Appendix B: Visual Basic .NET Quick Start

Major Topics

B.1 About Visual Basic .NET
B.2 A First Program
   Hands-On Tutorial: Pet Adoption Program
B.3 Variables and Data Types
   Dates and Times
B.4 Using Interactive Controls
   Form Properties and Methods
   Label Control
   TextBox Control
   CheckBox Control
   Button Control
   Tutorial: Pet Adoption Registry
   Events and Handlers
B.5 Using Text Boxes and Labels
   Date String Example
   Formatting Numeric Data
B.6 Commission Calculator Example
   Version 1
   Version 2: Adding a Salary Advance
   Version 3: Varying the Commission Rate
B.7 Windows Console Programs
B.8 Loops
B.9 Conditional Statements
   If-Then
   If-Else-End If
   Nested-If
B.10 Arrays
    Declaring an Array
    Array of Objects
    Arrays as Method Parameters
    Two-Dimensional Arrays
B.1 ABOUT VISUAL BASIC .NET

Visual Basic .NET provides a powerful, yet safe environment for creating MS-Windows applications. In earlier times, you had to be a C++ wizard to create high-performance Windows applications, and you had to know a great deal about the Win32 API (Application Programmers Interface).

Previous versions of Visual Basic were easy to learn, but lacked the full power of object-oriented languages. The current version has all the power of languages like C++ and Java, and retains the drag-and-drop developers interface that has made Visual Basic so popular.

Rapid Application Development (RAD) is the technique of designing, coding, and testing applications in a constantly repeating sequence. RAD developers design and code interactively, producing results in a surprisingly short time. Visual Basic .NET is an ideal tool for RAD.

The Visual Studio .NET Environment, also known as the IDE (Integrated Developer Environment), provides all the tools you need to write and test applications. Here are some of the main tools:

- Visual controls for creating user interfaces in both Windows Desktop and ASP.NET (Web) applications. They are found in the ToolBox window.
- Database connection components, also found in the ToolBox window.
- Solution Explorer: A window that lists all files in your project.
- Server Explorer: A window that you use to create and manage database connections and database tables.
- Properties Window: Displays the properties of a form, component, or control.
- Interactive Debugger: You can single-step through your program's code, set breakpoints, and inspect variables.

Forms and Code

Visual interfaces are created using Forms (Windows or Web). In Windows applications, the code file creates instances of all the visual controls on the form. You can write additional code that responds to runtime events. You can also declare your own classes, and add methods to forms.

In Web applications, your visual interface is controlled by HTML code inside the Web form file (extension .aspx). There is a separate codebehind file that contains your runtime event handler code, objects, and classes. It's extension is .aspx.vb.

Types of Windows Desktop Applications

You can create two basic types of Windows Desktop applications in Visual Basic:

- **Windows Application**: Displays a graphical user interface, similar to all standard MS-Windows applications.
- **Console Application**: All output is plain text, written to a single command window, identical to the one shown when you run the cmd.exe program from the Windows Start/Run menu.
- **ASP.NET Web Application**: All output displays in a Web Browser window.
B.2 A FIRST PROGRAM

Windows Controls

*Windows Controls* are both visual and non-visual objects that provide any sort of service or function in a program. Examples are controls that display text and graphics, controls that let the user input text, a timer control that acts as a stopwatch, and a control that displays a list of items. If we're working on a Windows Desktop application, we just call them *controls*.

Every control has a **Name** property, which is usually set in design mode (before the program runs). The names assigned to controls should be explanatory, particularly when you plan to reference them in your program code. Following are a few example controls, along with their standard name prefixes and an example of a typical name:

<table>
<thead>
<tr>
<th>Control Type</th>
<th>Name Prefix</th>
<th>Name Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label</td>
<td>lbl</td>
<td>lblEmployeeName</td>
</tr>
<tr>
<td>TextBox</td>
<td>txt</td>
<td>txtAge</td>
</tr>
<tr>
<td>Button</td>
<td>btn</td>
<td>btnOK</td>
</tr>
<tr>
<td>PictureBox</td>
<td>pbx</td>
<td>pbxAnimal</td>
</tr>
</tbody>
</table>

As we introduce more controls, you'll see their common name prefixes.

Form Objects

Also known as a System.Windows.Forms.Form object, a *form* is a container for both code and controls. When you add a form to a program, a class definition for that form is created automatically by Visual Studio .NET. Figure B-1 shows a blank form as it would be visible in *design mode* (when the program is not running). Figure B-2 shows the same form as it appears when its host program is running.

*Figure B-1*  Blank form, in design mode.
When you create a Windows Desktop application, a single form named **Form1** is automatically inserted into the project. The form is the first thing the project displays when it runs.

### A Form is a Class

Behind every form is Visual Basic source code that describes the form's underlying class. A *class* is a description of the data and procedures that work together to give the form its behavior. When your Visual Basic program displays a form, it is displaying an instance of the form's class. Following is a minimal class definition for a form named Form1:

```vbnet
Public Class Form1
    Inherits System.Windows.Forms.Form

    ' Windows Form Designer generated code

End Class
```

There's quite a bit of additional source code hidden behind the "Windows Form Designer generated code" entry. If you click on the + sign, that code is revealed as well. Until you reach a much more advanced level of Visual Basic programming, you won't need to use the hidden code. Here are a few essential properties that you should know about forms:

- **Text**: Change a form's title by setting its *Text* property. The default is the form's name.

- **FormBorderStyle**: Change a form's border type, using its *FormBorderStyle* property. The default is sizeable, meaning that the user can resize the form when the program is running. If you set the FormBorderStyle to *FixedSingle*, the user cannot resize the form at runtime.

- **StartPosition**: Determines where the form will be located when it is displayed. If you set it to CenterScreen, the form will appear in the middle of the screen.

### PictureBox Control

A PictureBox control holds a graphics image that typically originates in a file having any of the following filename extensions: bmp, png, gif, jpg, jpeg, emf, wmf, or ico. An example of a PictureBox is shown in Figure B-3, shown while a program is running. Inserting a photo on a form requires almost no effort at all.
PictureBox controls are able to resize themselves to fit their image files. They can also stretch or shrink images to fit the PictureBox's boundaries. Two PictureBox properties are immediately important:

**Image**: Use the Image property to insert an image into the PictureBox. You do that by selecting an image filename.

**SizeMode**: Control whether the image remains the same size when the PictureBox is resized, or whether the picture is stretched. Table B-1 describes the various choices.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AutoSize</td>
<td>Size of the PictureBox's control's bounding box is automatically adjusted to fit the size of the image.</td>
</tr>
<tr>
<td>CenterImage</td>
<td>If the control's bounding box is larger than the image, the image is displayed in the center of the control. If the image is larger than the control's bounding box, the image is displayed in the center of the control and is clipped to fit within the bounding box.</td>
</tr>
<tr>
<td>Normal</td>
<td>The image is aligned with the upper-left corner of the control's bounding box. If the image is larger than the bounding box, it is clipped.</td>
</tr>
<tr>
<td>StretchImage</td>
<td>The image is scaled to fit within the control's bounding box. If the image is smaller, it is stretched. If the image is larger, it is reduced.</td>
</tr>
</tbody>
</table>

**ToolBox Window**

The ToolBox window is a container that holds controls that may be used in Visual Basic programs. It consists of several containers, named Data, Components, Windows Forms, Clipboard Ring, and General. For the moment we're only interested in the **Windows Forms** container.

To view the Toolbox window, hover the mouse over the Toolbox label at the left side of the Visual Studio .NET window. Notice that the Toolbox window pops out from the side, as in Figure B-4. Each tool represents a visual control that can be placed on a Form. Over the next several chapters, we will show you how to use the most commonly-used tools.
General Tips for Working with Forms and Controls

- To access the Properties window for any object, select the object with the mouse. The Properties window for that object will appear on the right side of the Visual Studio window. If the Properties window does not appear, select Properties Window from the View menu (or press the F4 key).

- To change the form's properties, click the mouse in any open area on the form.

Setting Option Strict to On

Before beginning to create programs in Visual Studio, we recommend that you set the Option Strict option to On. Doing this will prevent Visual Basic from performing implicit conversions between data types. Instead, conversions will be performed explicitly when you call standard conversion functions. In this way, you can avoid unintended data conversions and hard-to-find programming errors. Do the following steps:

1. Select Options from the Tools menu.

2. Double-click on Projects in the Options list box.

3. Select VB Defaults.

4. In the Default Project Settings, select On for the Option Strict setting.

Program Examples

Certain program examples are supplied along with topics and tutorials in this chapter. When you see a CDROM icon and a name, look for the associated program in the samples. For example, this icon and name refers to the Pet Adoption example program:

PetAdoption

Hands-On Tutorial: Pet Adoption Program

PetAdoption

Your first Visual Basic .NET program will be directed toward a good social cause: The first version of
our Pet Adoption program. The program will display a picture of a cute animal that just about anyone would want to adopt. Figure B-5 shows what the program will look like when it runs.

Figure B-5 Pet Adoption program.

Step 1. Select New from the File menu, and select Project. When the New Project window appears, you can select the project's Location and Name (see Figure B-6). The Location can be either an existing directory or the name of a directory that you want to create. Enter PetAdoption as the project name, and select your working directory for Visual Basic projects as the location.

Figure B-6 New Project window

Step 2. Click once on Form1 and look at the Properties window (lower right side of screen). Find the Name property of the program's form, shown in Figure B-7. Change the Name property from Form1 to frmPicture. Change the FormBorderStyle property to FixedSingle. Change the StartPosition property to CenterScreen. Change the Text property to Adopt Me!. Figure B-8 shows the form with the new title. Notice the small squares (called sizing handles) at the form's corners: you can resize the form by dragging the handles with the mouse.
Step 3. Notice the Solution Explorer window (Figure B-9) on the right side of the Visual Studio .NET window. It shows the Solution name (PetAdoption, the name of our program), as well as the Project name (also PetAdoption). The Solution Explorer also lists the files that belong to your project. For example, Form1.vb is the program's form file. Let's rename it by right-clicking on the form name, selecting Rename from the popup menu, and change the name to frmPicture.vb.

Tip: It's generally a good idea to make a form's filename similar or identical to its Name property.

Step 4. Right-click on the project name (PetAdoption) in the Solution Explorer window, select Properties, and display the Project Properties window, shown in Figure B-10. Notice that the Startup Object entry currently lists Form1.
Change the startup object to \texttt{frmPicture} (by selecting from the list).

\textbf{Step 5.} Add a PictureBox control to the form by dragging it onto the form from the ToolBox window. Resize the control so that it fills the entire form.

\textbf{Step 6.} Select the PictureBox with the mouse, and click on the ellipses button (\ldots) in its Image property. When the Open File window pops up, locate the file named kitten.jpg. You should see the same image shown earlier in Figure B-5.

If you have a picture of your own cat, dog, or other pet, feel free to use yours instead. Just resize it to approximately 256 by 319 pixels.

\textbf{Step 7.} Save the project by selecting Save All from the File menu. Or, you can click on the Save All button on the Visual Studio .NET toolbar. We hope you will do this often, to avoid losing valuable work!

\textbf{Step 8.} Run the program by selecting \texttt{Start Without Debugging} from the Debug menu. The program's window should appear right in the middle of the screen, containing the kitten. Stop the program by clicking on the X in the upper-right corner of the window.

\textit{Tip:} You can start a program by selecting \texttt{Start} from the Debug menu. If you do so, the program will run in debug mode. It starts a little more slowly, but the debugger allows you to step through your program code and display variables.

(End of Tutorial)

\section*{B.3 VARIABLES AND DATA TYPES}

\textit{VarTypes}

\textbf{Declared Inside Methods}

Variables declared inside methods use the following syntax:

\texttt{Dim varname As vartype}

The name of the variable is in \texttt{varname}, and the variable's type is in \texttt{vartype}. Following are common
examples of declaring primitive types:

    Dim isFullTime As Boolean
    Dim myByte As Byte
    Dim count As Integer
    Dim ratio As Single
    Dim beamWeight As Double
    Dim federalTax As Decimal
    Dim firstName As String

Table B-2 lists the Value types in Visual Basic .NET. There are additional types that belong to the .NET Framework, but we will not discuss them here. Ordinarily, you choose a variable's type based on the type of data that the variable will hold. For example, an employee's salary could be represented easily by a variable of type Single:

    Dim weeklySalary As Single = 1213.56

A calculated value used for engineering work often requires a variable of type Double:

    beamWeight = totalWeight / numberOfBeams

Financial values are often stored in variables of type Decimal, to avoid rounding errors:

    federalTax = grossPay * fedTaxRate

It is possible to declare more than one variable in the same statement:

    Dim A, B, C As Integer

---

### Table B-2 Visual Basic .NET value types.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>Holds the values True and False.</td>
</tr>
<tr>
<td>Byte</td>
<td>Holds an integer between 0 and 255.</td>
</tr>
<tr>
<td>Char</td>
<td>Holds a single Unicode character (2 bytes).</td>
</tr>
<tr>
<td>Date</td>
<td>Holds a date and time between Jan 1, 01 and December 31, 9999.</td>
</tr>
<tr>
<td>Decimal</td>
<td>Holds a 64-bit signed real number, with up to 29 significant digits. Suitable for financial calculations.</td>
</tr>
<tr>
<td>Double</td>
<td>Holds a precise real number, with approximately 17 significant digits. Suitable for engineering calculations.</td>
</tr>
<tr>
<td>Integer</td>
<td>Holds a 32-bit signed integer between −2,147,483,648 and +2,147,483,647.</td>
</tr>
<tr>
<td>Long</td>
<td>Holds a 64-bit signed integer in the range +/− 9 quintillion.</td>
</tr>
<tr>
<td>Object</td>
<td>Holds a reference (address of) an object.</td>
</tr>
<tr>
<td>Short</td>
<td>Holds a 16-bit signed integer in the range −32,768 to +32,767.</td>
</tr>
<tr>
<td>Single</td>
<td>Holds a 32-bit signed real number, with approximately 7 significant digits.</td>
</tr>
</tbody>
</table>

### Default Values and Initialization

If a variable is not assigned a value, it is assigned a default value. Numeric variables equal zero by default, Boolean variables equal False, and Date variables equal 12:00:00 am, Jan 1, of the year 1. String and Object variables are automatically assigned a value of Nothing. Nothing is a Visual Basic keyword that is equivalent to null in other programming languages. You can initialize a variable at the same time it is declared. For example:
Dim isFullTime As Boolean = True
Dim count As Integer = 15
Dim ratio As Single = 1.2
Dim lastName As String = ""

Assigning an empty string to a String variable is different from just leaving it equal to Nothing.

Because Decimal values can use more digits than other data types, it is often necessary to append the D character suffix to a constant when assigning a value:

Dim collegeTuition As Decimal = 922337203.6854775808D

Converting Between Types

There is a set of globally defined functions that perform conversions (called casts). For example, CInt casts an expression into an integer:

Dim temp As String = "22"
Dim N As Integer = CInt(temp)

If the input string contains a value that cannot be converted to an integer, CInt throws an exception, which stops the program. Table B-3 lists the available set of conversion functions:

<table>
<thead>
<tr>
<th>Name</th>
<th>Converts expression To:</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBool(expression)</td>
<td>Boolean</td>
</tr>
<tr>
<td>CByte(expression)</td>
<td>Byte</td>
</tr>
<tr>
<td>CChar(expression)</td>
<td>Char</td>
</tr>
<tr>
<td>CDate(expression)</td>
<td>Date</td>
</tr>
<tr>
<td>CDbl(expression)</td>
<td>Double</td>
</tr>
<tr>
<td>CDec(expression)</td>
<td>Decimal</td>
</tr>
<tr>
<td>CInt(expression)</td>
<td>Int</td>
</tr>
<tr>
<td>CLng(expression)</td>
<td>Long</td>
</tr>
<tr>
<td>CObj(expression)</td>
<td>Object</td>
</tr>
<tr>
<td>CShort(expression)</td>
<td>Short</td>
</tr>
<tr>
<td>CSng(expression)</td>
<td>Single</td>
</tr>
<tr>
<td>CStr(expression)</td>
<td>String</td>
</tr>
</tbody>
</table>

Value Types and Reference Types

Visual Basic classifies variables according to whether they are value types or reference types. A value type holds its data within its own memory allocation. A reference type contains a pointer to another memory location that holds the data. Sometimes pointers are also called references.

- Value types: all numeric types, Boolean, Char, Date, structures, and enumerations
- Reference types: strings, objects, arrays

(We will cover objects, structures, and enumerations at a later time.) Value types store their data in the same location as the variable. They support the assignment operator. In the following example, x contains a copy of y's data. When y is modified, x is not affected because its storage is separate from that of y:

Dim x As Integer = 10
Dim y As Integer = x
y = 22                          'x still equals 10

A reference type stores only a reference to an object inside the variable. The object's storage is separate from that of the variable. If we declared two Form variables and assigned one to the other, both variables would reference the same object:

Dim A As Form = New Form( )
Dim B As Form = A
B.Text = "This is Form B"        'A.Text = "This is Form B"

As a result, any change to a property in B affects the same property in A.

**Variable Scope**

Visual Basic .NET follows the same rules for variable scope as other popular programming languages such as Java and C++. Variables declared inside methods or procedures are only useable from the point of their declaration until the end of the procedure block. Variables declared at the class level, outside of any methods, exist as long as their surrounding object. Variables declared outside of any class exist for the duration of the program. Generally, global variables are used very sparingly, because they make the debugging of programs much more difficult.

**Variable Lifetime**

For value types (such as numbers), the lifetime of a variable is the same as its scope. For reference types (such as objects), the lifetime ends when no references to the object exist. The object becomes eligible for destruction by the garbage collector (a background process in Visual Basic), which eventually reclaims the memory used by the object. In the following example, the object referenced by obj is created by the New operator. It can be referenced until it is set to Nothing, the VB equivalent to null:

```
Dim obj As New Object          'object becomes reachable
   .
   .
obj = Nothing                 'object is no longer reachable
```

After obj is set to Nothing, it cannot be referenced without causing a runtime error. The object it references may still be kept in memory, depending on when the garbage collector decides to reclaim its storage.

**Dates and Times**

Date variables hold both a date and a time. Time literals must be enclosed by # symbols:

```
nextAppointment = #3/15/2004 3:35:00 PM#
```

The following statement assigns the current date and time, using the predefined property named Now:

```
nextAppointment = Now
```

You can initialize a Date variable with just a date:

```
nextAppointment = #4/20/2004#
```

You can also initialize a Date variable with just a time:

```
nextAppointment = #3:35 PM#
```

The ToDateTime function (in the System.Convert namespace) converts a string to a Date object. For example, the following converts the contents of a text box into a Date:
nextAppointment = System.Convert.ToDateTime(txtAppointment.Text)

**Current Date and Time**

You can get the date and time from the host system's internal clock using the following three Visual Basic properties:

- **Now** – The current date and time
- **TimeOfDay** – Only the current time
- **Today** – Only the current date

All three are demonstrated here:

```vbnet
Dim D As Date
D = Now
D = TimeOfDay
D = Today
```

**B.4 INTERACTIVE CONTROLS**

**Form Properties and Methods**

Before we write our next program, let's survey a few essential concepts, techniques, controls, and terms.

**Viewing the Code Inside a Form**

There are at least three ways to view a form's Code window.

1. Select the form while in Design mode. Press the F7 key.
2. Select the form file in the Solution Explorer window. Then click on the *View Code* icon on the Solution Explorer's toolbar.
3. Select the form while in Design mode. Select *Code* from the *View* menu.

Most programmers like to open both Design and Code windows for the same form. All currently open windows are listed in the Visual Studio .NET *Window* menu, so you can use the menu to switch between windows.

**Me Keyword**

The *Me* keyword can be used within a class (including forms) to stand for the current object. If you're writing code inside a form named *frmMain*, for example, *Me* represents the current instance of the *frmMain* class.

**ControlBox Property**

The *ControlBox* property of a form determines whether three icons will appear in the form's upper-right corner. By default, the property is True (as in our previous program). If you set *ControlBox* to False, all three icons disappear.
MaximizeBox and MinimizeBox Properties

The MaximizeBox property can be used to control whether a maximize icon appears in the upper right corner of a window. The property is True by default, but you can set it to False if you don't think it's appropriate for users to be maximizing your program's window. Removing the icon also tends to give the form's title bar a cleaner appearance.

The MinimizeBox property can be used to control whether a minimize icon appears in the upper right corner of a window. The property is True by default, but you can set it to False if you don't think it's appropriate for users to be minimizing your program so that it appears as an icon on the Windows status line. Removing the icon also tends to give the form's title bar a cleaner appearance.

Close Method

Every form has a Close method that when called, causes the form to close. If no other forms are open, the program ends.

Label Control

A Label control displays static text, that is, text which cannot be directly modified by the user. Labels are useful for displaying information such as titles and field labels, and for displaying information generated by the program. In Figure B-11, for example, both "First Name" and "LastName" are displayed by Label controls.

Figure B-11  Adoption Registry program.

 TextAlign Property

The TextAlign property applies primarily to Label and TextBox controls. When you select the property with the mouse, the following interactive tool appears:

By clicking on individual rectangles, you set the alignment of text within the control. The various settings are: TopLeft, TopCenter, TopRight, MiddleLeft, MiddleCenter, MiddleRight, MiddleCenter, MiddleRight, BottomLeft, BottomCenter, and BottomRight.

TextBox Control

A TextBox control provides an area on a form into which the user can type text. The text they type displays immediately, and is stored in the Text property of the TextBox. Programs can then retrieve the text and use it in some way, perhaps storing the information in a database. In Step 7, which we saw
earlier, the names "Joe" and "Smith" were typed into TextBox controls.

**CheckBox Control**

A CheckBox control lets the user select an item using the mouse or the keyboard spacebar. It is typically used for boolean true/false values, or values that can be represented by "on" and "off". When the user clicks on an empty check box, an "X" appears in the box, and the CheckBox control's Checked property is set to True. When the box is cleared, the Checked property becomes False. You can also set the Checked property in design mode. The CheckBox control also has a Text property, which you can use to display text next to the check box icon.

**Button Control**

The Button control, also known as a push button, responds to a mouse click by bouncing up and down. It is used whenever we want some program action to be carried out immediately. Most of the time, we write code in a special class method called a handler that responds when the button is clicked at runtime.

**Tutorial: Pet Adoption Registry**

Let's create a simple Pet Adoption Registry program that inputs information about a single person who would like to adopt a pets. A check box will indicate whether the person is a new customer. A push button will close the form.

**Hands-On Steps**

1. Create a new Windows Application Project named AdoptionRegistry.
2. Delete the startup form and create a new form named frmMain.
3. In the Solution Explorer window, right-click on the Project name, select Properties, and change the Startup object to frmMain.

From this point forward, when we say "set the startup form to <name>", you should delete the existing startup form, create a new form with the given name, and set the Project's startup form to the form you just created.

4. Change frmMain's Text property to Pet Adoption Registry. Set its FormBorderStyle property to FixedSingle. Set its StartPosition property to CenterScreen. Set its MaximizeBox and MinimizeBox properties to False. The form should appear as follows:
Step 5. Drag two Label controls and two TextBox controls from the ToolBox onto the form. Change the Text property of the first label to **First Name**, and change the Text property of the second label to **Last Name**.

Step 6. Change the TextAlign property of both Label controls to **MiddleRight**. Position the four controls as shown here:

![Form with Label and TextBox controls](image)

From this point forward, we will refer to a TextBox control as a *text box*. Similarly, a Label control will be referred to as a *label*. The Name property of a control will be referred to as its *name*.

Step 7. Change the name the upper text box to **txtFirstName**. Change the name of the second text box to **txtLastName**.

Step 8. Add a CheckBox control to the form, just below the txtLastName text box. Set its name to **chkNewCustomer** and set its Text property to **New Customer**.

Step 9. Add a Button control to the lower center area of the form. Set its name to **btnOK**, and set its Text property to **OK**. Your form should now look like the following:

![Form with additional controls](image)

Step 10. Save and run the program. Type your name into the two text boxes, and click on the New Customer check box. Click on the OK button (it does nothing, except bounce up and down). End the program by clicking on the X in the upper-right corner of the window.

**Adding a Button Click Handler**

Before we finish with this program, it would be nice to use the OK button to display a popup message and close the window. To do that, we will have to write program code inside a specific type of procedure named an event handler.

Step 11. Assuming that you're back in design mode, double-click the OK button, causing a code window to appear that contains form's source code. Notice that a procedure named **btnOK_Click** was created automatically:
Notice that the first line in the edit area of this window contains Public Class frmMain. Every form in a Visual Basic program contains an underlying class by the same name as the form. It is in this class that you write code statements that respond to events generated by the user, such as mouse clicks and keyboard input.

The name btnOK_Click identifies a special kind of method named an event handler. It executes when the user clicks on the OK button at runtime. A method is a procedure that is located inside a class.

Step 12. Insert the following statement inside the btnOK_Click procedure:

```vbnet
Me.Close()
```

The keyword Me stands for the current object, which in this case is the form. Close the Code window (click on the X in the upper right corner of the Code window, not the Visual Studio .NET window).

Step 13. Run the program. Click on the OK button, and note that the form closes and the program stops.

### Setting the Tab Order

If you run the AdoptionRegistry program and press the Tab key, the input focus will move between the two text boxes, check box, and the push button. The default tab order might or might not be logical, because it's based on the order in which you created the controls. Users like to have a predictable tab sequence, so Visual Basic provides a great tool for setting the tab order. It's activated by selecting Tab Order from the View menu. When you do this, each control will display a number next to it (starting at 0) that shows the current tab order, as in Figure B-12.

**Figure B-12  Viewing the tab order.**

To change the tab order, click the mouse on the blue square next to each field, starting with index zero. In Figure B-13, for example, four of the controls have been selected, and the label controls have been ignored. Ordinarily, label controls do not respond to user input. When you're finished, select Tab Order
from the View menu to hide the tab numbers.

If you don't want an input control to have a tab stop, you can set the control's TabStop property to False.

**Figure B-13  Setting a new tab order.**

![Image of a tab order]

**Step 14.** Set the tab order for the Adoption Registry program, and run it again to verify that your changes were made. Save the program.

*(End of tutorial)*

**Events and Handlers**

One of the essential aspects to using an event-driven language such as Visual Basic is that events are always being passed to your program. Some events occur as a result of actions by the user, such as clicking the mouse on a control or typing text into a control. Other events occur as a result of Visual Basic itself. Every control on a form has a rather large list of event messages that it can send to your program. Chances are that you will use only a few. Every control also has a default event handler, which you can create simply by double-clicking on the control in design mode. Following are the default events for the controls we have seen so far:

<table>
<thead>
<tr>
<th>Event</th>
<th>Default For</th>
<th>Occurs When...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Click</td>
<td>Button, CheckBox, PictureBox</td>
<td>Mouse clicked on the control</td>
</tr>
<tr>
<td>TextChanged</td>
<td>TextBox</td>
<td>Contents of the Text property have changed</td>
</tr>
<tr>
<td>Load</td>
<td>Form</td>
<td>The form is loaded into memory, before the form is displayed</td>
</tr>
</tbody>
</table>

Your skill in writing event handlers ultimately determines the level of sophistication of your Visual Basic applications. There are too many events for even a skilled programmer to remember, so you will get used to looking up their information when needed.

For any control, you can view its list of events by doing the following: Open the form's code window; select the control name from the objects pulldown list on the left side of the window; pull down the event list from the right side of the window.

**More about the Click Event Handler**

Let's revisit the Click event handler in our Adoption Registry program and fill in some more details. It is a...
Sub (subroutine) procedure containing a number of elements:

```vbnet
Private Sub btnOK_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnOK.Click
    Me.Close()
End Sub
```

If you look at this in the Visual Studio .NET editor, you probably notice that the first line is considerably longer. Because of printing limitations, we have divided the original line into two lines, using the line continuation character (_ or underscore) to break the two lines. We will continue to break long lines throughout the book, to improve readability.

Let's dissect the first two lines, one element at a time:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private Sub</td>
<td>Marks the beginning of the procedure</td>
</tr>
<tr>
<td>btnOK_Click</td>
<td>Event handler's procedure name</td>
</tr>
<tr>
<td>ByVal sender as System.Object</td>
<td>The control that sent the message</td>
</tr>
<tr>
<td>ByVal e as System.EventArgs</td>
<td>Extra information about the control that can be used to make decisions inside the handler</td>
</tr>
<tr>
<td>Handles btnOK.Click</td>
<td>The event name that is being handled</td>
</tr>
</tbody>
</table>

## B.5 USING TEXT BOXES AND LABELS

TextBox and Label controls have quite a few capabilities, which we only touched on during the previous program example. It's time to take a closer look.

### Using the Text Property in Code

You can access the Text property of a text box or label in code by using the form: `controlName.propertyName`. For example, we could assign a string to a TextBox control named `txtLastName` thus:

```vbnet
txtLastName.Text = "Maria Gonzalez"
```

The following statement copies the contents of a text box to a Label control named `lblCustomer`:

```vbnet
lblCustomer.Text = txtLastName.Text
```

### Clearing Text Boxes and Labels

You can clear a text box by calling its `Clear` method:

```vbnet
txtLastName.Clear()
```

Or, you can clear it by setting its Text property to an empty string:

```vbnet
txtLastName.Text = ""
lblCustomer.Text = ""
```

Label controls do not have a Clear method, so the empty string approach must be used.
**String Concatenation**

The & operator can be used to concatenate (combine) strings to other strings, variables, or constants. The following statement, for example, concatenates the contents of two text boxes with a single space to produce a person's complete name:

```
lblFullName.Text = txtFirstName.Text & " " & txtLastName.Text
```

If txtFirstName.Text contained "George" and txtLastName.Text contained "Washington", then the value stored in lblFullName.Text would be "George Washington".

By default, strings can also be combined with numeric constants, as in the following:

```
lblResult.Text = "Your monthly salary is " & 3500.00
```

More often, however, programs concatenate strings and variables. In the following example, a variable named `salary` is defined and set to 3,500. Then the variable and string are assigned to `lblResult.Text`:

```
Dim salary As Decimal = 3500.00
lblResult.Text = "Your monthly salary is " & salary
```

Visual Basic also allows the + operator to be used rather than & for string concatenation, but the latter operator is used more often.

**Hiding and Disabling Controls**

All Visual Basic controls can be hidden by setting their Visible property to False. To make an invisible control visible, set its Visible property to True. This property can be manipulated both in design mode and at runtime.

All Visual Basic controls can be disabled by setting their Enabled property to False. A disabled control is still visible, but it has a grayed-out appearance and it does not respond to user input. To re-enable a control, set its Enabled property to True. This property can be manipulated both in design mode and at runtime.

**Date String Example**

Let's write a simple program that shows uses text boxes, labels, and push buttons. We'll also demonstrate string concatenation, as well as the hiding, clearing, and disabling of controls at runtime. As you make changes to the program, we will ask you to run and test the program so you can better see the effects of your code.1

**Step 1.** Create a new Windows Desktop application named **DateString**.

**Step 2.** Insert four text boxes onto the startup form, with a Label control to the left of each text box. Use Figure B-14 as a guide.

---

1 From *Starting Out with Visual Basic .NET*, by Gaddis, Irvine, and Denton, copyright 2003. From this point on, the book will be identified by the acronym *SOVisual Basic*. 

Copyright Scott-Jones Publishing, 2004-2005. All rights reserved. 20
Step 3. Set the form's Text property to **Date String**.

Step 4. For each of the four labels, set their TextAlign property to **MiddleRight**. Set their Text properties as shown in the figure.

Step 5. Rename the text boxes as shown in the figure. Clear the contents of their Text properties. Set their Name properties to the names connected to the controls by arrows.

Step 6. Set the Enabled property of the txtDayName text box to False. (We will use a button to enable it at runtime.)

Step 7. Add a single label below the text boxes and name it lblFullDate. Set its BorderStyle property to **Fixed3D**, giving it a "sunken" appearance. Clear its Text property. Set its TextAlign property equal to MiddleRight.

Step 8. Add four Button controls to the bottom of the form. Set their Text properties as shown in the figure, and set their Name properties as shown by the arrows pointing to the buttons.

Step 9. Double-click on the btnEnable button and insert a single following code in its Click event handler. Following is the entire handler procedure:

```vbnet
Private Sub btnEnable_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnEnable.Click
    txtDayName.Enabled = True
End Sub
```

Our single line of code will enable the txtDayName text box when this button is clicked.

Step 9. Run the program and observe that the first text box is disabled until you click the Enabled button. Stop the program.

Step 10. In design mode again, double-click on the **Show Date** button. In its event handler, you need to write the following code that concatenates the four text box values into a single date. The data is assigned to the Label control named lblFullDate:

```vbnet
lblFullDate.Text = txtDayName.Text & " " &
```

Copyright Scott-Jones Publishing, 2004-2005. All rights reserved. 21
& txtMonth.Text & " " &
& txtDayNum.Text & ", " &
& txtYear.Text

Note that each line ends with a space, followed by a line-continuation character (_). In this way, you can break long statements into shorter, more readable lines.

Step 11. Run and test the program by clicking on the Enable button, entering a complete date, and clicking on the Show Date button. Figure B-15 shows an example of the output when Monday, March 15, 2003 is entered by the user.

Figure B-15 Testing the Date String program.

Step 12. Back in design mode, your next step is to code a handler for the Clear button. Double-click on the button and insert the following statements in its event handler:

```
txtDayName.Clear()
txtMonth.Clear()
txtDayNum.Clear()
txtYear.Clear()
lblFullDate.Text = ""
```

Step 13. Finally, insert the following statement in the event handler for the Close button:

```
Me.Close()
```

Step 14. Save, run, and test the program. Verify that all buttons work correctly.

(End of tutorial)

Formatting Numeric Data

Dates and Times

Dates and times can be formatted by passing them to the FormatDateTime function, along with a specifier that indicates which type of format to use. The function returns a string. The syntax is:
FormatDateTime( dateTimeExpression, formatSpecifier ) As String

The formatSpecifier can be any of the following: GeneralDate, LongDate, LongTime, ShortDate, or ShortTime. For example, let's create the following Date object (May 21, 1999, at 2:30 pm, and 59 seconds):

Dim d As Date = New Date(1999, 5, 21, 14, 30, 59)

Suppose we call the FormatDateTime function in each of the following ways:

FormatDateTime( d, DateFormat.GeneralDate )
FormatDateTime( d, DateFormat.LongDate )
FormatDateTime( d, DateFormat.LongTime )
FormatDateTime( d, DateFormat.ShortDate )
FormatDateTime( d, DateFormat.ShortTime )

The following table displays the strings returned by calling FormatDateTime with the selected DateFormat values:

<table>
<thead>
<tr>
<th>DateFormat Type</th>
<th>Returned String Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeneralDate</td>
<td>5/21/1999 2:30:59 PM</td>
</tr>
<tr>
<td>LongDate</td>
<td>Friday, May 21, 1999</td>
</tr>
<tr>
<td>LongTime</td>
<td>2:30:59 PM</td>
</tr>
<tr>
<td>ShortDate</td>
<td>5/21/1999</td>
</tr>
<tr>
<td>ShortTime</td>
<td>14:30</td>
</tr>
</tbody>
</table>

You can also create custom formats for dates and times. For example, you might want to left-fill the day and month values with zeros, so that a column of dates will all have the same width:

D = #2/5/2005#
Dim S As String = Format(D, "MM/dd/yyyy") ' 02/05/2005

Notice that the Format function was used. The topic of custom formatting characters is too big to cover here, but you can read all about it in the Visual Basic help by looking for Format function, user-defined dates/times in the help index.

Real Numbers and Currency

The FormatCurrency function returns a string that formats Decimal values. For example, the following statement displays $36.23:

FormatCurrency(36.225) "$36.23"

The FormatNumber function returns a string that formats any numeric expressions. The output defaults to two decimal positions, with the last digit rounded upward:

FormatNumber(36.225) "36.23"

The FormatPercent function multiplies its argument by 100 and appends a % sign. It returns a string:

FormatPercent(0.0012) "0.12%"

If you want to be specific about the maximum number of digits to the right of the decimal point, add a second argument when calling FormatCurrency, FormatNumber, or FormatPercent. Following are examples of each:

FormatCurrency(36.225,1) "$36.2"
FormatNumber(36.225,3) "36.225"
FormatPercent(0.0012,4) "0.1200%"
If the foregoing functions do not provide the exact format you're looking for, you can call the **Format** function and pass a string format expression. Its syntax is:

```
Format( numExpression, formatExpression ) As String
```

Where *numExpression* is an integer or real expression, *formatExpression* includes a combination of the characters shown in Table B-4.

### Table B-4  FormatExpression Characters

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>Percent placeholder. Appears in the expression where indicated, and multiplies the expression by 100.</td>
</tr>
<tr>
<td>,</td>
<td>Thousands separator.</td>
</tr>
<tr>
<td>0</td>
<td>Zero (0) character</td>
</tr>
<tr>
<td>#</td>
<td>Digit placeholder</td>
</tr>
<tr>
<td>E, E+, e, e+</td>
<td>Scientific format.</td>
</tr>
<tr>
<td>−, +, $, (</td>
<td>Literal characters, inserted in the string exactly as shown.</td>
</tr>
</tbody>
</table>

To learn more about the Format function, search for *Format function, user-defined numbers* in the MSDN help index. Table B-5 shows a number of example formats and their resulting string values, which you may want to keep for future reference.

### Table B-5  Format Function Examples

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Returned String Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format(36.225, &quot;###,###.##&quot;)</td>
<td>36.23</td>
</tr>
<tr>
<td>Format(36.2, &quot;000,000.00&quot;)</td>
<td>000,036.20</td>
</tr>
<tr>
<td>Format(.4, &quot;0.##&quot;)</td>
<td>.4</td>
</tr>
<tr>
<td>Format(.4, &quot;#.##&quot;)</td>
<td>.4</td>
</tr>
<tr>
<td>Format(.4, &quot;#.00&quot;)</td>
<td>.40</td>
</tr>
<tr>
<td>Format(123.456, &quot;##,###.##%&quot;)</td>
<td>12,345.6%</td>
</tr>
<tr>
<td>Format(123.456, &quot;0.00E+00&quot;)</td>
<td>1.23E+02</td>
</tr>
</tbody>
</table>

**ToString Method**

The **ToString** method uses the same format specifiers as the Format function. Because all numeric types understand **ToString**, you can format them easily. Examples are shown in Table B-6.

### Table B-6  Formatting with ToString (X As Double)

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>X = 36.225</td>
<td></td>
</tr>
<tr>
<td>X.ToString(&quot;###,###.##&quot;)</td>
<td>36.23</td>
</tr>
<tr>
<td>X = 36.2</td>
<td></td>
</tr>
<tr>
<td>X.ToString(&quot;000,000.00&quot;)</td>
<td>000,036.20</td>
</tr>
<tr>
<td>X = 0.4</td>
<td></td>
</tr>
<tr>
<td>X.ToString(&quot;0.##&quot;)</td>
<td>0.4</td>
</tr>
</tbody>
</table>
Run the **FormatExample** program from the chapter examples folder. It permits you to enter a number and experiment with different custom formats. When you run the program, as in Figure B-16, enter a real number and click on various buttons to select any of the predefined formats. As each format type is selected, the formatted value appears just below the text box. Alternatively, you can enter a custom format into the text box at the bottom of the form and click on the Use Custom Format button. The formatted value will appear in the label (with the black frame).

**Figure B-16 Formatting Examples program.**

### B.6 SALES COMMISSION CALCULATOR EXAMPLE

We will design and implement a program that calculates monthly sales commission earned by company employees. It will be developed in steps, to make it easier to move from relative simplicity to a more robust, complex end result. Each new version adds new features:

- **Version 1**: Input monthly sales, calculate commission amount, using a fixed commission rate.
- **Version 2**: Input a salary advance amount, which an employee may request during the month. At the end of the month, the salary advance is deducted from the earned commission amount.
- **Version 3**: Select different commission rates, depending on whether the employee is a manager, a full-time employee, or a part-time employee.

**Version 1**

**SalesCommission1**

In Version 1 of the Sales Commission calculator, the user is prompted for a sales amount. When they click on the Calculate button, the Amount Due shows the commission amount the person will be paid. The commission rate (10%) is fixed. Figure B-17 shows the output when the sales amount is $50,000. If, on the other hand, the user enters a non-numeric (or blank) value for the sales amount, the program...
displays a message in the status bar at the bottom of the form (Figure B-18).

**Figure B-17**  Running the Sales Commission program, Version 1.

<table>
<thead>
<tr>
<th>Sales Amount</th>
<th>50000</th>
</tr>
</thead>
<tbody>
<tr>
<td>× 10% Commission rate</td>
<td></td>
</tr>
<tr>
<td>Amount Due</td>
<td>$5,000.00</td>
</tr>
<tr>
<td>Calculate</td>
<td>Exit</td>
</tr>
</tbody>
</table>

**Figure B-18**  Entering a non-numeric Sales Amount.

Sales amount must be numeric

**Hands-On Tutorial**

In this tutorial, as in all the hands-on tutorials in this book, you will probably get the most out of it by performing the described steps at the computer. Your fingers will help your brain remember what you did!

**Step 1.** Create a new Windows Application project named SalesCommission.

**Step 2.** In the project's Property Pages, set Option Strict to On. *(Right-click the project name in the Solution Explorer window, select Properties, select Common Properties / Build.)*

**Step 3.** Add a TextBox control, four Label controls, two Button controls, and a StatusBar control to the form, using Figure B-17 as a guide to their locations.

**Step 4.** Set the control and form properties according to the following table:

<table>
<thead>
<tr>
<th>Control Type</th>
<th>Name</th>
<th>Property Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form</td>
<td>Form1</td>
<td>Text: Sales Commission, Version 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>StartPosition: CenterScreen</td>
</tr>
</tbody>
</table>
Step 5. It's time to add some code to Form1. First, the commission rate is declared at the top of the class. Then the Click handler for the Exit button calls Me.Close to close the main form:

```vbnet
Private commissionRate As Decimal = 0.1D

Private Sub btnExit_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnExit.Click
    Me.Close()
End Sub
```

Step 6. Next, create the Click handler for the Calculate button. Before doing any calculations, it must find out if the user has entered a number into the txtSales control:

```vbnet
Private Sub btnCalculate_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnCalculate.Click
    If Not IsNumeric(txtSales.Text) Then
        sbMessage.Text = "Sales amount must be numeric"
        Exit Sub
    End If

    Dim amtDue As Decimal
    amtDue = CDec(txtSales.Text) * commissionRate
    lblAmountDue.Text = FormatCurrency(amtDue)
    sbMessage.Text = ""
End Sub
```

If a non-numeric value is found, we display a helpful message in the StatusBar control and exit. Otherwise, the amount due is calculated by multiplying the value in txtSales by the commission rate.

And, of course, the StatusBar control is cleared so that no error message will display.

Step 7. Save and run the program. Test it by entering various sales values and clicking on the Calculate button. Try entering random character values, to test the program's handling of nonnumeric data. Of course, we do not want the program to halt unexpectedly!

What if the user enters a negative value for the sales amount? The program just calculates a negative
value for the amount due. Perhaps you can find a way to prevent the program from doing that!

(End of tutorial)

**Version 2: Adding a Salary Advance**

Now we're ready to create the second version of the Sales Commission program. It permits the user to enter a salary advance amount. This amount will be deducted from the amount due, the amount paid the employee at the end of the month. For example, in Figure B-19, the commission would have been $7,000, but after deducting the $1,000 salary advance, the amount due is $6,000.

*Figure B-19   Version 2 of the Sales Commission program.*

Here are the required steps:

**Step 1.** Make a copy of the SalesCommission project folder and rename it to **SalesCommission2**. Open the project from this new folder.

**Step 2.** Change the form's Text property to "Sales Commission, Version 2".

**Step 3.** Add new controls to the form so that it looks like Figure B-19. Name the new text box **txtAdvance**.

**Step 4.** Modify the code inside the btnCalculate_Click procedure:

```vbnet
Private Sub btnCalculate_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnCalculate.Click
    If Not IsNumeric(txtSales.Text) Then
        sbMessage.Text = "Sales amount must be numeric"
        Exit Sub
    End If

    'Get the salary advance amount
    Dim advance As Decimal = 0
    If IsNumeric(txtAdvance.Text) Then
        advance = CDec(txtAdvance.Text)
```

Copyright Scott-Jones Publishing, 2004-2005. All rights reserved. 28
End If

Dim amtDue As Decimal
    amtDue = CDec(txtSales.Text) * commissionRate - advance
    lblAmountDue.Text = FormatCurrency(amtDue)
    sbMessage.Text = ""
End Sub

The salary advance might be either blank or it might contain nonnumeric characters. In either case, we simply set it to a default value of zero. In the calculation of amtDue, the salary advance is subtracted from the commission amount.

Step 5. Save and run the program. Input various values for both sales amount and salary advance. It's getting better all the time!

(End of tutorial)

Version 3: Varying the Commission Rate

In the third version of this program, we would like to vary the commission rate, depending on the employee classification:

- Full-time employees are paid at a 10% rate
- Managers are paid at a 12% rate
- Part-Time employees are paid at an 8% rate

The program uses a group of RadioButton controls to let the user select the employment status of the person being paid. In Figure B-20, for example, a Manager is paid a 12% commission on sales of $70,000 minus a $2,000 advance, so the amount due is $6,400.

Figure B-20 Sales Commission program, Version 3.

Step 1. Make a copy of the SalesCommission2 project folder and rename it to SalesCommission3. Open the project from this new folder.

Step 2. Change the form's Text property to "Sales Commission, Version 3".
Step 3. Add a GroupBox control to the right side of the form and set its Text property to "Employment".

Step 4. Insert three RadioButton controls into the GroupBox and name them radFullTime, radManager, and radPartTime. Set their Text properties as shown in Figure B-20.

Step 5. Following is the revised code for this program:

```vbnet
Const FullTimeRate As Decimal = 0.1D
Const ManagerRate As Decimal = 0.12D
Const PartTimeRate As Decimal = 0.08D
Private commissionRate As Decimal

Private Sub btnExit_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnExit.Click
    Me.Close()
End Sub

Private Sub Calculate_Pay()
    If Not IsNumeric(txtSales.Text) Then
        sbMessage.Text = "Sales amount must be numeric"
        Exit Sub
    End If

    'Get the salary advance amount
    Dim advance As Decimal = 0
    If IsNumeric(txtAdvance.Text) Then
        advance = CDec(txtAdvance.Text)
    End If

    Dim amtDue As Decimal
    amtDue = CDec(txtSales.Text) * commissionRate - advance
    lblAmountDue.Text = FormatCurrency(amtDue)
    sbMessage.Text = ""
End Sub

Private Sub btnCalculate_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnCalculate.Click
    Calculate_Pay()
End Sub

Private Sub radFullTime_CheckedChanged(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles radFullTime.CheckedChanged, radManager.CheckedChanged, radPartTime.CheckedChanged
    'Handles all three Employment radio buttons
    If radFullTime.Checked Then
        commissionRate = FullTimeRate
    ElseIf radManager.Checked Then
        commissionRate = ManagerRate
    ElseIf radPartTime.Checked Then
        commissionRate = PartTimeRate
    End If
```
commissionRate = PartTimeRate
End If
Calculate_Pay()
End Sub

Step 6. Save and run the program.

(End of tutorial)

B.7 WINDOWS CONSOLE PROGRAMS

ConsoleSample

Windows console programs provide a simple way to use text-based input/output. They can be useful for testing new ideas, because there is hardly any effort directed toward the user interface. Java and C++ programmers find the text interface quite familiar.

Console-based applications must the System.Console namespace. This can be done in one of two ways: First, you can insert the following line at the beginning of any source code file containing input-output statements:

Imports System.Console

Or, you can right-click on the project name in the Solution Explorer window, select Properties, select Imports under Common Properties, and type System.Console into the Namespace input box. Click on Add Import and click on OK to save your changes:

Writing to the Console Window

To write to the console, call either Write or WriteLine. It may contain one or more strings, value types, and object types. There are versions of Write and WriteLine that accept all standard Visual Basic types as parameters. You can combine strings and variables using the & operator. Here are some examples:

Write("Please enter your name: ")
WriteLine("Your monthly salary is " & mSalary)
WriteLine("There are " & count & " employees ")

**Reading from the Console Window**

To read from the console, call the `ReadLine` function, which returns a string:

```vbnet
Dim custName As String
custName = ReadLine()
```

To read a numeric value, you must convert the input string to the appropriate type. For example, let's declare two variables, N and temp:

```vbnet
Dim N As Integer
Dim temp As String
```

We prompt the user to enter an integer, which is temporarily stored in the string variable:

```vbnet
Write("Enter an integer: ")
temp = ReadLine()
```

As we saw earlier in the chapter, the CInt function converts the string into an Integer:

```vbnet
N = CInt(temp)
```

If the conversion is impossible, CInt throws an `exception`, which stops the program. Here's an example:

```vbnet
Enter an integer: xx
Unhandled Exception: System.FormatException: Input string was not in a correct format.
```

In Chapter 2 we will show how to let programs recover from exceptions. All of the standard numeric classes contain a `Parse` method that converts a string into the appropriate type.

The `Console.Read` method reads a single integer value from standard input (the keyboard):

```vbnet
Dim N As Integer = Read()
```

The user must press Enter to tell the Read method to stop waiting for input. You can convert the returned integer to a character by calling the `Chr` function:

```vbnet
Dim ch As Char = Chr(N)
```

### B.8 LOOPS

#### LoopExamples

These is a rich set of looping constructs in Visual Basic that should satisfy every need. Most are modeled after traditional loops in the BASIC language (such as the For-Next loop). Some loop statements test the loop condition before starting the loop (called pretesting): Do While, Do Until, and For Next. Other statements execute the loop body at least once before testing the loop condition: Do Loop While, and Do Loop Until.

#### Do While Loop

The Do While Loop statement is essentially the same as the while statement in Java. The `<expression>` must be True before the body of the loop can be executed:

```vbnet
Do While <expression>
   <statement-list>
Loop
```
The following loop uses the return value from calling a Boolean function named `CheckStatus` to determine whether the loop will continue:

```
Do While CheckStatus() = True
    'loop body...
Loop
```

**Do Until Loop**

The Do Until Loop statement also requires `<expression>` to be false before the body of the loop is executed:

```
Do Until <expression>
    <statement-list>
Loop
```

The following loop continues until the `CheckStatus` function returns False:

```
Do Until CheckStatus() = False
    'loop body
Loop
```

**Do Loop While**

A post-test loop, which will always execute once. It will only repeat if `<expression>` is true:

```
Do
    <statement-list>
Loop While <expression>
```

The following loop executes once and repeats if `CheckStatus` returns True:

```
Do
    'loop body...
Loop While CheckStatus() = True
```

**Do Loop Until**

A post-test loop, which will always execute once. It will only repeat if `<expression>` is false:

```
Do
    <statement-list>
Loop Until <expression>
```

The following loop executes once and continues if `CheckStatus` returns False:

```
Do
    'loop body
Loop Until CheckStatus() = False
```

**For Next**

The For Next statement provides a flexible counting loop. The `<counter>` variable must be declared before the loop statement. Elements within square brackets are optional:

```
For <counter> = <startval> To <endval> [Step <increment>]
    <statement-list>
Next [<counter>]
```

`<counter>` can be any type of numeric variable, including real numbers. `<increment>` is 1 by default, but
it can be an integer or real, positive or negative. Its type must be consistent with that of `<counter>`. The following loop counts from 0 to 9:

```vbnet
Dim i As Integer
For i = 0 To 9
    WriteLine(i)
Next
```

The following loop counts from 9 down to 0:

```vbnet
Dim i As Integer
For i = 9 To 0 Step -1
    WriteLine(i)
Next
```

The following loop counts from –5.0 to +5.0 in increments of 0.5:

```vbnet
Dim n As Single
For n = -5.0 To +5.0 Step 0.5
    WriteLine(n)
Next
```

**For Each**

The For Each statement has neither a loop condition nor an explicit counter. It iterates over an entire array or collection of objects:

```vbnet
For Each <elementVar> In <arrayVar>
    <statement-list>
Next [<elementVar>]
```

The `<elementVar>` variable must be declared as the same type as the array elements. Here's an example that displays each element in an array of employee names:

```vbnet
Dim employees() As String
' (assume employees is filled by other code)
Dim name As String
For Each name In employees
    WriteLine(name)
Next name
```

**Exit For**

The Exit For statement breaks out of a For Next loop without finishing the remaining statements in the loop. Here's an example:

```vbnet
Dim i As Integer
For i = 1 To 10
    If OpenFile() = False Then Exit For
    ReadFile()
    WriteToConsole()
Next
```

The Exit For statement should be used only when an If statement would be awkward. If we rewrite the loop just shown to eliminate the Exit For statement, we are forced into the risky proposition of altering the loop counter (i) inside the loop:

```vbnet
Dim i As Integer
```
For i = 1 To 10
    If OpenFile() Then
        ReadFile()
        WriteToConsole()
    Else
        i = 11  'force loop to stop??
    EndIf
Next

We might prefer to rewrite the loop using the Do While Loop statement instead:

    Dim i As Integer = 1
    Do While i <= 10
        If OpenFile(i) Then
            ReadFile(i)
            WriteToConsole(i)
        Else
            i = 11
        End If
    Loop

*Exit Do*

Used to break out of a Do While or Do Until loop. Here's an example:

    Dim i As Integer, x As Integer
    i = 1
    x = 10
    Do While i < 1000
        x += 10
        If x > 50 Then Exit For
        i += 1
    Loop

### B.9 CONDITIONAL STATEMENTS

#### If-Then

The If-Then statement evaluates a boolean expression and executes a single statement if the expression returns True:

    If <expression> Then <statement>

For example, the WriteLine statement executes when the value of A is greater than the value of B:

    If A > B Then WriteLine("A is greater")

The setResult method is called if the call to checkStatus returns True:

    If checkStatus() = True Then setResult()

The WriteLine statement executes if the compound condition is True:

    If A < B And A < C Then WriteLine("A is smaller than B and C")
### If- End If

The If-End If statement, known also as a Block-If, evaluates a boolean expression and executes one or more statements if the expression returns True:

```
If <expression> Then
  <statement-1>
  <statement-2>
  ...
  <statement-n>
End If
```

Following are examples:

```
If A > B Then
  WriteLine("A is greater than B")
  PerformCalculations()
End If

If A < B And A < C Then
  WriteLine("A is smaller than B and C")
  PerformValidation(A)
End If
```

### If-Else-End If

The If-Else-End If statement, also known also as a Block-If, evaluates a boolean expression and executes one or more statements if the expression returns True. It executes an alternate block of statements if the expression is False:

```
If <expression> Then
  <statement-1>
  <statement-2>
  ...
  <statement-n>
Else
  <statement-a>
  <statement-b>
  ...
  <statement-z>
End If
```

Following are examples:

```
If A > B Then
  WriteLine("A is greater than B")
  PerformCalculations()
Else
  WriteLine("A is NOT greater than B")
End If

If A < B And A < C Then
  WriteLine("A is smaller than B and C")
  PerformValidation(A)
Else
```
Nested-If

As in other programming languages, you can nest an If statement inside another If statement. The nesting can be as deep as you wish. For example:

```vbc
Dim A As Integer
Dim B As Integer
Dim C As Integer
If A < B Then
    Write("A is smaller than B")
    If A < C Then
        WriteLine(" and smaller than C")
    Else
        WriteLine(", but not smaller than C")
    End If
Else
    Write("A is NOT smaller than B")
    If A < C Then
        WriteLine(", but A is smaller than C")
    Else
        WriteLine(" and NOT smaller than C")
    End If
End If
```

Note from this example that an If statement nested inside another If statement must have a terminating End If.

Select Case

The Select Case statement provides a convenient multiway selection structure. You can use it to select between a set of numeric or string values. When a selection is matched, the statements immediately following the matching Case clause are executed.

**Integer Example**

Suppose the variable studentYear contains an integer that identifies the student's academic standing ('1=Freshman, 2=Sophomore, etc.). We can use a Select Case statement to display a message and assign a Boolean variable for each known value of studentYear:

```vbc
Dim studentYear As Integer
Dim mayOwnCar As Boolean

Select Case studentYear
    Case 1
        WriteLine("Student is a Freshman")
        mayOwnCar = False
    Case 2
        WriteLine("Student is a Sophomore")
        mayOwnCar = True
End Select
```
Case 3
    WriteLine("Student is a Junior")
    mayOwnCar = True
Case 4
    WriteLine("Student is a Senior")
    mayOwnCar = True
Case Else
    WriteLine("Unknown Student classification")
End Select

The Case Else clause matches any value not yet matched by the other Case clauses.

**String Values**

The Select Case statement is not limited to integers. You can use it to select strings:

```plaintext
Dim gradeLetter As String
Dim gradeVal As Single

Select Case gradeLetter
    Case "A"
        gradeVal = 4
    Case "B"
        gradeVal = 3
    Case "C"
        gradeVal = 2
    Case "D"
        gradeVal = 1
    Case "F"
        gradeVal = 0
End Select
```

Boolean expressions and ranges can be useful when looking for ranges of values. Suppose a state income tax rate is based on earning brackets:

```plaintext
Dim amountEarned As Decimal
Dim taxRate As Decimal

Select Case amountEarned
    Case Is < 10000D
        taxRate = 0D
    Case 10000D To 15999D
        taxRate = 0.03D
    Case 16000D To 25999D
        taxRate = 0.05D
    Case 26000D To 39999D
        taxRate = 0.06D
    Case Is >= 40000D
        taxRate = 0.07D
End Select
```

Boolean expressions must be preceded by the Is keyword. Range expressions must use the keyword To between the minimum and maximum values.
B.10 ARRAYS

ArrayExamples

Arrays in Visual Basic are zero-based, meaning that the first index position is zero. Arrays are reference types, making their behavior similar to objects.

Declaring an Array

There are two possible formats for declaring an array variable:

\[
\text{Dim } \langle \text{arrayName} \rangle \text{ As } \langle \text{type} \rangle() \\
\text{Dim } \langle \text{arrayName} \rangle() \text{ As } \langle \text{type} \rangle
\]

Arrays are objects, so you must use the New operator when creating an array instance. The value in parentheses indicates the array's upper bound (largest possible index). Here's an example:

\[
\text{Dim scores}() \text{ As Integer} \\
scores = \text{New Integer}(50) \{ \}
\]

The scores array, in other words, may contain 51 elements, indexed 0 to 50. You can combine the declaration and initialization into a single statements:

\[
\text{Dim scores}() \text{ As Integer} = \text{New Integer}(50) \{ \}
\]

Array Initializer

An array initializer is a list of elements that are loaded into the array when it is constructed. Any uninitialized elements are assigned default values. Here is an example of an initializer list that fills an array and determines its size:

\[
scores = \text{New Integer}() \{10, 20, 30, 40, 50\}
\]

The Length property returns a count of the number of elements in the array. The GetUpperBound method returns the largest possible subscript.

\[
scores.\text{Length} \quad '5 \\
scores.\text{GetUpperBound}(0) \quad '4
\]

Here is an example of using GetUpperBound in a For-Next loop that displays all array elements:

\[
\text{Dim i As Integer} \\
\text{For } i = 0 \text{ To scores.\text{GetUpperBound}(0)} \\
\text{\hspace{1cm} WriteLine(scores(i))} \\
\text{Next i}
\]

Reversing and Sorting

The Array class contains a Reverse method that reverses the order of elements in an array. It must be invoked using the Array class name, passing the array object as an argument. This is how we would reverse the scores array:

\[
\text{Array.Reverse(scores)}
\]

It also contains a Sort method that arranges the elements in ascending order. Here is how we would sort the scores array:

\[
\text{Array.Sort(scores)}
\]
**Copying an Array**

The **Copy** method in the Array class copies the elements from one array to another. Pass it the source array, the target array, and the number of elements to copy. The target array must have already been created, using the New operator. In the following example, `target` is created with the same size as the `scores` array:

```plaintext
Dim target() As Integer = New Integer(scores.GetUpperBound(0)) {}
```

The following statement copies the elements of `scores` into `target`:

```plaintext
Array.Copy(scores, target, scores.Length)
```

**Array of Objects**

An array of objects is declared in the same way as an array of primitives. The only difference is that object array elements must be initialized with object instances. Suppose a `Customer` class exists. We can then declare an array of `Customer` (indexes 0 to 10):

```plaintext
Dim customers(10) As Customer
```

All array positions default to null (called `Nothing` in Visual Basic). The following statements insert a reference to a `Customer` object:

```plaintext
Dim cust As New Customer
customers(0) = cust
```

You can set any array element to null (called `Nothing` in Visual Basic):

```plaintext
customers(1) = Nothing
```

**Iterate Over an Array of Objects**

The For-Each statement makes it easy for you to iterate (loop) over an array of objects. Suppose our `Customer` class had a BirthDate property. The following loop would write each birthdate to the console:

```plaintext
Dim cust As Customer
For Each cust In customers
    WriteLine(cust.BirthDate)
Next
```

The For-Each statement takes care of the hidden work of incrementing the array index and checking for the end of the array. It would not, however, skip any null values in the array. The following loop checks each object reference before accessing its BirthDate property:

```plaintext
Dim cust As Customer
For Each cust In customers
    If Not cust Is Nothing Then
        WriteLine(cust.BirthDate)
    End If
Next
```

**Arrays as Method Parameters**

Because arrays are reference types, they are implicitly passed to methods by reference. That means a called method can modify the state of the array. This is true even when the array parameter is declared ByVal:

```plaintext
Sub PrintArray(ByVal array As Integer())
    array(0) = 96
```

Copyright Scott-Jones Publishing, 2004-2005. All rights reserved.
End Sub

Sub main()
    Dim scores() As Integer = New Integer(5) { }
    PrintArray(scores)  
    // scores(0) now equals 96
End Sub

After returning from the call to PrintArray, position 0 of the scores array has a new value (96). There is no way to prevent this type of behavior (called a side effect), so you should avoid it unless absolutely necessary.

If we declare an array parameter ByRef, the called method can reassign the parameter to some other array object:

Sub ReplaceArray(ByRef array As Integer())
    array = New Integer() {0,1,2,3}
End Sub

Needless to say, such behavior would be quite confusing unless it were well documented. In general, side effects are discouraged in object-oriented programming because they create difficult-to-find logic bugs.

Two-Dimensional Arrays

The format for declaring a two-dimensional array variable is:

    Dim <arrayName> As <type>(,)

Suppose we have 50 students, and each has 10 project scores:

    Dim scores As Integer(,)
    scores = New Integer(50,10) { }

We can combine both statements into one:

    Dim scores As Integer(,) = New Integer(50,10)

Following is an example of a two-dimensional initializer list that automatically determines both array dimensions:

    scores = New Integer(,) {{89, 85, 76, 81}, {77, 86, 92, 94}, {99, 91, 88, 83}}

The following examples reference single array elements using two subscripts:

    scores(1,2)   'value = 96
    scores(2,0)   'value = 99

To get the upper bounds of each array dimension, pass an argument that indicates the array dimension (0 or 1):

    scores.GetUpperBound(0)  '2 (highest row index)
    scores.GetUpperBound(1)  '3 (highest column index)

Two-Dimensional Array Parameters

To pass a two-dimensional array to a method, declare the parameter type using (,) after the parameter name. For example:

    ByVal scores(,) As Integer

The following PrintScores method prints the two-dimensional scores array, writing each row to a
Sub PrintScores(ByVal scores(,) As Integer)
    Dim iRow As Integer, iCol As Integer
    For iRow = 0 To scores.GetUpperBound(0)
        For iCol = 0 To scores.GetUpperBound(1)
            Write(scores(iRow, iCol) & " ")
        Next iCol
        WriteLine()
    Next iRow
End Sub

Here is the output produced by PrintScores:

89  85  76  81
77  86  92  94
99  91  88  83