2. Visual Basic Programming

2.1. Prerequisites and Objectives

You may skip this chapter if:

You are already familiar with Visual Basic, Delphi, Visual Age, or some other component-based visual design tool.

Before starting this chapter you should have:

1. A general knowledge of programming.
2. Access to a computer. You should read this chapter while sitting at your computer.

After completing this chapter you will have:

1. A general familiarity with visual component-based programming.
2. A knowledge of how to use existing components to design an application.
3. Some knowledge of Visual Basic.

2.2. Introduction

Before one can develop new components, one must have an appreciation for how components are used. Learning Visual Basic is the easiest way to accomplish this. (Other reasonable starting points are the Delphi programming system, and the different JavaBeans visual editors.) Readers who are already familiar with component-based programming may wish to skip this chapter.

Learning Visual Basic is not particularly difficult. You can teach yourself in a few hours. Nevertheless, the whole thing is much easier with a few guidelines. This chapter is
not intended to be an extensive course in Visual Basic programming, but merely intended
to familiarize you with the fundamentals so you can begin learning on your own.

In this chapter, we will demonstrate how components are used to build applications,
and familiarize you with the concepts of properties, events, and methods. The examples
will be given in Visual Basic 6.0, and in VB.NET. The two versions of Visual Basic are
significantly different, so much so that version 6.0 will probably remain in use for many
years, despite being superseded by VB.NET.

2.3. Properties

Our first program demonstrates the use of properties. We begin by starting Visual
Basic 6.0 and selecting New Project from the File menu. Then we select Standard EXE
and click OK. (Your copy of Visual Basic may display this dialog automatically upon
startup.) Figure 2.1 shows the Visual Basic version 6.0 development environment with
the New Project dialog box displayed.
Figure 2.1. Creating A Visual Basic Project.

The preceding steps will create a new project and display the full development window. To perform the same task in VB.NET, we start Visual Studio (VS.NET) and select **New** and then **Project** from the **File** menu. This will display the dialog box that appears in Figure 2.2. In this dialog box select **Visual Basic Projects** from the list at the left. (This may change the choices that appear in the window at the right.) From the window at the right, choose **Windows Application**. Type the name of the project the **Name** box, and click OK.
At this point, we have a project window with a prototype view of our program’s main window. We are going to create an application that does some simple text-processing. To accomplish this we will to draw two text-boxes into the prototype view of our window, which is entitled **Form1**. To do this in Visual Basic 6.0, we click on in the toolbox at the left of the Visual Basic window. This gives us a cross-hairs cursor that will allow us to draw a rectangle on the Form1 window. Drawing the rectangle will cause a text-box to appear in the drawn rectangle. We repeat this process to draw a second text box. We now have two text boxes as illustrated in Figure 2.3. These two text-boxes are examples of Visual Basic **controls**. Visual Basic allows us to use several different types of **components**, Visual Basic controls are one such type.
To perform the same task in VB.NET, we must first select the correct section of the tool-box by clicking on **Windows Forms**. This will display the section of the toolbox containing the standard controls, such as text boxes and buttons. Select the textbox control by clicking on **TextBox** from this section of the tool-box. As in Visual Basic 6.0, selecting this control will give you a cross-hairs cursor that will enable you to draw a textbox. Draw two text-boxes, giving the window shown in Figure 2.4.

**Figure 2.3. Adding Two Text Boxes.**
Surprisingly enough, we now have a complete, running, Visual Basic Program. We could run it as-is, but the results would not be impressive. (If you are following along with this on your own computer, click on in the upper tool-bar, in either version of Visual Basic, to see the program run.) Visual Basic programs are passive. They sit and wait for something to happen, or more correctly, they wait for an event to occur. As yet, we have no way to cause events. However, before we proceed to that, let’s clean up the existing program. Our two text boxes already have text in them. These labels may be helpful while we are designing our program, but if we don’t do something about them, they will also appear when our program runs. What we would like to see when the program runs is two blank text boxes. To achieve this, we must alter the properties of our

Figure 2.4. Adding Text Boxes in VB.NET.
two controls. This can be done either at design-time, or at run-time during the initialization process. We will do this at design-time.

First we click on the upper text-box to select it. We can tell that the upper text-box is selected, because it is now displayed with eight *grab-handles* that allow us to change its size. Figure 2.5 shows the control with the grab-handles displayed.

![Figure 2.5. Selecting a Component.](image)
Next, we go to the properties sub-window on the right-hand side of the Visual Basic window. The properties window is shown in Figure 2.6.

![Properties Window](image)

**Figure 2.6. The Property Sub-Window.**

The first thing we see in the properties window is the name of the control and its type. In the window below we see an alphabetic list of the component’s properties, and their current values. On the left is the name of each property supported by the component. *Text* is an example of such a property. Visual Basic adds a number of additional properties, such as *TabIndex*, to those supported by the component. If a component has run-time-only properties, they will not be displayed in this list.

To the right of each property name is the value of the property for this instance of the component. Any two components of the same type will have the same list of property names, but the values of these properties will generally be different for each instance of the component. To get rid of the “Text1” or “TextBox1” label, we must change the value
of the *Text* property. (If the *Text* property is not displayed, we must use the scroll-bar at
the right of the property list to find it.) We want to delete this label entirely, so we select
the label in the property list, and press **Delete**. In VB.NET you may also need to press
**Enter** to confirm the change. As we erase the text from the *Text* property, it also
disappears from the text-box in the **Form1** window. Even though we are in design-mode,
the component-instance is loaded, and running. It is monitoring the state of its properties,
and updating its display accordingly. We use the same procedure to erase the other label.

It is important to remember that the *name* of the upper text box is *Text1* or *TextBox1* and
the *name* of the lower text box is *Text2* or *TextBox2*. We will use these names in the
remainder of our program.

In our simple text-processing example, the user will type some text into the
*Text1/TextBox1* textbox. We will examine this text and delete each instance of the letter
“e”. We will place the result into the *Text2/TextBox2* textbox. The following Visual Basic
code will accomplish this, in either version of Visual Basic.

```vbnet
' Declare Two Work Variables and a Loop Counter
Dim Work1 As String
Dim Work2 As String
Dim I As Integer

' Retrieve the contents of Text1
Work1 = Text1.Text
' Initialize Work2
Work2 = ""
' Examine each character of Text1
For I = 1 To Len(Work1)
    If Mid$(Work1, I, 1) <> "e" Then
        ' Add to Work2 if not equal to "e"
        Work2 = Work2 & Mid$(Work1, I, 1)
    End If
Next I

' Save result in Text2
Text2.Text = Work2
```
This code illustrates several of the most important features of the Visual Basic language. A comment begins with an apostrophe and continues to the end of the line. Variable Declarations begin with the keyword Dim (for Dimension). The For loop, with a loop counter has the form “For <loop counter> = <starting value> To <ending value>.”

The body of the loop is terminated by a Next statement, which optionally names the counter variable specified in the For statement. An If statement has the form “If <condition> Then”. The body of the If statement is terminated by an End If statement. The Else body of an if statement, if any, is introduced with an Else statement. If an Else statement is present, only one End If statement is used, and it terminates the Else body.

Assignments are specified with an equal sign. The Len function is used to determine the length of a string, while the Mid$ function is used to extract substrings. The Mid$ function has three parameters, the first is the source string, the second is the starting position of the substring, and the third is the length of the substring. The first character in a string is position 1. Assignment statements use the typical +, -, *, and / symbols to perform arithmetic. Comparison symbols are =, for equals, <> for not equals, and the usual <, >, <=, and >= for the other four comparisons. The logical connectives are And, Or, and Not. The ampersand, &, is used for string concatenation. String constants are enclosed in double quotes, with “” representing the null string. As in C, Integers are used in place of Boolean variables, with zero representing false, and non-zero representing true. The constants True and False can be used. They have the numeric values –1, and 0 respectively. In addition to For loops there are also While loops which have the form “While <condition>”. The body of the While loop is terminated by a Wend statement. Code is written one statement per line. Under normal circumstances
there cannot be more than one statement per line, nor can a statement be split between
lines. (See the Visual Basic documentation for exceptions to this rule.) Capitalization is
not significant, so **Abc**, **ABC**, and **aBC** represent the same variable. References to
component properties have the form “<instance name>.<property name>”, with a period
separating the instance name and the property name.

Unfortunately, we still haven’t answered the most important question. Where do we
put our code?

As stated above, a Visual Basic program is passive; it responds only to events. Therefore, we must arrange for this code to be executed in response to some event. There
are many different kinds of events, but the simplest are those that correspond to
keystrokes and mouse clicks. We will arrange for our code to be executed in response to
a mouse click. There are many ways to do this, but the simplest is to provide a button so
that we have an obvious place to click the mouse, and some visual feedback for the
action. A button is a component just like the text box. To create a button, we click on
![Visual Basic 6.0](Visual Basic 6.0) or ![VB.NET](VB.NET) in the toolbox to the left of the Visual
Basic window. This will give us the cross-hairs cursor, which we will use to draw a
rectangle on the **Form1** window. The result is pictured in Figure 2.7. We have changed
the label of the button so that the label on the button reads “My Button” instead of
“Command1” or “Button1”. In Visual Basic 6.0, the **Caption** property is used for this
purpose, while in VB.NET, the **Text** property is used. The caption of the button is not the
same as the **name** of the button. The name of the button remains **Command1** or **Button1**.
Figure 2.7. Adding a Button.
We want our code to execute whenever somebody clicks this button. To accomplish this, simply double-click on the button. This will display the code window pictured in Figure 2.8. Our code must be placed between the header line that reads “Private Sub Command1_Click(…)” or “Private Sub Button1_Click(…)” and termination line “End Sub”. In Visual Basic, subroutines are declared using the Sub keyword followed by the name of the subroutine. The Private keyword is optional. Any parameters are placed in parentheses following the function name. Parameter declarations have the same form as a Dim statement without the Dim keyword, so MyStr As String, and MyVal As Integer, are legitimate parameter declarations. Multiple parameter declarations are separated by commas. If there are no parameters, a pair of empty parentheses follows the subroutine name. The subroutine body begins immediately following the Sub statement, and is terminated by an End Sub statement. Subroutines do not return values. (We will discuss function declarations later.) In Visual Basic 6.0, the name of the subroutine is used to associate it with the button Command1 and the Click event. In VB.NET, the Handles clause is used for this purpose. (The Handles clause permits one subroutine to be used for many different events. In Visual Basic 6.0, one subroutine can handle only one event.)
Figure 2.8. The Code Window.
Once we type our code into this window, we’re ready to go. All we need to do is go up to the tool bar and click on \(\rightarrow\). However, before we run our program, it is a good idea to save our project. To do this, go up to the tool bar and click on \(\rightarrow\). Unlike some compilers, Visual Basic 6.0 does not automatically save our changes before running our program. If our program were to blow up for some reason, Visual Basic itself might be terminated taking our changes with it! (This happens quite often when testing new components.) When saving our code, we will be prompted for a project-file name and a form-file name. We will first create a new directory named \textit{Properties} for our project, and then save the form file as \textit{Properties.frm} and the project file as \textit{Properties.vbp}. VB.NET \textit{will} save our changes before compiling.

Once our program starts, it will look as pictured in Figure 2.9.
To test the program, we type some text in the top text-box, and click on “My Button”.

The result should look something like Figure 2.10.
We now have our first working Visual Basic program.

2.4. Events

Now that we’re familiar with Visual Basic programming, we’re ready to try something more adventurous. For the next application we will create an application that makes use of events. Because event-handling is quite different in Visual Basic 6.0 and VB.NET, we will describe the two programs separately.

2.4.1. Visual Basic 6.0

First, we will create a window that has a button and a red circle. When the button is clicked, we will make the circle flash red and green for ten seconds. To accomplish this we will create a new project and will add three components to it. One of these components will be a button, which we already know how to do. The other two will be a Shape and a Timer. To add a shape, we must select from the tool box, and draw a
rectangle on the Form1 window. To add a timer, we must select \( \text{ } \) from the tool box, and either draw a rectangle or double-click on the Form1 window. (The size of the timer component cannot be changed.) We must change several properties of our shape component, which will be named Shape1. These properties are the Shape property, which must be set to 2 – Oval, the FillColor property which must be set to red (we choose the red color from the palette of colors that is offered to us) and we must set FillStyle to 0 – Solid. We then must adjust the boundaries of the oval so it looks round.

We will change the Caption property of the Command1 button to “Flash”, and we will check to make sure that the Interval property of the Timer1 component is set to zero. The end result should look something like Figure 2.11.
The key to this program is the correct management of the Timer1 component. This component has only one event called Timer, and only one important property called Interval. When the Interval property is set to something other than zero, the timer component will issue repeated Timer events at regular intervals. The interval at which the events are issued is determined by the Interval property. The value of this property is assumed to be the number of milliseconds between events. Thus if Interval property is set to 100, ten events are issued every second. If it is set to 1, one thousand events will be issued every second, if it is physically possible to do so. We will use the Timer events to flash the color of the circle. Clicking the button will start the process, and we will terminate the process after a fixed number of timer events have occurred. We will set the Interval property of Timer1 to 100 to cause the circle to change color ten times per second, and we will stop the process after 100 Timer events have occurred. We will use a global variable to count the events.

First let us double-click on the button and put in the code to start the process. This code will look as follows.

```vba
Private Sub Command1_Click()
    Timer1.Interval = 100
    Counter = 0
    Shape1.FillColor = 255
End Sub
```

In this subroutine, we first set the Interval property of Timer1 to 100, waking it up. Then we set the global variable Counter to zero, indicating that zero timer events have occurred. Finally, we set the FillColor property of Shape1 to red, to make sure we’re starting with the correct color. (The constant 255 specifies the color red. See the Visual
Basic Documentation for more information about how colors are represented.) It is necessary to define the global variable Counter. If we fail to do so, Visual Basic will create a declaration for us making Counter a local variable of type Variant. This is not what we want, so we must go to the top of the code window, and add the following line before the line reading “Private Sub Command1_Click( )”

```
Dim Counter As Integer
```

This global variable will be available to all event subroutines. Now, let us add the continuing behavior to the program. We do this by double-clicking on the Timer1 component, and creating the following subroutine.

```
Private Sub Timer1_Timer()
  If Counter >= 100 Then
    Shape1.FillColor = 255
    Timer1.Interval = 0
    Exit Sub
  End If
  If (Counter Mod 2) = 0 Then
    Shape1.FillColor = &HFF00&
  Else
    Shape1.FillColor = &HFF&
  End If
  Counter = Counter + 1
End Sub
```

In this subroutine, the first thing we test for is to see if enough events have occurred. If so, we set the shape color back to red, we set the Interval property of Timer1 to 0, turning it off, and exit the event subroutine using the Exit Sub statement. In the If statement, we decide whether to set the FillColor property of Shape1 to red or green. The Mod operator is actually the remainder operator. This expression will tell us whether the global variable Counter contains an even or an odd value. If the value is even, we set the
color to green otherwise we set it to red. This If statement also demonstrates how to specify hexadecimal constants. A hexadecimal constant begins with &H and ends with &. Legal digits are 0-9 as well as A, B, C, D, E, and F. The constant &HFF& is numerically equal to 255, while the constant &HFF00& is numerically equal to 65,280.

Finally, we will add one finishing touch to our program. To do this we double-click anywhere in the Form1 window except on a component. This will bring up the code window with the following subroutine.

```vbnet
Private Sub Form_Load()
End Sub
```

If we needed to initialize our program, this is where we would do it. However, this is not the event we want. What we want is the Unload event of the form. The Load event occurs when the program starts, while the Unload event occurs when the program window disappears. In some cases, the program will not terminate when its window disappears, so we will force this to happen by adding some code to the Unload event. First, we go to the top of the code window, where we see the word “Load” and click on the small button to the right of it. This displays the list of events for the form, as shown in Figure 2.12. We scroll this list down as far as it will go and click on the word “Unload”. This will create the subroutine for the Form Unload event. We can now delete the code for the Form_Load event, since it doesn’t do anything. We must make sure to also delete the End Sub statement that terminates the subroutine.
Figure 2.12. Selecting an Event.

In the *Form_Unload* subroutine we place one statement, the **End** statement. This statement always forces the termination of the program, regardless of where it is executed. The *Form_Unload* subroutine looks as follows

```vbnet
Private Sub Form_Unload(Cancel As Integer)
    End
End Sub
```

Note that the *Form_Unload* subroutine has one parameter. It is possible for events to have many different parameters, depending on the type of event. In this case, the *Cancel* parameter is used if it is necessary to refuse to close the window. To do this set *Cancel* to some non-zero value, such as **True**. (If you do this, make sure you have some other way of terminating your program, since the X button in the upper right will no longer work!)

We are now ready to test our program, but it is *absolutely necessary* to save it first. We are in some danger of sending our program into an infinite loop, and if this happens
using **Ctrl-Alt-Delete** may be the only way of getting out of it. (In this case the danger is not so great, but in other examples it will be.) If we **Ctrl-Alt-Delete** out of our program, Visual Basic will go with it, destroying any changes since the last save.

We will call our program “Events” and store it in its own directory with appropriately named form and project files. Then, and only then, will we run our program. This should produce a display similar to that of Figure 2.13. Clicking on the “Flash” button should cause the red circle to flash red and green.

![Figure 2.13. Running the Events Program.](image)

In addition to defining global variables, we can use the upper part of the code window to define general subroutines that are not associated with any event. To demonstrate how this works, we will change the pair of colors used to flash the circle. Instead of using explicit constants for red and green, we will use two more global variables, of type Long to contain the starting color and the flash color. Appropriately enough, we will call these **StartColor** and **FlashColor**. In the event routines, we will replace the explicit constants
with references to the correct global variables. Once we finish our changes, we will have
the following code.

```vbnet
Dim Counter As Integer
Dim StartColor As Long
Dim FlashColor As Long

Private Sub Command1_Click()
    Timer1.Interval = 100
    Counter = 0
    Shape1.FillColor = StartColor
End Sub

Private Sub Form_Unload(Cancel As Integer)
    End
End Sub

Private Sub Timer1_Timer()
    If Counter >= 100 Then
        Shape1.FillColor = StartColor
        Timer1.Interval = 0
        Exit Sub
    End If
    If (Counter Mod 2) = 0 Then
        Shape1.FillColor = FlashColor
    Else
        Shape1.FillColor = StartColor
    End If
    Counter = Counter + 1
End Sub
```

We will use a subroutine to set the values of these two global variables. This

```vbnet
Sub SetColors(SC As Long, FC As Long)
    FlashColor = FC
    StartColor = SC
End Sub
```

This code must immediately follow the line that reads “Dim FlashColor As Long” As

soon as we type the first line of this function, Visual Basic will automatically supply the

End Sub statement. Now we will add a Form_Load subroutine to make sure the global
variables start with the correct values. To create this, double-click anywhere in the Form1 window, except on a component, and then select Load from the list of events. This code also makes sure that \textit{Shape1} starts with the correct color. Note that when a subroutine is called in Visual Basic, the parameters are listed without parentheses. The call to \textit{SetColors} would be syntactically incorrect if we placed parentheses around the parameter list.

\begin{verbatim}
Private Sub Form_Load()
    SetColors &HFF&, &HFF00&
    Shape1.FillColor = StartColor
End Sub
\end{verbatim}

Now, to make things more interesting, we will add two more buttons to switch between the “standard” colors and two alternate colors. For the alternate colors we will use magenta and yellow, with magenta being the starting color. The hexadecimal constants for magenta and yellow are \&HFF00FF\& and \&HFFFF\& respectively. The button to select the alternate colors will be named \textit{Command2}, while the button for the standard colors will be named \textit{Command3}. The captions of these buttons will be \textit{Alternate} and \textit{Standard} respectively. The following is the \textit{Click} event code for these two buttons.

\begin{verbatim}
Private Sub Command2_Click()
    SetColors &HFF00FF, &HFFFF&
    Shape1.FillColor = StartColor
End Sub

Private Sub Command3_Click()
    SetColors &HFF&, &HFF00&
    Shape1.FillColor = StartColor
End Sub
\end{verbatim}
Figure 2.14 shows our new program just after it starts running, while Figure 2.15 shows it after clicking on the “Alternate” button.

**Figure 2.14. The Enhanced Events Program.**

In addition to defining our own subroutines, we can also define our own functions. The difference between a subroutine and a function is that a function returns a value. The following is a function that squares its two arguments, computes the sum and returns the result.

```plaintext
In addition to defining our own subroutines, we can also define our own functions.
```

```plaintext
The difference between a subroutine and a function is that a function returns a value. The
following is a function that squares its two arguments, computes the sum and returns the
result.
```
This function definition (if we had any use for it) must be placed at the top of the code window, just as the definition for *SetColors*. A function must have a declaration of its return type following the parameter list. The result is returned by assigning a value to the function name. The function body is terminated with an *End Function* statement. Similarly, to exit a function in the middle, an *Exit Function* statement is used.

To use this function, we would use a statement similar to the following.

\[ X = \text{Squareit}(3,5) \]

Unlike subroutines, it is necessary to enclose function parameters in parentheses.

By now you may be getting tired of these silly examples. The next few examples will allow us to do some things that, at least in some sense, are more useful than flashing colors.

### 2.4.2. VB.NET

There are many changes between Visual Basic 6.0 and VB.NET, and many of them will be evident as we develop the VB.NET version of the Events program. Because the shape control has been eliminated, we must either provide our own circle control, or use something else as a substitute. The easiest alternative is to change the circle to a rectangle and replace it with a label control that has no text. From the Windows Forms section of
the tool bar, we select the following two controls, the label \textbf{Label}, and the timer \textbf{Timer}. The label is drawn on the prototype window, just as any other control. You can do the same for the timer, but the icon will not appear in the prototype window. Instead it will be added to the list of non-visual controls below the prototype window. The usage of the component has not changed, but it no longer consumes valuable screen real estate. After adding a button, we end up with the design illustrated in Figure 2.16.

![Figure 2.16. The Design of the VB.NET Events Program.](image)

To create the red square, one simply adds a label control, sets the Text property to the empty string, sets the border to style to FixedSingle, and the BackColor to Red.

The Click event for the button requires some modification in VB.NET, since setting the Interval property to zero is no longer the method used to disable it. In fact, it is now
illegal to set the Interval property of a timer to zero. The Enabled property is now used to enable and disable the timer. We no longer use global variables in VB.NET. Anything that would have been specified as a global variable must now be specified as a member of the private class “Form1”. This also holds for event handlers and auxiliary functions.

Double click the button to add an event handler. Like all event handlers in VB.NET, this event handler has two properties. For now, we can safely ignore them. After adding the event handler Button1_Click, we add the counter variable immediately preceding the definition of the event handler. The BackColor property of the label is used to change the color of the square, but colors must be handled differently in VB.NET. When assigning to a color variable, one must use a member of the Color class. In this case we will use the constant member Red, which will set the background to red. The resultant code appears in Figure 2.17.

```vbnet
Dim Counter As Integer
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
    Timer1.Interval = 100
    Timer1.Enabled = True
    Counter = 0
    Label1.BackColor = Color.Red
End Sub
```

**Figure 2.17. Timer Enabling Code.**

To add the timer Tick event handler, we double click on the timer icon below the prototype window. This code is essentially the same as the Visual Basic 6.0 code, except we will disable the timer by setting the Enabled property to False, and we will use Color.Red and Color.Green as our color constants. The code for the Tick event is given in Figure 2.18.
Private Sub Timer1_Tick(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Timer1.Tick
    If Counter >= 100 Then
        Label1.BackColor = Color.Red
        Timer1.Enabled = False
        Exit Sub
    End If
    If (Counter Mod 2) = 0 Then
        Label1.BackColor = Color.Green
    Else
        Label1.BackColor = Color.Red
    End If
    Counter = Counter + 1
End Sub

**Figure 2.18. The VB.NET Tick Event Handler.**

Adding an event handler for the form Unload event is complicated by the fact that this event no longer exists, and has been replaced by the Closed event. Furthermore, the mechanism for creating event handlers for form events has also changed.

First, bring up the code window in any convenient fashion. Double clicking on the prototype window will create an event handler for the form Load event. Once the code window has been displayed, go to the left combo box, and display the list, as illustrated in Figure 2.19. Select (Base Class Events) from this list. Once this has been done, the list of available events will appear in the right-hand drop-down list as illustrated in Figure 2.20.

**Figure 2.19. Getting At the Form Events.**
To create the event handler for the Closed event, we select “Closed” from the right-hand side list. This will give us the event handler given in Figure 2.21, to which we have added the End statement.

```vbnet
Private Sub Form1_Closed(ByVal sender As Object, ByVal e As System.EventArgs) Handles MyBase.Closed
    End
End Sub
```

**Figure 2.21. The Event Handler for the Closed Event.**

Once this final event handler has been added, we can run the program. It is not necessary to save our work, because VB.NET automatically saves all files before compiling. (You may also notice that the compile step is now separate, so we can compile our program without running it.)

From here, the development of the enhanced version of the Events program proceeds much like the Visual Basic 6.0 enhanced version. The definition of the SetColors subroutine is much the same, except for the type declarations of the arguments. This function must be a member of the Form1 class. The variables FlashColor and StartColor must also be members of this class. Figure 2.22 shows the new definitions.
Dim FlashColor As Color
Dim StartColor As Color
Sub SetColors(ByVal SC As Color, ByVal FC As Color)
    FlashColor = FC
    StartColor = SC
End Sub

Figure 2.22. New Definitions.

One important difference between Visual Basic 6.0 and VB.NET is the manner in which functions must be called. In Visual Basic 6.0, subroutine arguments must be specified without parentheses, while function arguments must be specified with parentheses. In VB.NET both subroutine arguments and function arguments must be specified with parentheses. Note the call to SetColors in the event handlers of Figure 2.23.

Private Sub Button2_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button2.Click
    SetColors(Color.Magenta, Color.Yellow)
    Label1.BackColor = Color.Magenta
End Sub

Private Sub Button3_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button3.Click
    SetColors(Color.Red, Color.Green)
    Label1.BackColor = Color.Red
End Sub

Figure 2.23. Calling Subroutines in VB.NET.

The enhanced Events program is illustrated in Figure 2.24.
2.5. Methods

Methods are simply functions and subroutines that are implemented by a component. (Visual Basic distinguishes functions and subroutines by whether or not a value is returned.) To illustrate the use of a method, we will create our own web-browser. At first this may seem a bit fantastic, but such is the power of component-level programming.

Up to this point the only components we have used are Visual Basic controls. For this application, we will add an ActiveX control to our project. The first step, of course, is to start Visual Basic and create a new project. To add a new component to our project, we use the Components command of the Project menu, as illustrated in Figure 2.25.
The components command will cause the component selection dialog box to appear. As Figure 2.26 shows, this dialog box contains a list of all ActiveX controls installed on the system. (The list is generally large.) It is necessary for us to find the correct control, and click on the check box in front of it to add it to our project. As Figure 2.26 shows, we want to add Microsoft Internet Controls to our project.
Adding this new control to our project will cause a new icon to appear in the toolbox on the left of the Visual Basic window. This new icon resembles a globe, as Figure 2.27 shows.

In VB.NET, the procedure for adding the internet control is slightly different. To add this component to your project, right-click on the toolbox and select Customize Toolbox ..., as shown in Figure 2.28. Click on the COM Components tab and locate the entry
Microsoft Web Browser. Click the box in front of this entry, as shown in Figure 2.29, to add the browser control to your toolbox. The appearance of the control in the VB.NET toolbox is similar to that of Visual Basic 6.0, as Figure 2.30 shows. Once this control has been inserted into your toolbox, you can relocate it to the most convenient section.

![Figure 2.28. Customizing The VB.NET Toolbox.](image)
Figure 2.29. Selecting The Browser Control in VB.NET.

Figure 2.30. The VB.NET Browser Icon.
We select the new control from the toolbox, and draw a large instance of it. This control will be our web-browser window, so it is advantageous for it to be as large as possible. We will also add a button to our project which will contain the following code. (The VB.NET code follows the Visual Basic 6.0 code.)

```
Private Sub Command1_Click()
    WebBrowser1.Navigate "http://www.csee.usf.edu/~maurer"
End Sub
```

```
Private Sub Button1_Click(ByVal sender As System.Object, 
    ByVal e As System.EventArgs) Handles Button1.Click
    AxWebBrowser1.Navigate("http://www.csee.usf.edu/~maurer")
End Sub
```

The Navigate Method of the Web Browser control is used to display a web page. If your computer is connected to the Internet, you can substitute any URL you wish for http://www.csee.usf.edu/~maurer. If you are not connected to the Internet, you should change it so it refers to a file on your system. Figure 2.31 shows the finished VB.NET program, ready to run. The Visual Basic 6.0 version is similar.
Running our program and clicking on the **Navigate** button will produce the results shown in Figure 2.32. The hyperlinks are active, so it is possible to navigate to other pages without leaving our program. The ActiveX control that we added to our project is, in fact, the Internet Explorer display window. It can be used to display many different types of files.
You may have noticed that there are many other ActiveX controls available on the system. Figure 2.33 shows my Visual Basic 6.0 toolbox after adding about three fourths of my ActiveX controls to a project. (I don’t exactly collect these things either.)
2.6. Database Applications

No discussion of Visual Basic would be complete without an example of a database application. Database applications are what convinced the world that Visual Basic was a
good idea, and what ultimately created the component-level programming revolution. The techniques for creating database applications have been more or less the same in all versions of Visual Basic, starting with version 3.0. VB.NET, however, introduces an enhanced set of procedures that are significantly different from those used in earlier versions of Visual Basic. For simplicity, we will present only the Visual Basic 6.0 version of this program.

For an example of a database we will use a list of Hugo Award winning science fiction novels. Every year (or almost every year) since 1953, the World Science Fiction Convention has given an award for the best science fiction novel of the year. The list includes many of the classic science fiction works like “Rendezvous with Rama,” and “Starship Troopers.” I have created a database of all novels that have won this award along with the year and the name of the author. The database contains two tables, one with the names of the books, and one with the names of the authors. These tables are called Books and Authors respectively. The database also contains a query named List that ties these two tables together and provides a list of books with the names of the authors. Figure 2.34 shows the two tables, and Figure 2.35 shows the query. The database is stored in a file named “hawn.mdb”. (If you don’t have this file, you can download it from URL http://www.csee.usf.edu/~maurer.)
As always, the first step is to create a new Visual Basic project. After doing this, we add a data control to our project, by clicking on \( \text{Data Control} \) in the toolbox, and drawing a rectangle. We will change the caption of the resultant data control to *Hugo Awards*, but
we will leave its name as *Data1*. The result is shown in Figure 2.36. We must change two other properties of this data control. The first is the *DatabaseName* property. The value field of this property contains a small button when the property is selected. We need to click on this button and locate the file *hawn.mdb* on our hard drive. This associates the data control with the database file. The next property is *RecordSource*. This property cannot be changed until the *DatabaseName* property has a legal value. The value field of the *RecordSource* property also contains a button when the property is selected. When we click on this button we get a list of the tables and queries contained in *hawn.mdb*. We select the name *List* from this list. We are now ready to hook our data control to other controls.

---

![Figure 2.36. A Data Control Instance.](image)

60
As shown in Figure 2.35, the List query has four fields, *Year*, *Title*, *FirstName*, and *LastName*. We must provide four text boxes, one for each field. The arrangement of these text boxes is arbitrary. The arrangement illustrated in Figure 2.37 will do just fine.

![Figure 2.37. A Complete Database Application.](image)

It is not necessary to change the Text property of these four text boxes, since they will be initialized with data from the first record in the List query. It is necessary, however, to associate each textbox with a field in the database. This is done by hooking the textboxes to the data control using the *DataSource* and *DataField* properties of each textbox. Like most of the other properties we have dealt with, the value of the *DataSource* property will have a button in it when the property is selected. Clicking on this button will give us a list of all data controls in the project. In this case, we have only one data control named *Data1*, which we select for each of the four textboxes.
Once the DataSource property is set for each of the four textboxes, we can set the DataField property of each. The DataField property will also have a button in its value field. Clicking on this button will give a list of the fields in the List query. (Recall that Data1 is now hooked into the List query.) For Text2, we select the field Year, for Text1 we select the field Title, for Text3 we select the field FirstName, and for Text4 we select the field LastName. These changes will make no visible change in the appearance of the project. Once they are complete, however, we can run the project to obtain the display shown in Figure 2.38.

![Figure 2.38. The Database Application Running.](image)

We can use the buttons on the data control to scroll from one record to the next. As we do this, the contents of the four fields will change. This program is not just a data-display program, however. To demonstrate this, scroll forward to 1970, which will give us the display shown in Figure 2.39.
We will change the author’s last name by deleting the space. The result will be “LeGuin”. Now scroll forward to 1975. This will give us the display of Figure 2.40.

Notice how the change appears in this record as well. This is because there is really only one copy of the author’s name in the database, and changing it in one place will permanently change it everywhere it is displayed. Changes to any of the four text boxes will be permanently recorded in the database. If we wished, we could add buttons that allowed us to add and delete records. These changes are left as an exercise for you, the reader.
2.7. “Real” Programming in Visual Basic

Over the past several years there has been much discussion about whether or not Visual Basic is a “real” programming language, or whether it is just a prototyping language. This argument is silly, because Visual Basic is obviously a real programming language. It supports loops, if statements, assignments, functions, and subroutines. Any language that does that is a real programming language.

The real question on people’s minds is this. When creating a new application, can a significant amount of the code be written in Visual Basic, or must I write nearly everything in C++ or some other language, and just use Visual Basic to glue things together? My vote would be in favor of developing a significant portion of the code in Visual Basic. Even if there is a performance penalty, it may not be noticeable. If the performance penalty is too great, one can always rewrite the most time consuming portions of the code in another language. To prove that you can do “real” programming in Visual Basic, we will develop a “real” program directly in Visual Basic. I’ve put the

![Figure 2.40. The Second Le Guin Award.](image)
word “real” in quotes because our “real” program is really just an exercise taken from an introductory programming book.

The problem we will solve is finding all prime numbers between 1 and 1000 using the Sieve of Eratosthenes. We will start with 2 and cross out all multiples of 2, then cross out all multiples of 3, and so forth until we have the desired list of prime numbers. Then we will display each prime number on a separate line in our textbox. The only new thing in this program is setting the MultiLine property of the textbox to True and the ScrollBars property to “3 – Both”.

The actual program will be inserted into the Click event of our button. The following is our program.
Private Sub Command1_Click()
    Dim Sieve(1000) As Integer
    Dim I As Integer, J As Integer

    'Set up a list of -Potential- primes
    'Zero will indicate that the corresponding index is not prime
    For I = 1 To 1000
        Sieve(I) = I
    Next I
    'One is not prime
    Sieve(1) = 0

    'The following loop will search for a prime and then cross out
    'All of its non-unit multiples. We begin the search with 1.
    I = 1
    'Using I*I instead of I to terminate the search works,
    'And is more efficient than using I.
    While I * I <= 1000
        'Find the next prime, if any
        While I * I <= 1000 And Sieve(I) = 0
            I = I + 1
        Wend
        'if we found a prime
        If I * I <= 1000 Then
            'skip the prime itself
            J = I + I
            'cross out all non-unit multiples of the prime
            While J <= 1000
                Sieve(J) = 0
                J = J + I
            Wend
            'continue searching for primes with the next value of I
            I = I + 1
        End If
    Wend
    'Blank out the textbox
    Text1.Text ="
    For I = 1 To 1000
        'if this number is prime, convert it to a string and add it
        'to the textbox. Chr(13) is a return Chr(10) is a linefeed.
        If Sieve(I) <> 0 Then
            Text1.Text = Text1.Text & Format(Sieve(I)) & Chr(13) & Chr(10)
        End If
    Next I
End Sub

Running the “real” program will produce the results shown in Figure 2.41. On my computer the prime numbers appear instantaneously as soon as the button is clicked.
2.8. Conclusion

A number of visual languages have arisen over the past several years, of which Visual Basic is, by far, the most successful. The Visual Basic component interface, which is now more-or-less standard, is the Properties/Methods/Events interface. Once you have familiarized yourself with Visual Basic, you will have a good idea of what the external interface of a component is like. In the remaining chapters we will describe how to construct this interface, and how to construct the internals of a component.

2.9. Exercises

1. Create a Visual Basic database program to index your CD collection. Make sure that the program will work for both classical and popular music.

2. Use the Visual Basic image control to create a file viewer. Your file viewer should list the contents of a directory, and allow a user to select a file. You must then load the selected file into the image control to display it.
3. Accessing the following web page will install the “cards” ActiveX control on your computer:  http://www.csee.usf.edu/~maurer/vdal/cardgames/stowers.htm

Use this control to create an implementation of your favorite solitaire game.