a leading zero displayed with fractional numbers, use 0 as the first digit placeholder to the left of the decimal separator.

Continued on next page

Table 14: User-Defined Number Formatting *(continued)*

Character	Description	Explanation
%	Percentage placeholder	The expression is multiplied by 100. The percent character (%) is inserted in the position where it appears in the format string.
,	Thousands separator	Separates thousands from hundreds within a number greater than 1,000. Two adjacent thousands separators or a thousands separator immediately to the left of the decimal separator (whether or not a decimal is specified) means "scale the number by dividing it by 1,000, rounding as needed." For example, you can use the format string "##0,," to represent 100 million as 100. Numbers smaller than 1 million are displayed as 0. Two adjacent thousands separators in any position other than immediately to the left of the decimal separator are treated as a thousands separator.
:	Time separator	Separates hours, minutes, and seconds when time values are formatted.
/	Date separator	Separates the day, month, and year when date values are formatted.
E-, E+, e-, e+	Scientific format	If the format expression contains at least one digit placeholder (0 or #) to the right of E-, E+, e-, or e+, the number is displayed in scientific format, and E or e is inserted between the number and its exponent. The number of digit placeholders to the right determines the number of digits in the exponent. Use E- or e- to place a minus sign next to negative exponents. Use E+ or e+ to place a minus sign next to negative exponents and a plus sign next to positive exponents.
+ $ (space)	Displays a literal character	To display a character other than one of those listed, precede it with a backslash (\) or enclose it in double quotation marks (" ").
\	Escape character; displays the next character in the format string	To display a character that has special meaning as a literal character, precede it with a backslash (\). The backslash itself isn't displayed. Using a backslash is the same as enclosing the next character in double quotation marks. To display a backslash, use two backslashes (\\). Examples of characters that can't be displayed as literal characters are the date-formatting and time-formatting characters (a, c, d, h, m, n, p, q, s, t, w, y, / and :), the numeric-formatting characters (#, 0, %, E, e, comma, and period), and the string-formatting characters (@, &, <, >, and !).
"ABC"	Displays the string inside the double quotation marks (" ")	To include a string in format from within code, you must use Chr(34) to enclose the text (34 is the character code for a quotation mark (")).

Table 15: User-Defined String Formatting

Character	Description	Explanation
@	Character placeholder	Displays a character or a space. If the string has a character in the position where the @ symbol appears in the format string, it is displayed. Otherwise, a space in that position is displayed. Placeholders are filled from right to left unless there is an exclamation point character (!) in the format string.
&	Character placeholder	If the string has a character in the position where the ampersand (&) appears, it is displayed. Otherwise, nothing is displayed. Placeholders are filled from right to left unless there is an exclamation point character (!) in the format string.
<	Force lowercase	All characters are first converted to lowercase.
>	Force uppercase	All characters are first converted to uppercase.
!	Scans placeholders from left to right	The default order is to use placeholders from right to left.

FormatCurrency(expression[, numDigitsAfterDecimal] [, includeLeadingDigit][, useParensForNegativeNumbers][, groupDigits])

This function returns a numeric expression formatted as a currency value (dollar amount) using the currency symbol defined in Control Panel. All arguments are optional, except for the *expression* argument, which is the number to be formatted as currency. *numDigitsAfterDecimal* is a value indicating how many digits will appear to the right of the decimal point. The default value is –1, which indicates that the computer's regional settings must be used. *includeLeadingDigit* is a tristate constant that indicates whether a leading zero is displayed for fractional values. The *useParensForNegativeNumbers* argument is also a tristate constant that indicates whether to place negative values within parentheses. The last argument, *groupDigits*, is another tristate constant that indicates whether numbers are grouped using the group delimiter specified in the computer's regional settings.

Note *A tristate variable is one that has three possible values:* `True`, `False`, *and* `UseDefault`. *The last value uses the computer's regional settings. When one or more optional arguments are omitted, values for omitted arguments are provided by the computer's regional settings.*

FormatDateTime(date[, namedFormat])

This function formats a date or time value. The *date* argument is a date value that will be formatted, and the optional argument *namedFormat* indicates the date/time format to be used. It can have the values shown in Table 16.

Table 16: The DateFormat Enumeration

Value	Description
GeneralDate	Displays a date and/or time. If a date part is present, it is displayed as a short date. If a time part is present, it is displayed as a long time. If both parts are present, both parts are displayed.
LongDate	Displays a date using the long date format, as specified in the client computer's regional settings.
ShortDate	Displays a date using the short date format, as specified in the client computer's regional settings.
LongTime	Displays a time using the time format specified in the client computer's regional settings.
ShortTime	Displays a time using the 24-hour format.

FormatNumber(expression[, numDigitsAfterDecimal] [, includeLeadingDigit][, useParensForNegativeNumbers][, groupDigits])

This function returns a numeric value formatted as a number. The arguments of the FormatNumber() function are identical to the arguments of the FormatCurrency() function, described earlier.

FormatPercent(expression[, numDigitsAfterDecimal] [, includeLeadingDigit][, useParensForNegativeNumbers][, groupDigits])

This function returns an expression formatted as a percentage (multiplied by 100) with a trailing % character. Its syntax and arguments are identical to the FormatCurrency() function, described earlier.

LSet(string, len), RSet(string, len)

These two functions left- or right-align a string within a string variable and return a new string with the proper number of spaces before or after. The statements:

```
Console.WriteLine("[" & RSet("Hohnecker", 20) & "]")
Console.WriteLine("[" & LSet("Richard", 20) & "]")
```

will print the following string in the Output window:

```
[          Hohnecker]
[Richard             ]
```

The last name is right-aligned in a string of 20 characters, and the first name is left-aligned in a string of 20 characters. If you create multiple strings like the previous and place them one below the other (on a TextBox or ListBox control, for example), the commas will not align unless a monospaced font such as Courier is used.

The LSet() and RSet() functions can also be used with numeric values. The statements

```
Console.WriteLine(RSet(34.56, 10))
Console.WriteLine(RSet(4356.99, 10))
Console.WriteLine(RSet(4.01, 10))
```

will produce the following output on the Output window:

```
  34.56
4356.99
   4.01
```

Val(string), Str(number)

The Val() function accepts as argument a string and returns its numeric value. The Str() function accepts as argument a number and returns its string representation. The Val() function starts reading the string from the left and stops when it reaches a character that isn't part of a number. If the value of the variable *a* is:

```
Dim a As String = "18:6.05"
```

then the statement

```
Console.WriteLine(Val(a))
```

returns 18. Conversely, the following statement returns the string "18":

```
Dim a As Integer = 18
Console.WriteLine(Str(a))
```

Math

All the Math functions of VB6 have been replaced by the methods of the Math class. Since I haven't discussed these methods in the book, I'm listing them in this section. To use any of the math methods of VB.NET, you must import the Math class or prefix the names of the methods with the name of the class (`Math.Abs`, `Math.Cos`, and so on).

The following methods perform math operations. Their arguments are double-precision values and so are their results.

Abs(expression)

This method returns the absolute value of its argument. Both `Abs(1.01)` and `Abs(-1.01)` return the value 1.01.

Atan(expression)

This method returns the arctangent of an angle. The value returned is in radians.

TIP *To convert a radian value to degrees, multiply by (180 / pi).*

Cos(expression)

This method returns the cosine of an angle. The value of *expression* must be in radians.

TIP *To convert a degree value to radians, multiply by (pi / 180).*

Exp(expression)

This method returns the base of the natural logarithm to a power. The *expression* variable is the power, and its value can be a noninteger, positive or negative value. The Exp() method complements the operation of the Log() method and is also called *antilogarithm*.

Int(expression), Fix(expression)

Both of these methods accept a numeric argument and return an integer value. If *expression* is positive, both methods behave the same. If it's negative, the Int() method returns the first negative integer less than or equal to *expression*, and Fix() returns the first negative integer greater than or equal to *expression*. For example, `Int(-1.1)` returns –2, and `Fix(-1.1)` returns –1. The expressions `Int(1.8)` and `Fix(1.8)` both return 1.

If you want to get rid of the decimal part of a number and round it as well, you can use the following expression:

```
Int(value + 0.5)
```

The *value* argument is the number to be rounded. However, the new Round() method (see next entry) provides a simpler technique for rounding values to any desired precision.

Round(expression[, numdecimalplaces])

This method returns a numeric expression rounded to a specified number of decimal places. The *numdecimalplaces* argument is optional and indicates how many places to the right of the decimal are included in the rounding. If it is omitted, an integer value is returned.

The expression `Round(3.49)` returns 3, and the expression `Round(3.51)` returns 4. Both `Round(3.49, 1)` and `Round(3.51, 1)` return 3.5.

Log(expression)

The Log() method returns the natural logarithm of a number. The *expression* variable must be a positive number. The expression `Log(Exp(N))` returns N, and so does the expression `Exp(Log(N))`. If you combine the logarithm with the antilogarithm, you end up with the same number.

The natural logarithm is the logarithm to the base e, which is approximately 2.718282. The precise value of e is given by the expression `Exp(1)`. To calculate logarithms to other bases, divide the natural logarithm of the number by the natural logarithm of the base. The following statement calculates the logarithm of a number in base 10:

```
Log10 = Log(number) / Log(10)
```

Hex(expression), Oct(expression)

These two methods accept a decimal numeric value as an argument and return the octal and hexadecimal representation of the number in a string. The expression `Hex(47)` returns the value "2F", and the expression `Oct(47)` returns the value "57". To specify a hexadecimal number, prefix it with &H. The equivalent notation for octal numbers is &O. Given the following definitions:

```
Dvalue = 199: Ovalue = &O77
```

the expression `Oct(Dvalue)` returns the string "307", and the expression `Hex(Ovalue)` returns "3F". To display the decimal value of 3F, use a statement such as the following:

```
MsgBox ("The number 3F in decimal is " & &H3F)
```

The actual value that will be displayed is 63.

Pow(value1, value2)

The Pow() method accepts two numeric arguments and returns the first number raised to the power specified by the second argument. The statement

```
Console.WriteLine(Math.Pow(2, 3))
```

will print the value 8 (2 to the power of 3, or $2 \times 2 \times 2$) on the Output window.

Sin(expression)

This method returns the sine of an angle, specified in radians. See the Cos() entry earlier in this reference.

Sqrt(expression)

This method returns the square root of its expression.

Tan(expression)

This method returns the tangent of an angle, which must be expressed in radians. See the Atan() entry earlier in this reference.

Date and Time

Figuring out the number of hours, days, or weeks between two days could be a project on its own. Not with Visual Basic. There are so many functions and statements for manipulating time and date values, all you have to do is select the one you need for your calculations and look up its arguments.

Now()

This function returns both the system date and time. The statement

```
MsgBox(Now())
```

displays a date/time combination such as 9/13/1998 09:23:10 PM in a message box. There's only one space between the date and the time.

To extract the date or time part of the value returned by the Now() function, use the Date and TimeOfDay properties:

```
Console.WriteLine(Now.Date.ToString)
Console.WriteLine(Now.TimeOfDay.ToString)
```

Day(date)

This function returns the day number of the date specified by the argument. The *date* argument must be a valid date (such as the value of the Now() function). If the following function had been called on 12/15/2000, it would have returned 15:

```
Day(Now())
```

Weekday(date[, firstdayofweek])

This function returns an integer in the range 1 through 7, representing the day of the week (1 for Sunday, 2 for Monday, and so on). The first argument, *date*, can be any valid date expression. The second argument, which is optional, specifies the first day of the week and can have any of the values shown in Table 11, in the Format() entry earlier in this reference.

WeekdayName(weekday[, abbreviate[, firstdayofweek]])

This function returns the name of the weekday specified by the *weekday* argument (a numeric value, which is 1 for the first day, 2 for the second day, and so on). The optional *abbreviate* argument is a Boolean value that indicates whether the name is to be abbreviated. By default, day names are not abbreviated. The last argument, *firstdayofweek*, is also optional and determines the first day of the week. Its valid values are shown in Table 11, in the Format() entry earlier in this reference. By default, the first day of the week is Sunday.

Month(date)

This function returns an integer in the range 1 through 12, representing the number of the month of the specified date. `Month(Date())` returns the current month number.

MonthName(month[, abbreviate])

This function returns the name of the month specified by the *month* argument (a numeric value, which is 1 for January, 2 for February, and so on). The optional *abbreviate* argument is a Boolean value that indicates whether the month name is to be abbreviated. By default, month names are not abbreviated.

Year(date)

This function returns an integer representing the year of the date passed to it as an argument. The function `Year(Now())` returns the current year.

Hour(time)

This function returns an integer in the range 0 through 24 that represents the hour of the specified time. The following statements:

```
Console.WriteLine(Now())
Console.WriteLine(Hour(Now()))
```

produce something such as:

```
9/5/2001 1:43:19 AM
1
```

Minute(time)

This function returns an integer in the range 0 through 60 that represents the minute of the specified time. The following statements:

```
Console.WriteLine(Now())
Console.WriteLine(Minute(Now()))
```

produce something such as:

```
9/5/2001 1:43:19 AM
43
```

Second(time)

This function returns an integer in the range 0 through 60 that represents the seconds of the specified time. The following statements:

```
Console.WriteLine(Now())
Console.WriteLine(Second(Now()))
```

produce something such as:

```
9/5/2001 1:43:19 AM
19
```

DateSerial(year, month, day)

This function accepts three numeric arguments that correspond to a year, a month, and a day and returns the corresponding date. The following statement:

```
MsgBox(DateSerial(2002, 10, 1))
```

displays the string "10/1/02" in a message box.

The DateSerial function can handle arithmetic operations with dates. For example, you can find out the date of the 90th day of the year by calling DateSerial() with arguments like these:

```
DateSerial(1996, 1, 90)
```

(3/30/1996, if you are curious). To find out the date 1,000 days from now, call the DateSerial() function as follows:

```
MsgBox(DateSerial(Year(Now.Date), Month(Now.Date), Weekday(Now.Date) + 1000))
```

You can also add (or subtract) a number of months to the *month* argument and a number of years to the *year* argument.

DateValue(date)

This function accepts a string that represents a Date value and returns the corresponding date value If you call the DateValue() function with the argument "December 25, 2002", you will get back the date 12/25/2002. DateValue() is handy if you are doing financial calculations based on the number of days between two dates. The difference in the following statement:

```
MsgBox(DateDiff(DateInterval.Day, DateValue("12/25/1993"), _
                                  DateValue("12/25/1996")))
```

is the number of days between the two dates, which happens to be 1,096 days.

TimeSerial(hours, minutes, seconds)

This function returns a time, as specified by the three arguments. The following function:

```
TimeSerial(4, 10, 55)
```

returns:

```
4:10:55 AM
```

The TimeSerial() function is frequently used to calculate relative times. The following call to TimeSerial() returns the time 2 hours, 15 minutes, and 32 seconds before 4:13:40 P.M.:

```
TimeSerial(16 - 2, 13 - 15, 40 - 32)
```

which is 1:58:08 P.M..

TimeValue(time)

This function accepts a string as argument and returns a date value. Like the DateValue() function, it can be used in operations that involve time.

DateAdd(interval, number, date)

This function returns a date that corresponds to a date plus some interval. The *interval* variable is a time unit (days, hours, weeks, and so on), *number* is the number of intervals to be added to the initial date, and *date* is the initial date. If *number* is positive, the date returned by DateAdd() is in the future. If it's negative, the date returned is in the past. The *interval* argument can take one of the values in Table 17.

TABLE 17: THE DATEINTERVAL ENUMERATION

VALUE
Year
Quarter
Month
DayOfYear

Continued on next page

TABLE 17: THE DATEINTERVAL ENUMERATION *(continued)*

VALUE
Day
WeekDay
WeekOfYear
Hour
Minute
Second

To find out the date one month after December 31, 2002, use the following statement:

```
Console.WriteLine(DateAdd(DateInterval.Month, 1, #12/31/2002#))
```

The result is:

```
1/31/2003 12:00:00 AM
```

The DateAdd() function is similar to the DateSerial() function (described earlier), but it takes into consideration the actual duration of a month. For DateSerial(), each month has 30 days. The following statements:

```
day1 = #1/31/2002#
Console.WriteLine(DateSerial(year(day1), month(day1) + 1, day(day1)))
```

result in:

```
3/2/02
```

which is a date in March, not February.

DateDiff(interval, date1, date2[, firstdayofweek[, firstweekofyear]])

This function is the counterpart of the DateAdd() function and returns the number of intervals between two dates. The *interval* argument is the interval of time you use to calculate the difference between the two dates (see Table 17, in the preceding DateAdd() entry, for valid values). The *date1* and *date2* arguments are dates to be used in the calculation, and *firstdayofweek* and *firstweekofyear* are optional arguments that specify the first day of the week and the first week of the year.

Table 11 shows the valid values for the *firstdayofweek* argument, and Table 12 shows the valid values for the *firstweekofyear* argument. (These tables can be found earlier, in the Format() entry.)

You can use the DateDiff() function to find how many days, weeks, and even seconds are between two dates. The following statement displays the number of days and minutes since the turn of century:

```
Dim century As Date
century = #1/1/2000#
MsgBox(DateDiff(DateInterval.Day, century, Now()))
```

DatePart(interval, date[,firstdayofweek[, firstweekofyear]])

This function returns the specified part of a given date. The *interval* argument is the desired format in which the part of the date will be returned (see Table 13, earlier in this reference, for its values), and *date* is the date you are examining. The optional arguments *firstdayofweek* and *firstdayofmonth* are the same as for the DateDiff() function. On October 23, 2001, the following WriteLine statements would produce results like those shown in bold:

```
Dim Day1 As DateTime
day1 = Now()
Console.WriteLine(DatePart("yyyy", day1))
     2001
Console.WriteLine(DatePart("q", day1))
     3
Console.WriteLine(DatePart("m", day1))
     10
Console.WriteLine(DatePart("d", day1))
     23
Console.WriteLine(DatePart("w", day1))
     3
Console.WriteLine(DatePart("ww", day1))
     43
Console.WriteLine(DatePart("h", day1))
     15
Console.WriteLine(DatePart("n", day1))
     3
Console.WriteLine(DatePart("s", day1))
     30
```

Financial

The following functions can be used to calculate the parameters of a loan or an investment. I will explain only the functions that return the basic parameters of a loan (such as the monthly payment or the loan's duration). The more advanced financial functions are listed in a table at the end of this section and described in the Visual Basic online documentation.

IPmt(rate, per, nper, pv[, fv[, type]])

This function returns the interest payment for a given period of an annuity based on periodic, fixed payments and a fixed interest rate. The result is a Double value.

The *rate* argument is a Double value specifying the interest rate for the payment period. For example, if the loan's annual percentage rate (APR) is 10 percent, paid in monthly installments, the rate per period is 0.1 / 12 = 0.0083.

The *per* argument is a Double value specifying the current payment period; *per* is a number in the range 1 through *nper*.

The *nper* argument is a Double value specifying the total number of payments. For example, if you make monthly payments on a five-year loan, *nper* is 5 × 12 = 60.

The *pv* argument is a Double value specifying the principal or present value. The loan amount is the present value to the lender of the monthly payments and it's a negative value.

The *fv* argument is a Double specifying the future value or cash balance after the final payment. The future value of a loan is $0 because that's its value after the final payment. If you want to accumulate $10,000 in your savings account over some period of time, however, the future value is $10,000. If the *fv* argument is omitted, 0 is assumed.

The *type* argument specifies when payments are due, and its value can be a member of the DueDate enumeration. Use `DueDate.EndOfPeriod` if payments are due at the end of the payment period; use `DueDate.BegOfPeriod` if payments are due at the beginning of the period. If the *type* argument is omitted, `EndOfPeriod` is assumed.

Suppose you borrow $30,000 at an annual percentage rate of 11.5%, to be paid off in three years with payments at the end of each month. Here's how you can calculate the total interest, as well as the monthly interest:

```
Dim PVal, FVal, mPayments As Integer
Dim APR, iPayment, TotInt As Decimal
PVal = 30000
FVal = 0
APR = 0.115 / 12
mPayments = 3 * 12
Dim period As Integer
For period = 1 To mPayments
   iPayment = IPmt(APR, period, mPayments, -PVal, FVal, 1)
   Console.WriteLine(iPayment)
   TotInt = TotInt + iPayment
Next
Console.WriteLine("Total interest paid: " & TotInt)
```

The interest portion of the first payment is $287.10, and the interest portion of the last payment is less than $10. The total interest is $5,276.

PPmt(rate, per, nper, pv[, fv[, type]])

This function is similar to the IPmt() function except that it returns the principal payment for a given period of a loan based on periodic, fixed payments and a fixed interest rate. For a description of the function's arguments, see the IPmt() entry.

The code for calculating the principal payment of the previous example is nearly the same as that for calculating the interest:

```
Dim PVal, FVal, mPayments As Integer
Dim APR, pPayment, TotPrincipal As Double
PVal = 30000
FVal = 0
APR = 0.115 / 12
mPayments = 3 * 12
Dim period As Integer
For period = 1 To mPayments
   pPayment = PPmt(APR, period, mPayments, -PVal, FVal, 1)
```

```
    Console.WriteLine(pPayment)
    TotPrincipal = TotPrincipal + pPayment
Next period
Console.WriteLine("Total principal paid: " & TotPrincipal)
```

In this example, the principal payments increase with time (that's how the total payment remains fixed). The total amount will be equal to the loan's amount, of course, and the fixed payment is the sum of the interest payment (as returned by the IPmt() function) plus the principal payment (as returned by the PPmt() function).

Pmt(rate, nper, pv[, fv[, type]])

This function is a combination of the IPmt() and PPmt() functions. It returns the payment (including both principal and interest) for a loan based on periodic, fixed payments and a fixed interest rate. For a description of the function's arguments, see the IPmt() entry. Notice that the Pmt() function doesn't require the *per* argument because all payments are equal.

The code for calculating the monthly payment is similar to the code examples in the IPmt() and PPmt() entries.

FV(rate, nper, pmt[, pv[, type]])

This function returns the future value of a loan based on periodic, fixed payments and a fixed interest rate. The arguments of the FV() function are explained in the IPmt() entry, and the *pmt* argument is the payment made in each period.

Suppose you want to calculate the future value of an investment with an interest rate of 6.25%, 48 monthly payments of $180, and a present value of $12,000. Use the FV() function with the following arguments:

```
Dim PVal, FVal As Integer
Dim APR, Payment As Double
Dim TotPmts As Integer
Payment = 180
APR = 6.25 / 100
TotPmts = 48
PVal = 12000
FVal = FV(APR / 12, TotPmts, -Payment, -PVal, DueDate.BegOfPeriod)
MsgBox("After " & TotPmts & " months your savings will be worth $" & FVal)
```

The actual result is close to $25,000.

NPer(rate, pmt, pv[, fv[, type]])

This function returns the number of periods for a loan based on periodic, fixed payments and a fixed interest rate. For a description of the function's arguments, see the IPmt() entry.

Suppose you borrow $25,000 at 11.5%, and you can afford to pay $450 per month. To figure out what this means to your financial state in the future, you would like to know how may years it will take you to pay off the loan. Here's how you can use the NPer() function to do so:

```
Dim PVal, FVal As Integer
Dim APR, Payment As Double
```

```
Dim TotPmts As Integer
FVal = 0
PVal = 25000
APR = 0.115 / 12
Payment = 450
TotPmts = NPer(APR, -Payment, PVal, FVal, DueDate.EndOfPeriod)
If Int(TotPmts) <> TotPmts Then TotPmts = Int(TotPmts) + 1
Console.WriteLine("The loan's duration will be: " & TotPmts & " months")
```

The actual duration of this loan is 80 months, which corresponds to nearly 6.5 years. If the payment is increased from $450 to $500, the loan's duration will drop to 69 months, and a monthly payment of $550 will bring the loan's duration down to 60 months.

Rate(nper, pmt, pv[, fv[, type[, guess]]])

You use this function to figure out the interest rate per payment period for a loan. Its arguments are the same as with the preceding financial functions, except for the *guess* argument, which is the estimated interested rate. If you omit the *guess* argument, the value 0.1 (10%) is assumed.

Table 18 lists and describes the remaining financial functions. All return values are Doubles.

TABLE 18: ADDITIONAL FINANCIAL FUNCTIONS

FUNCTION	RETURNS
PV()	The present value of an investment
NPV()	The net present value of an investment based on a series of periodic cash flows and a discount rate
IRR()	The internal rate of return for an investment
MIRR()	The modified internal rate of return for a series of periodic cash flows
DDB()	The depreciation of an asset for a specific time period using the double-declining balance method or some other method you specify
SYD()	The sum-of-years' digits depreciation of an asset for a specified period
SLN()	The straight-line depreciation of an asset for a single period

File I/O

An important aspect of any programming other language, is its ability to access and manipulate files. Visual Basic supports three types of files:

- Sequential
- Random-access files
- Binary files

Sequential files are mostly text files (the ones you can open with a text editor such as Notepad). These files store information as it's entered, one byte per character. Even the numbers in a sequential file are stored as string and not as numeric values (that is, the numeric value 33.4 is not stored as a Single or Double value, but as the string "33.4"). These files are commonly created by text-processing applications and are used for storing mostly text, not numbers.

Sequential files are read from the beginning to the end. Therefore, you can't read and write at the same time to a sequential file. If you must read from and write to the file simultaneously, you must open two sequential files, one for reading from and another one for writing to.

If your application requires frequent access to the file's data (as opposed to reading all the data into memory and saving them back when it's done), you should use random-access files. Like the sequential files, random-access files store text as characters, one byte per character. Numbers, however, are stored in their native format (as Integers, Doubles, Singles, and so on). You can display a random-access file in a DOS window with the TYPE command and see the text, but you won't be able to read the numbers.

Random-access files are used for storing data that are organized in segments of equal length. These segments are called *records.* Random-access files allow you to move to any record, as long as you know where the desired record is located. Since all records have the same length, it's easy to locate any record in the file by its index. Moreover, unlike sequential files, random-access files can be opened for reading and writing at the same time. If you decide to change a specific record, you can write the new record's data on top of the old record, without affecting the adjacent records.

Binary files, finally, are similar to sequential files, and they make no assumption as to the type of data stored in them. The bytes of a binary file can be characters, or the contents of an executable file. Images, for instance, are stored in binary files.

The manipulation of files is more or less independent of its type and involves three stages:

Opening the file The operating system reserves some memory for storing of the file's data. If the file does not exist, it's first created and then opened. To open a file (and create it if necessary), use the FileOpen() function.

Processing the file A file can be opened for reading from, writing to, or reading and writing. Data are read, processed, and then stored back to the same, or to another, file.

Closing the file When the file is closed, the operating system releases the memory reserved for the file. To close an open file, use the FileClose() function.

In the following sections, we'll look at Visual Basic's file-manipulation functions. The .NET Framework provides high-level functions for accessing files, and you should use these functions in your projects. The file I/O functions provided by the Framework are discussed in Chapter 13.

FreeFile(file_number)

During the course of an application, you may open and close many files, and you may not always know in advance which file numbers are available. Visual Basic provides the FreeFile() function, which returns the next available file number. The FreeFile() function is used in conjunction with the FileOpen() function to open a file:

```
fNum = FreeFile()
FileOpen(fNum, fileName)
```

After these two statements execute, all subsequent commands that operate on the specified file can refer to it as *fNum*. The FreeFile() function returns the next available file number, and unless this number is assigned to a file, FreeFile() returns the same number if called again. The following statements will not work:

```
fNum1 = FreeFile()
fNum2 = FreeFile()        ' WRONG !
FileOpen(fNum1, file1)
FileOpen(fNum2, file2)
```

Each time you call FreeFile() to get a new file number, you must use it. The statements should have been coded as follows:

```
fNum1 = FreeFile()
FileOpen(fNum1, file1)
fNum2 = FreeFile()
FileOpen(fNum, file2)
```

FileOpen(number, path, mode[, access][, share][, recordLen])

To use a file, you must first open it—or create it, if it doesn't already exist. The FileOpen() function, which opens files, accepts a number of arguments, most of which are optional.

The FileOpen() function replaces the Open statement of VB6, which is no longer supported in VB.NET.

The argument *path* is the path of the file to be opened, and *number* is a number you assign to the file (usually through the FreeFile function). The *mode* argument determines the mode in which the file will be opened and can be one of the constants shown in Table 19.

TABLE 19: THE OPENMODE ENUMERATION

VALUE	DESCRIPTION
`Input`	File is opened for input (reading from) only.
`Output`	File is opened for output (writing to) only.
`Append`	File is opened to appending new data to its existing contents.
`Random`	File is opened for random access (read or write one record at a time).
`Binary`	File is opened in binary mode.

The first three file modes refer to sequential files. `Random` is used with random-access files, and `Binary` is used with binary files. When you open a sequential file, you can't change its data. You can either read them (and store them to another file) or overwrite the entire file with the new data. To do so, you must open the file for `Input`, read its data, and then close the file. To overwrite it, open it again (this time for `Output`) and save the new data to it.

If you don't want to overwrite an existing file, but just to append data to it (without changing any of the existing data), open it for `Append`. If you open a file for `Output`, Visual Basic wipes out its contents, even if you don't write anything to it. Moreover, VB won't warn you that it's about to overwrite a file, as applications do. This is how the FileOpen() function works and you can't change your mind after opening a sequential file for `Output`.

The *access* argument determines whether the file can be opened for reading from (`Read`), writing to (`Write`), or both (`ReadWrite`). If you open a file with `Read` access, your program can't modify it even by mistake. The access method has nothing to do with file types. Sequential files are open for `Input` or `Output` only, because they can't be opened in both modes. The access type is specified for reasons of safety. If you need to open a file only to read data from it, open it with `Read` access (there's no reason to risk modifying the data).

The *share* argument allows you to specify the rights of other Windows applications, while your application keeps the file open. Under Windows, many applications can be running at the same time, and one of them may attempt to open a file that is already open. In this case, you can specify how other applications are to access the file. The *share* argument can have one of the values listed in Table 20.

TABLE 20: THE OPENACCESS ENUMERATION

VALUE	DESCRIPTION
`Shared`	Other applications can share the file.
`LockRead`	The file is locked for reading.
`LockWrite`	The file is locked for writing.
`LockReadWrite`	Other applications can't access this file.

File locking is a very important function, especially in a networked environment. Imagine two users attempting to write to the same file at the same time. Using the file-locking features, you can write programs that work properly in networked environments, too. However, if you are going to build applications that will be run by many users who access the same files, you should probably consider building a database.

Finally, if the file is a random-access one, you must declare the length of the record with the last argument, which is the record's length in bytes. When you create a random-access file, Visual Basic doesn't record any information regarding the record's length, or structure, to the file. You should know, therefore, the structure of each record in a random-access file before you can open it. The record's length is the sum of the bytes taken by all record fields. You can either calculate it, or you can use the function `Len(record)` to let Visual Basic calculate it. The *record* argument is the name of the structure you use with the random-access file.

The following command opens the file `c:\samples\vb\cust.dat` as a sequential file with a number obtained through the FreeFile() function:

```
Dim Fnum As Integer = FreeFile()
FileOpen(Fnum, "c:\samples\vb\cust.dat", OpenMode.Output, OpenAccess.ReadWrite )
```

FileClose(file_number)

The FileClose() function closes an open file, whose number is passed as argument. The statement

```
FileClose(fNum1)
```

closes the file opened as *fNum1*.

Reset()

The Reset() function closes all files opened with FileOpen(). Use this statement to close all the files opened by your application.

EOF(file_number), LOF(file_number)

These are two more frequently used functions in file manipulation. The EOF() function accepts as an argument the number of an open file and returns True if the end of the file (EOF) has been reached. The LOF() function returns the length of the file, whose number is passed as argument.

You use the EOF() function to determine whether the end of the file has been reached, with a loop such as the following:

```
{get first record}
While Not EOF(fNum)
   { process current record }
   { get next record }
End While
```

With the help of the LOF() function, you can also calculate the number of records in a random-access file:

```
Rec_Length = LOF(file_number) / Len(record)
```

Print (file_number, output_list), PrintLine(file_number, output_list)

The Print() function writes data to a sequential file. The first argument is the number of the file to be written, and the following arguments are the values, or variables, to be written to the file. The second argument is a parameter array, which means you can pass any number of values to the function:

```
Print(fNum, var1, var2, "some literal", 333.333)
```

The Print() function doesn't insert line breaks between successive calls. The following statements write a single line of text to the file opened as *fNum*:

```
Print(fNum, "This is the first half of the line ")
Print(fNum, "and this is the second half of the same line.")
```

The PrintLine() function does the same thing as the Print() function, but it also adds a new line character at the end of the values it writes to the file. Its syntax is identical to the syntax of the Print() function. Multiple values are separated by commas, and each comma specifies that the next character will be printed in the next *print zone*. Each print zone corresponds to 14 columns. In other words, the Print() function writes data to the file exactly as the TYPE command (of DOS) displays

them on the screen. (That's why the data saved by the Print() function are called *display-formatted* data.) You must keep in mind that the text will be displayed correctly only when printed with a monospaced typeface, such as Courier. If you place the text on a TextBox with a proportional typeface, the columns will not align.

Data saved with the Print() function can be read with the LineInput() and Input() functions. However, isolated fields are not delimited in any way, and you must extract the fields from the line read. The Print() function is used to create text files that can be viewed on a DOS window. To format the fields on each line, you can use the Tab to position the pointer at the next print zone, or `Tab(n)` to position the pointer at an absolute column number. The following statements create a text file:

```
On Error Resume Next
Kill("c:\test.txt")
Dim fNum As Integer = FreeFile()
FileOpen(fNum, "c:\test.txt", OpenMode.Output)
PrintLine(fNum, "John", TAB(12), "Ashley", TAB(25), "Manager", TAB(45), 33)
PrintLine(fNum, "Michael", TAB(12), "Staknovitch", TAB(25), "Programmer", _
          TAB(45), 28)
PrintLine(fNum, "Tess", TAB(12), "Owen", TAB(25), "Engineer", TAB(45), 41)
PrintLine(fNum, "Joe", TAB(12), "Dow", TAB(25), "Administrator", TAB(45), 25)
PrintLine(fNum, "************************")
PrintLine(fNum, "John", TAB, "Ashley", TAB, "Manager", TAB, 3)
PrintLine(fNum, "Michael", TAB, "Staknovitch", TAB, "Programmer", TAB, 28)
PrintLine(fNum, "Tess", TAB, "Owen", TAB, "Engineer", TAB, 41)
PrintLine(fNum, "Joe", TAB, "Dow", TAB, "Administrator", TAB, 25)
FileClose(fNum)
```

This is the output produced by this example:

```
John       Ashley       Manager             33
Michael    Staknovitch  Programmer          28
Tess       Owen         Engineer            41
Joe        Dow          Administrator       25
************************
John          Ashley         Manager        33
Michael       Staknovitch    Programmer     28
Tess          Owen           Engineer       41
Joe           Dow            Administrator  25
```

Input(file_number, var)

The Input() function reads data from a sequential file and assigns them to the variable passed with the second argument The following lines read two values from the open file, a numeric value and a date:

```
Dim numVal As Long, DateVal As Date
Input(1, numVal)
Input(1, DateVal)
```

LineInput(file_number)

To read from sequential files, use the LineInput() function. The *file_number* argument is the file's number, and the function returns the next text line in the file. This statement reads all the characters from the beginning of the file to the first newline character. When you call it again, it returns the following characters, up to the next newline character. The newline characters are not part of the information stored to or read from the file, and they are used only as delimiters. If we close the file of the last example and open it again, the following lines will read the first two text lines and assign them to the string variables *Line1* and *Line2:*

```
Line1 = LineInput(fNum)
Line2 = LineInput(fNum)
```

If you want to store plain text to a disk file, create a sequential file and store the text there, one line at a time. To read it back, open the file and read one line at a time with the LineInput() function, or use the FileGet() function to read the entire text.

FilePut(file_number, value[, record_number]), FileGet(file_number, value[, record_number])

These functions are used for writing records to and reading records from a random-access file. Both functions need know the record number you want to access (write or read).

The *record_number* argument is the number of the record we are interested in, and *value* is a record variable that is written to the file. The *record_number* argument is optional; if you omit it, the record will be written to the current record position. After a record is written to or read from the file, the next record becomes the current one. If you've read the second record, the FilePut() function will store the field values in the third record in the file. If you call FilePut() 10 times sequentially without specifying a record number, it will create (or overwrite) the first ten records of the random-access file.

The arguments of the FileGet() function have the same meaning.

At this point, I'll outline the basics of random-access file manipulation, since this is the most flexible file type. Let's say you want to create a random-access file for storing a product list. Each product's information is stored in a *ProductRecord* variable, whose declaration is shown next:

```
Structure ProductRecord
   ProductID As String
   Description As String
   Price As Decimal
End Type
```

The structure *ProductRecord* will be used for storing each product's information before moving it to the file. Let's start by defining a variable of type *ProductRecord*:

```
Dim PRec As ProductRecord
```

You can then assign values to the fields of the *PRec* variable with statements such as the following:

```
PRec.ProductID = "TV00180-A"
PRec.Description = "SONY Trinitron TV"
PRec.Price = 799.99
```

The *PRec* record variable can be stored to a random-access file with the FilePut() function. Of course, you must first create the file with the following statements:

```
fNum = FreeFile()
FileOpen(fNum, "c:\products.dat",OpenMode.Random)
```

You will notice that I've skipped the last argument, which is the length of the record. Since our record contains strings, it has a variable length, so we'll let the function handle the records. (Each string's length is stored along with the string and you need not worry about the actual length of each record.) You can then write the *PRec* variable to the file with the statement

```
FilePut(fNum, Prec)
```

Notice that you can omit the number of the record where the data will be stored. You can change the values of the fields and keep storing additional records with the same Put statement (as long as *PRec* is populated with different field values). After all the values are stored to the file, you can close the file with this statement:

```
FileClose(fNum)
```

To read the records, open the file with the same FileOpen() function you used to open it for saving the records:

```
fNum = FreeFile()
FileOpen(fNum, "c:\products.dat", OpenMode.Random)
```

You can then set up a loop to read the records.

The following code segment demonstrates how to write records of different lengths to a random file with the FilePut() function and read them with the FileGet() function. First, insert the following structure declaration somewhere on the form's declarations section:

```
Structure ProductRecord
   Dim ProductID As String
   Dim Description As String
   Dim Price As Decimal
End Structure
```

Then enter the following statements in a button's Click event handler:

```
Dim PRec As ProductRecord
PRec.ProductID = "TV00180-A"
PRec.Description = "SONY Trinitron TV"
PRec.Price = 799.99

fNum = FreeFile()
FileOpen(fNum, "c:\products.dat", OpenMode.Random)
FilePut(fNum, PRec)

PRec = New ProductRecord()
PRec.ProductID = "TV-RCA"
PRec.Description = "This is an RCA Trinitron TV"
PRec.Price = 699.99
FilePut(fNum, PRec)
```

```
PRec = New ProductRecord()
PRec.ProductID = "TV810X"
PRec.Description = "And this is the real cheap BIG Trinitron TV"
PRec.Price = 399.99
FilePut(fNum, PRec)

FileClose(fNum)

fNum = FreeFile()
FileOpen(fNum, "c:\products.dat", OpenMode.Random)
PRec = New ProductRecord()
FileGet(fNum, PRec, 2)
FileClose(fNum)

Console.WriteLine(PRec.ProductID)
Console.WriteLine(PRec.Description)
Console.WriteLine(PRec.Price)
```

As you can see, the IDs and descriptions of the various products are strings of different lengths. The first segment of the code writes three records to the random file and then closes it. The last part of the code reads the second record and prints its fields to the Output window. In previous versions of VB, each record in a random-access file had to have the same length. The new functions allow you to create records with strings, which inherently are records of variable length. The Put() and Get() functions handle all the details, and you can access the random-access file using the record as the basic unit of length.

Write(file_number, output_list), WriteLine(file_number, output_list)

The Write() function writes data to a sequential file. The data to be written are supplied in the *output_list*, which is a comma-separated list of variables and literals. Data written with the Write() function are usually read with the Input function. The following line will write a numeric and a date value to a sequential file:

```
NumVal = 3300.004
DateVal = #04/09/1999#
Write(1, NumVal, DateVal)
```

The WriteLine() function does the same, but it also inserts a newline character at the end of each line of data.

The following lines write the same data as the example of the FilePrint() function, explained earlier in the entry of the Print() function.

```
Dim fNum As Integer = FreeFile()
FileOpen(fNum, "c:\test.txt", OpenMode.Output)
WriteLine(fNum, "John Ashley", 33, "Manager")
WriteLine(fNum, "Michael Staknovitch", 24, "Programmer")
WriteLine(fNum, "Tess Owen", 37, "Engineer")
WriteLine(fNum, "Joe Dow", 28, "Administrator")
```

The structure of the text file, however, is quite different. Here's the output of the Write() function:

```
"John Ashley",33,"Manager"
"Michael Staknovitch",24,"Programmer"
"Tess Owen",37,"Engineer"
"Joe Dow",28,"Administrator"
```

Seek(file_number[, position]), Loc(file_number)

The Loc() function returns the current read/write position in a file. The Seek() function does the same if called without the *position* argument. For a random-access file, the value returned by either function is the number of the last record read from, or written to, the file. For sequential files, this value is the current byte divided by 128. For binary files, it's the number of the last byte read from, or written to, the file. If you specify the position argument of the Seek() function, you can set the current read/write position in the file. To move to the beginning of the third record in a random access file, use a statement like the following:

```
Seek(fNum, 3)
```

Lock(file_number[, fromRecord][, toRecord]), Unlock(file_number[, fromRecord][, toRecord])

The Lock() function allows you to lock a file or some of the records in a random-access file. The locked records are not available to other applications that are currently running. Your application, however, has access to the entire file. If another application attempts to open a locked file or to access one of the locked records, Visual Basic will generate a runtime exception.

If you omit the optional arguments, then the entire file is locked. If you specify the *fromRecord* argument, then all following records are locked. Finally, you can lock a range of records by specifying both optional arguments.

Width(fNum, length)

This is another useful statement that applies to sequential files only. The Width() function sets the maximum line length that can be written to a file. The maximum line length is specified by the second argument, *length*. A line with fewer characters than *length* is stored to the file as is. Longer lines are broken; Visual Basic automatically inserts newline characters to enforce the specified maximum line length. Use this function with the Print() and Write() functions, which append data to the same line, to limit the length of each data line. Even better, you should use the PrintLine() and WriteLine() methods to control how much information goes to the same line.

FileAttr(file_number)

The FileAttr() function returns an integer representing the file mode for files opened using the FileOpen() function. The *file_number* argument is the number of the file. The value returned is one of those in Table 21.

TABLE 21: VALUES RETURNED BY THE FILEATTR() FUNCTION

VALUE	MODE
1	Input
2	Output
4	Random
8	Append
32	Binary

Random-Number Generation

Visual Basic .NET supports the random-number generator of VB6, but it also provides a class for generating random numbers, the `System.Random` class.

Rnd([seed])

This function returns a pseudo-random number in the range 0 to 1. The optional argument is called a *seed* and is used as a starting point in the calculations that generate the random number.

NOTE *The sequence of random numbers produced by Visual Basic is always the same! Let's say you have an application that displays three random numbers. If you stop and rerun the application, the same three numbers will be displayed. This is not a bug. It's a feature of Visual Basic that allows you to debug applications that use random numbers (if the sequence were different, you wouldn't be able to re-create the problem). To change this default behavior, call the Randomize statement at the beginning of your code. This statement will initialize the random-number generator based on the value of the computer's Timer, and the sequences of random numbers will be different every time you run the application.*

If *seed* is negative, the Rnd() function always returns the same sequence of random numbers. As strange as this behavior may sound, you may need this feature to create repeatable random numbers to test your code. If *seed* is positive (or omitted), the Rnd() function returns the next random number in the sequence. Finally, if *seed* is zero, the Rnd() function returns the most recently generated random number.

In most cases, you don't need a random number between 0 and 1, but between two other integer values. A playing card's value is an integer in the range 1 through 13. To simulate the throw of a dice, you need a number in the range 1 through 6. To generate a random number in the range *lower* to *upper*, in which both bounds are integer numbers, use the following statement:

```
randomNumber = Int((upper - lower + 1) * Rnd() + lower)
```

The following statement displays a random number in the range 1 to 49:

```
Console.WriteLine(Int(Rnd() * 49 + 1))
```

Randomize [seed]

The Randomize statement initializes the random-number generator. The *seed* argument is a numeric value, used to initialize the random-number generator. To create a different set of random numbers every time the application is executed, use the current date as seed. However, using the same seed will not return the same run of random numbers.

Graphics

This section discusses the two Visual Basic functions for color definition. The LoadPicture() and SavePicture() functions, as well as the drawing statements of VB6, are no longer supported by VB.NET.

QBColor(color)

This function returns an Integer representing the RGB color code corresponding to the specified color number. The *color* argument is a number in the range 0 through 15. Each value returns a different color, as shown in Table 22.

TABLE 22: VALUES FOR THE *COLOR* ARGUMENT

VALUE	COLOR
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Yellow
7	White
8	Gray
9	Light Blue
10	Light Green
11	Light Cyan
12	Light Red
13	Light Magenta
14	Light Yellow
15	Bright White

Use the QBColor() function to specify colors if you want to address the needs of users with the least-capable graphics adapter (one that can't display more than the basic 16 colors). Also use it for business applications that don't require many colors.

RGB(red, green, blue)

This function returns an Integer representing a color value. The *red*, *green*, and *blue* arguments are integer values in the range 0 through 255, representing the values of the three basic colors. Table 23 lists some of the most common colors and their corresponding red, green, and blue components. The colors correspond to the eight corners of the RGB color cube.

TABLE 23: COMMON COLORS AND THEIR CORRESPONDING RGB COMPONENTS

COLOR	RED	GREEN	BLUE
Black	0	0	0
Blue	0	0	255
Green	0	255	0
Cyan	0	255	255
Red	255	0	0
Magenta	255	0	255
Yellow	255	255	0
White	255	255	255

The statement

```
Text1.BackColor = RGB(255, 0, 0)
```

assigns a pure red color to the background of the *Text1* control.

Registry

Visual Basic provides a few special functions for storing values in the Registry. These functions are safer than manipulting the Registry directly, and they access only a single branch of the Registry (in other words, you can't ruin by mistake the branch of another application).

SaveSetting(appname, section, key, setting)

This function stores a new setting in the Registry or updates an existing one. The *appname* argument is the name of the application (or project) that stores the information in the Registry. It doesn't have to be the actual name of the application; it can be any string you supply, as long as it's unique for your application. The *section* argument is the name of the Registry section in which the key setting will be saved. The *key* argument is the name of the key setting that will be saved. The last argument,

setting, is the value of the key to be saved. If *setting* can't be saved for any reason, a runtime trappable error is generated.

The following statements store the keys "Left" and "Top" in the Startup section of the application's branch in the Registry:

```
SaveSetting("MyApp", "Startup", "Top", Me.Top)
SaveSetting("MyApp", "Startup", "Left", Me.Left)
```

These values should be saved to the Registry when the application ends, and they should be read when it starts, to place the form on the desktop. You can start the Registry Editor utility (select Start ➢ Run and type `regedit`) and search for the string "MyApp." When the corresponding branch in the Registry is found, you will see how the keys "Top" and "Left" were stored there.

DeleteSetting(appname, section[, key])

This function deletes a section or key setting from an application's entry in the Windows Registry. Its arguments are the same as the ones by the same name of the SaveSetting function except for the last one (the key's setting). If the last argument is omitted, then all the keys in the specified section are deleted. To remove the two keys added with the sample statements in the previous entry, use the following statements:

```
DeleteSetting("MyApp", "Startup", "Top")
DeleteSetting("MyApp", "Startup", "Left")
```

Or delete them both with a single call to the DeleteSetting function:

```
DeleteSetting("MyApp", "Startup")
```

GetSetting(appname, section, key[, default])

This function returns a key setting from an application's branch in the Registry. The arguments *appname*, *section*, and *key* are the same as in the previous entries. The last argument, *default*, is optional and contains the value to return if no value for the specified key exists in the Registry.

To read the key values stored in the Registry by the statements in the SaveSetting() entry, use the following code segment:

```
Me.Top = GetSetting("MyApp", "Startup", "Top", 100)
Me.Left = GetSetting("MyApp", "Startup", "Left", 150)
```

Don't omit the default values here, because the form may be sized oddly if these keys are missing.

GetAllSettings(appname, section)

This function returns a list of keys and their respective values from an application's entry in the Registry. The *appname* argument is the name of the application (or project) whose key settings are requested. The *section* argument is the name of the section whose key settings are requested. The GetAllSettings() function returns all the keys and settings in the specified section of the Registry in a two-dimensional array. The element (0,0) of the array contains the name of the first key, and the elements (0,1) contains the setting of this key. The next two elements (1,0) and (1,1) contain the key and setting of the second element, and so on. To find out how many keys are stored in the specific section of the Registry, use the Length property of the array.

The following statement retrieves all the keys in the Startup section for the *MyApp* application and stores them in the array *AllSettings*:

```
AllSettings = GetAllSettings("MyApp", "Startup")
```

You can then set up a loop that scans the array and displays the key and setting pairs:

```
For i = 0 To AllSettings.GetUpperBound(0)
   Console.WriteLine(AllSettings(i, 0) & " = " & AllSettings(i, 1) )
Next
```

Application Collaboration

In this section I present the Shell() function, which allows you to start another application from within your VB application. Once the application has been started successfully, you can activate its window from within another application. This is a rude way of automating another application, but if the application you want to automate doesn't support VBA, the Shell() function and related statements are the only options.

Shell(path_name[, style][, wait][, timeout])

This function starts another application and returns a value representing the program's task ID if successful; otherwise, it returns zero. The *path_name* argument is the full path name of the application to be started and any arguments it may expect. The optional argument *style* determines the style of the window in which the application will be executed, and it can have one of the values shown in Table 24.

TABLE 24: THE APPWINSTYLE ENUMERATION

VALUE	DESCRIPTION
`Hide`	The window is hidden, and focus is passed to it.
`NormalFocus`	The window has the focus and is restored to its original size and position.
`MinimizedFocus`	The window is displayed as an icon that has the focus.
`MaximizedFocus`	The window is maximized and has the focus.
`NormalNoFocus`	The window is restored to its most recent size and position. The currently active window remains active.
`MinimizedNoFocus`	The window is displayed as an icon. The currently active window remains active.

The *wait* argument is a True/False value indicating whether the calling application should wait for the shelled process to terminate or not. The default value is False. By default, the Shell() function runs other programs asynchronously. This means that a program started with Shell() might not finish executing before the statements following the Shell() function are executed. The last argument is the number of milliseconds you're willing to wait for the shelled process to terminate. If the specified number of milliseconds elapses, then a timeout exception will occur. The default value of the *timeout* argument is –1, which means that no timeout will ever occur.

To start Notepad from within your VB application, use the following statement:

```
NPAD = Shell("notepad.exe", AppWinStyle.NormalFocus)
```

Notice that you need not specify the path name of the executable file, if it's on the path (the `NOTEPAD.EXE` file resides in the Windows folder). The NPAD value identifies the specific instance of the Notepad application, and you can use it from within your code to activate the external application.

The SendKeys statement of VB6 is not supported by VB.NET.

AppActivate(title[, wait])

This function (which does not return a value) lets you activate an application that you started previously from within your VB code with the Shell() function. The first argument, *title*, specifies the title of the application (as it appears in the titlebar of the application's window). You can also use the ID of the application returned by the Shell() function. The *wait* argument is optional; it's a Boolean value that determines whether the calling application has the focus before activating another. If it's False (the default value), the specified application is immediately activated, even if the calling application does not have the focus. If it's True, your application must wait until it gets the focus before it can activate the external application. The AppActivate statement changes the focus to the named application or window but does not affect whether it is maximized or minimized.

If you're using the application's title to activate it, make sure you provide enough information to make it unique. Word's title is "Microsoft Word" followed by the name of the active document. If you specify only the string "Microsoft Word" and multiple instances of Word are running, you don't know which one will be activated by the AppActivate statement. It is best to use the ID of the application you started with the Shell() function.

Option Statements

The Option statements let you specify options (like the subscripting of arrays and sorting order) in a module. The Option statements appear at the beginning of a module and affect the code in that specific module.

Option Compare

This statement, which takes effect in the module in which it appears, determines how Visual Basic will perform string comparisons. While comparing numeric values is straightforward and there's no question as to the order of numeric values, things aren't as trivial with strings. String comparisons can be case-sensitive or case-insensitive, and the Compare option can be set to one of the following two values:

Option Compare Binary Sorts strings based on the internal binary representation of the characters. With the Binary sort option, all uppercase characters come before the lowercase characters, and foreign symbols are at the end, again with uppercase characters ahead of lowercase characters:

A < B < C … < Z < a < b < c … < z < À < Á < Â < …<à < á <â …

Option Compare Text Sorts strings using case-sensitive order (that is, A = a < À = à < … B = b < … so on). This is the natural sort order for names.

Option Explicit

This statement tells the compiler to check each variable in the module before using it and to issue an error message if you attempt to use a variable without having previously declared it. If you decide to declare all variables in your projects (to avoid excessive use of variants), you can ask Visual Basic to insert the `Option Explicit` statement automatically in every module by setting the Option Explicit item to On in the Build tab of the Project Property Pages.

When `Option Explicit` appears in a module, you must explicitly declare all variables. If you don't use the `Option Explicit` statement, all undeclared variables are of Object type.

Option Strict

The Explicit option forces you to declare all variables before using them in your code, but it doesn't force every variable to have a specific type. The Strict option requires that all variables be declared with a specific type. Moreover, when the Strict option is on, you must explicitly convert between data types. The defult value of this Strict option is off and you can turn it on in individual files, or turn it on for all the files in a project from within the Build tab of the Project Property Pages.

Miscellaneous

This section describes the functions and statements that don't fit in any other category.

Choose(index, choice1[, choice2, …])

The Choose() function selects and returns a value from a list of arguments. The *index* argument is a numeric value between 1 and the number of available choices. The following arguments, *choice1*, *choice2*, and so on, are the available options. The function will return the first choice if *index* is 1, the second option if index is 2, and so on.

One of the uses of the Choose() function is to translate single digits to strings. The function IntToString() returns the name of the digit passed as an argument:

```
Function IntToString(int As Integer) As String
   IntToString  = Choose (int + 1, "zero", "one", "two", "three", _
                          "four", "five", "six", "seven", "eight", "nine")
End Function
```

If *index* is less than one or larger than the number of options, the Choose() function returns a Null value. To test the IntToString() function, call it with a statement like the following:

```
MsgBox(IntToString(8))
```

IIf(expression, truepart, falsepart)

This function returns one of two parts, depending on the evaluation of *expression*. If the *expression* argument is True, the *truepart* argument is returned. If *expression* is not True, the *falsepart* argument is returned. The IIf() function is equivalent to the following If clause:

```
If expression Then
   result = truepart
Else
   result = falsepart
End If
```

In many situations, this logic significantly reduces the amount of code. The Min() and Max() functions, for instance, can be easily implemented with the IIf() function:

```
Min = IIf(a < b, a, b)
Max = IIf(a > b, a, b)
```

Switch(expression1, value1, expression2, value2, ...)

This function evaluates a list of expressions and returns a value associated with the first expression in the list that happens to be True. If none of the expressions is True, the function returns Null. The following statement selects the proper quadrant depending on the signs of the variables *X* and *Y*:

```
Quadrant = Switch(X>0 and Y>0, 1, X<0 and Y>0, 2, X<0 and Y<0, 3, X<0 and Y<0, 4)
```

If both *X* and *Y* are negative, the *Quadrant* variable is assigned the value 1. If *X* is negative and *Y* is positive, *Quadrant* becomes 2, and so on. If either *X* or *Y* is zero, none of the expressions are True, and *Quadrant* becomes Null.

Environ()

This function returns the environment variables (operating system variables set with the SET command). To access an environment variable, use a numeric index or the variable's name. If you access the environment variables by index, as in

```
Console.WriteLine(Environ(2))
```

you'll get a string that contains both the name of the environment variable and its value, such as:

```
TMP=C:\WINDOWS\TEMP
```

To retrieve only the value of the TMP environment variable, use the expression

```
Console.WriteLine(Environ("TMP"))
```

and the function will return the value:

```
C:\WINDOWS\TEMP
```

If you specify a nonexistent environment variable name, a zero-length string ("") is returned.

Beep

This statement sounds a tone of short duration through the computer's speaker. The pitch and duration of the beep depend on the target computer and can't be adjusted. This is the simplest form of audio warning you can add to your application, as it doesn't require a sound card.

CallByName(object, procedurename, calltype[, arguments()])

This function executes a method of an object, or sets/returns an object property. The *object* argument is the name of the object, whose method will be called or whose property will be set or read. *procedurename* is the name of a property or method of the object, on which the function will act. The *calltype* argument specifies the type of procedure being called and can have one of the values shown in Table 25.

Table 25: The CallType Enumeration

Constant	Definition
Get	Reads a property value
Set	Sets a property value
Method	Calls a method of the object

The last argument is an array, which contains the arguments required by the method, or the value of the property to be set.

To set the Text property of the *Text1* control, you'd write a statement like:

```
TextBox1.Text = "Welcome to VB.NET"
```

You can call the CallByName() function to set the same property to the same value as follows:

```
CallByName(TextBox1, "Text", CallType.Set, "Welcome to VB.NET")
```

Notice that the first argument is the actual object, not a string variable with the name of the object. The statements

```
ObjectName = "TextBox1"
CallByName ObjectName, "Text", CallType.Set, "Welcome to VB.NET"
```

will generate a runtime error. If you want to use a variable, you must first set it to an existing object, as shown next:

```
Dim MyObject As Object
MyObject = TextBox1
CallByName MyObject, "Text", CallType.Set, "Welcome to VB.NET"
```

To retrieve the value of the Text property of the same control, use the following statement:

```
MyText = CallByName(TextBox1, "Text", CallType.Get)
```

The CallByName() function can be used to invoke a method of an object.

It's simpler to invoke methods and set or read property values with the notation ojbect.property or object.method. There may be situations where you must call one of several methods that are not known at design time, and this is when the CallByName() function will come in handy.