Part 1

GUI Design and Event-Driven Programming

This Section:

◆ Chapter 1: Getting Started with Visual Basic 2005
◆ Chapter 2: Visual Basic: The Language
◆ Chapter 3: Procedures and Modules
Chapter 1

Getting Started with Visual Basic 2005

I’m assuming that you have installed one of the several versions of Visual Studio 2005. For this book, I used the Professional Edition of Visual Studio, but just about everything discussed in this book applies to the Standard Edition as well. Some of the features of the Professional Edition that are not supported by the Standard Edition concern database tools, which are discussed in Part 5 of this book.

You may have even already explored the new environment on your own, but this book starts with an overview of Visual Studio and its basic tools. It doesn’t even require any knowledge of VB 6, just some familiarity with programming at large. As you already know, Visual Basic 2005 is just one of the languages you can use to build applications with Visual Studio 2005. I happen to be convinced that this is also the simplest, most convenient language, but this isn’t really the issue; I’m assuming you have your reasons to code in VB, or else you wouldn’t be reading this book. What you should keep in mind is that Visual Studio 2005 is an integrated environment for building, testing, debugging, and deploying a variety of applications: Windows applications, Web applications, classes and custom controls, and even console applications. It provides numerous tools for automating the development process, visual tools to perform many common design and programming tasks and more features than any author would hope to cover.

The first thing you must learn is the environment in which you’ll be working from now on. In the first chapter of this book, you’ll familiarize yourself with the integrated development environment (IDE) and how its tools allow you to quickly design the user interface of your application, as well as how to program the application. It will be awhile before you explore all the items of the IDE and I will explain the various items as needed in the course of the book. In this chapter, we’ll look at the basic components of the IDE needed to build simple Windows applications.

The Integrated Development Environment

Visual Studio 2005 is an environment for developing Windows and Web applications. Visual Basic 2005 is just one of the languages you can use to program your applications. The language is only one aspect of a Windows application. The visual interface of the application isn’t tied to a specific language, and the same tools you’ll use to develop your application’s interface will also be used by all programmers, regardless of the language they’ll use to code the application.

To simplify the process of application development, Visual Studio provides an environment that’s common to all languages, which is known as integrated development environment (IDE). The purpose of the IDE is to enable the developer to do as much as possible with visual tools, before writing code. The IDE provides tools for designing, executing, and debugging your applications. It’s your second desktop, and you’ll be spending most of your productive hours in this environment.
CHAPTER 1  GETTING STARTED WITH VISUAL BASIC 2005

The Start Page

When you run Visual Studio 2005 for the first time, you will be prompted to select the type of projects you plan to build with VS, so that the environment can be optimized for the specific type of development. I’m assuming that you have initially selected the Visual Basic Development settings, which will optimize your copy of VS for building Windows and Web applications with Visual Basic 2005. You can always change these settings as explained at the end of this section. After the initial configuration, you will see a window similar to the one shown in Figure 1.1.

**Figure 1.1**
This is what you’ll see when you start Visual Studio for the first time.

**Recent Projects**  Here you see a list of the projects you opened with Visual Studio most recently and you can select the one you want to open again—chances are that you will continue working on the same project as the last time. Each project’s name is a hyperlink, and you can open it by clicking its name. At the bottom of the Recent Projects section there are two hyperlinks for opening another project and for creating new projects.

**Visual Studio Developer News**  This section is a browser window and it displays an MSDN page when the computer is connected to the Internet. After the release of the product you can view on this section news about Visual Studio, the supported languages, articles, and other interesting bits of information.

**Getting Started**  This section contains links to basic programming tasks in the product’s documentation.

Most developers will skip the Start Page. To do so, open the Tools menu and select the Import and Export Settings command to start a configuration wizard. On the first dialog box of the wizard check the option “Reset All Settings” and click the Next button. The next screen of the wizard prompts you
for the location where the new settings will be saved, so that Visual Studio can read them every time it starts. Leave the default location as-is and click Next again to see the last screen of the wizard, in which you’re prompted to select a default collection of settings. This collection depends on the options you’ve installed on your system. I installed Visual Studio 2005 with Visual Basic only on my system, and I was offered the following options: Business Intelligence, General Development, Visual Basic Development, and Web Development. For the default configuration of my copy of Visual Studio, and for the purposes of this book, I chose the Visual Basic Development settings, so that Visual Studio could optimize the environment for a typical VB developer. Click the Finish button to see a summary of the process and then close the wizard.

Starting a New Project

At this point, you can create a new project and start working with Visual Studio. To best explain the various items of the IDE, we will build a simple form—it’s not even an application. The form is the window of your application—it’s what users will see on their desktop when they run your application.

Open the File menu and select New Project, or click the Create Project/Solution in the Start Page. In the New Project dialog box that pops up (see Figure 1.2), you see a list of project types you can create with Visual Studio. Select the Windows Application template, and Visual Studio suggests the name WindowsApplication1 as the project name. Change it to MyTestApplication and then click the OK button to create the new project.

What you see now is the Visual Studio IDE displaying the Form Designer for a new project, as shown in Figure 1.3. The new project contains a Form already: the Form1 component in the Solution Explorer. The main window of the IDE is the Form Designer, and the gray surface on it is the window of your new application in design mode. Using the Form Designer, you’ll be able to design the visible interface of the application (place various components of the Windows interface on the form and set their properties) and then program the application.

The project hasn’t been saved anywhere yet. You can save it at any time by selecting the Save All command on the File menu. Alternatively, you can follow the instructions of this section to build the sample project, run it, and then close Visual Studio (or start a new project) and discard your first sample. As you will see shortly you can instruct Visual Studio to create a new folder for each new project the moment you create it.

Figure 1.2
The New Project dialog box
CHAPTER 1 GETTING STARTED WITH VISUAL BASIC 2005

The default environment is rather crowded, so let’s hide a few of the toolbars that we won’t use in the projects of the first few chapters. You can always show any of the toolbars at any time. Open the View menu and select Toolbars. You see a submenu with 28 commands that are toggles. Each command corresponds to a toolbar, and you can turn the corresponding toolbar on or off by clicking one of the commands in the Toolbar submenu. Turn off all the toolbars except for the Layout and Standard toolbars.

The last item in the Toolbars submenu is the Customize command, which leads to a dialog box in which you can specify which of the toolbars and which of the commands you want to see. Use this menu after you have established a work pattern to customize the environment for the way you want to work with Visual Studio.

**Figure 1.3**
The integrated development environment of Visual Studio 2005

**Using the Windows Form Designer**

To design the form, you must place on it all the controls you want to display to the user at runtime. The controls are the components of the Windows interface (buttons, text boxes, radio buttons, lists, and so on). Open the Toolbox by moving the pointer over the Toolbox tab at the far left; the Toolbox pulls out, as shown in Figure 1.4. This Toolbox contains an icon for each control you can use on your form.

The controls are organized into groups according to each control’s function on the interface. In the first part of the book, we’ll create simple Windows applications and we’ll use the controls on the Common Controls tab. When you develop Web applications, you will see a different set of icons on the Toolbox.

To place a control on the form, you can double-click the icon of the control. A new instance with a default size will be placed on the form. Then you can position and resize it with the mouse. Or you can select the control from the Toolbox with the mouse and then move the mouse over the form and draw the outline of the control. A new instance of the control will be placed on the form, and it will fill the rectangle you specified with the mouse. Start by placing a TextBox control on the form.
The control’s properties will be displayed in the Properties window (see Figure 1.5). This window, at the far left edge of the IDE, displays the properties of the selected control on the form. If the Properties window is not visible, open the View menu and select Properties, or press F4. If no control is selected, the properties of the selected item in the Solution Explorer are displayed.

In the Properties window, also known as the Property Browser, you see the properties that determine the appearance of the control and (in some cases) its function. The properties are organized in categories according to their role. The properties that determine the appearance of the control are listed alphabetically under the header Appearance, the properties that determine the control’s behavior are listed alphabetically under the header Behavior, and so on. You can click the AZ button on the window’s title bar to display all properties in alphabetical order. After you familiarize yourself with the basic properties, you will most likely switch to the alphabetical list.

Figure 1.4
Windows Forms Toolbox
of the Visual Studio IDE

Figure 1.5
Properties of a TextBox control
CHAPTER 1  GETTING STARTED WITH VISUAL BASIC 2005

REARRANGING THE IDE WINDOWS

As soon as you place a control on the form, the Toolbox retracts to the left edge of the designer. You can fix this window on the screen by clicking the icon with the pin on the Toolbox’s toolbar.

You can easily rearrange the various windows that make up the IDE by moving them around with the mouse. Move the pointer to a window’s title bar, press the button, and move the window around while holding the mouse button down. As you move the window, eight buttons with arrows appear on the screen, indicating the area where the window can be docked. Keep moving the window until the pointer hovers over one of these buttons, and the docking area appears in semitransparent blue color. Find the desired docking location for the window and release the mouse. If you release the mouse while the pointer is not on top of an arrow, the window is not docked; instead, it remains at the current location as a floating window. Most developers would rather work with docked windows, and the default positions of the IDE windows are quite convenient. If you want to open even more windows and arrange them differently on the screen, use the docking feature of the IDE to dock the additional windows.

Locate the TextBox control’s Text property and set it to “My TextBox Control” by entering the string (without the quotes) into the box next to property name. The control’s Text property is the string that appears in the control (the control’s caption), and most controls have a Text property.

Then locate its BackColor property and select it with the mouse. A button with an arrow appears next to the current setting of the property. Click this button, and you see a dialog box with three tabs (Custom, Web, and System), as shown in Figure 1.6. In this dialog box, you can select the color that will fill the control’s background. Set the control’s background color to yellow and notice that the control’s appearance changes on the form.

Locate the control’s Font property. You can click the plus sign in front of the property name and set the individual properties of the font, or you can click the button with the ellipsis to invoke the Font dialog box. Here you can set the control’s font and its attributes and then click OK to close the dialog box. Set the TextBox control’s Font property to Verdana, 14 points, bold. As soon as you close the Font dialog box, the control on the form is adjusted to the new setting.

Figure 1.6
Setting a color property in the Properties dialog box
There’s a good chance that the string you assigned to the control’s Text property won’t fit in the control’s width when rendered in the new font. Select the control on the form with the mouse, and you will see eight handles along its perimeter. Rest the pointer over any of these handles, and it will assume a shape indicating the direction in which you can resize the control. By the way, if you click somewhere on the form and resize it with its handles. Make the control long enough to fit the entire string. If you have to, resize the form as well. Click somewhere on the form and when the handles along its perimeter appear, resize it with the mouse. Some controls, such as the Label, the Button and the CheckBox controls, support the AutoSize property, which determines whether the control is resized automatically to accommodate its caption. The TextBox control, as well as many others, doesn’t support the AutoSize property.

If you attempt to make the control tall enough to accommodate a few lines of text, you’ll realize that you can’t change the control’s height. By default, the TextBox control accepts a single line of text. So far, you manipulated properties that determine the appearance of the control. Now you’ll change a property that determines not only the appearance but also the function of the control. Locate the Multiline property. Its current setting is False. Expand the list of available settings and change it to True. (You can also change it by double-clicking the name of the property. This action toggles the True/False settings.) Switch to the form, select the TextBox control, and make it as tall as you wish.

The Multiline property determines whether the TextBox control can accept one (if Multiline = False) or more (if Multiline = True) lines of text. Set this property to True, go back to the Text property, set it to a long string, and press Enter. The control breaks the long text into multiple lines. If you resize the control, the lines will change, but the entire string will fit in the control because the control’s WordWrap property is True. Set it to False to see how the string will be rendered on the control.

Multiline TextBox controls usually have a vertical scroll bar so that users can quickly locate the section of the text they’re interested in. Locate the control’s ScrollBars property and expand the list of possible settings by clicking the button with the arrow. This property’s settings are None, Vertical, Horizontal, and Both. Set it to Vertical, assign a very long string to its Text property, and watch how the control handles the text. At design time, you can’t scroll the text on the control; if you attempt to move the scroll bar, the entire control will be scrolled. The scroll bar will work as expected at runtime (it will scroll the text vertically).

You can also make the control fill the entire form. Start by deleting all other controls you may have placed on the form and then select the multiline TextBox. Locate the Dock property in the Properties window and keep double-clicking the name of the property until its setting changes to Fill (you’ll learn a lot more about docking controls in Chapter 4). The TextBox control fills the form and is resized as you resize the form, both at design time and runtime.

To examine the control’s behavior at runtime, press F5. The application will be compiled, and a few moments later, a window filled with a TextBox control will appear on the desktop (like the one shown in Figure 1.7). This is what the users of your application would see (if this were an application worth distributing, of course).

Enter some text on the control, select part of the text, and copy it to the Clipboard by pressing Ctrl+C. You can also copy text from any other Windows application and paste it on the TextBox control. When you’re done, open the Debug menu and select Stop Debugging. This will terminate your application’s execution, and you’ll be returned to the IDE. The Stop Debugging command is also available as a button with a blue square icon on the toolbar. Finally, you can stop the running application by clicking the Close button on the application’s window.
The design of a new application starts with the design of the application’s form, which is the application’s user interface, or UI. The design of the form determines to a large extent the functionality of the application. In effect, the controls on the form determine how the application will interact with the user. The form itself is a prototype, and you can demonstrate it to a customer before even adding a single line of code. By placing controls on the form and setting their properties you’re implementing a lot of functionality before coding the application. The TextBox control with the settings discussed in this section is a functional text editor.

Your First VB Application

In this section, we’ll develop a very simple application to demonstrate not only the design of the interface, but also the code behind the interface. We’ll build an application that allows the user to enter the name of their favorite programming language, and the application will evaluate the choice. Objectively, VB is a step ahead of all other languages, and it will receive the best evaluation. All other languages get the same grade: good, but not VB. The project you will build in this section is called WindowsApplication1, and you can find it in this chapter’s folder on the Web site. You can open the project you downloaded from the book’s Web site and examine it, but I suggest you follow the steps outlined in this paragraph to build the project from scratch. Start a new project and use the default name, WindowsApplication1, and place a TextBox and a Button control on the form. Use the mouse to position and resize the controls on the form, as shown in Figure 1.8.

**Figure 1.7**
A TextBox control displaying multiple text lines
Now you must insert some code to evaluate the user’s favorite language. Windows applications are made up of small code segments, called event handlers, which react to specific actions such as the click of a button, the selection of a menu command, the click of a check box, and so on. In the case of our example, we want to program the action of clicking the button. When the user clicks the button, we want to execute some code that will display a message.

To insert some code behind the Button control, double-click the control, and you’ll see the code window of the application, which is shown in Figure 1.9. The line “Private...” is too long to fit on the printed page, so I inserted a line continuation character (an underscore) to break it into two lines. When a line is too long, you can break it into two (or more) lines by inserting the line continuation character. Alternatively, you can turn on the WordWrap feature of the editor (you’ll see shortly how to adjust the editor’s properties). Notice that I also inserted quite a bit of space before the second half of the first code line. It’s customary to indent continued lines so that they can be easily distinguished from the other lines.

The editor opens a subroutine, which is delimited by the following statements:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
End Sub
```

At the top of the main pane of the Designer, you will see two tabs named after the form: In Figure 1.9, they’re the **Form1.vb [Design]** tab and the **Form1.vb** tab. The first tab is the Windows Form Designer (in which you build the interface of the application with visual tools), and the second is the code editor (in which you insert the code behind the interface). At the top of the code editor, which is what you see in Figure 1.9, are two ComboBoxes. The one on the left contains the names of the controls on the form. The other one contains the names of events each control recognizes. When you select a control (or an object, in general) in the left list, the other list’s contents are adjusted accordingly. To program a specific event of a specific control, select the name of the control in the first list (the Objects list) and the name of the event in the right list (the Events list).

The Click event happens to be the default event of the Button control, so when you double-click a Button on the form, you’re taken to the Button1_Click subroutine. This subroutine is an event handler, which is invoked automatically every time an event takes place. The event of interest in our example is the Click event of the Button1 control. Every time the Button1 control on the form is clicked, the Button1_Click subroutine is activated. To react to the Click event of the button, you must insert the appropriate code in this subroutine.

**Figure 1.9**
Outline of a subroutine that handles the Click event of a Button control
The definition of the event handler can’t be modified; this is the event handler’s signature (the arguments it passes to the application). All event handlers in VB2005 pass two arguments to the application: the `sender` argument, which is an object that represents the control that fired the event, and the `e` argument, which provides additional information about the event.

The name of the subroutine is made up of the name of the control, followed by an underscore and the name of the event. This is just the default name, and you can change it to anything you like (such as `EvaluateLanguage`, for this example, or `StartCalculations`). What makes this subroutine an event handler is the keyword `Handles` at the end of the statement. The `Handles` keyword tells the compiler which event this subroutine is supposed to handle. `Button1.Click` is the Click event of the `Button1` control. If there were another button on the form, the `Button2` control, you’d have to write code for a subroutine that would handle the `Button2.Click` event. Each control recognizes many events and you can provide a different event handler for each control and event combination. Of course, we never program every possible event for every control.

**NOTE** The controls have a default behavior and handle the basic events on their own. The `TextBox` control knows how to handle keystrokes. The `CheckBox` control (a small square with a check mark) changes state by hiding or displaying the check mark every time it’s clicked. The `Scrollbar` control moves its indicator (the button in the middle of the control) every time you click one of the arrows at the two ends. Because of this default behavior of the controls, you need not supply any code for the events of most controls on the form.

Rename `Button1_Click` subroutine to `EvaluateLanguage`. If you change the name of the control after you have inserted some code in an event handler, the name of the event handled by the subroutine will be automatically changed. The name of the subroutine, however, won’t change.

Let’s add some code to the Click event handler of the `Button1` control. When this button is clicked, we want to examine the text on the text box. If it’s “Visual Basic,” display a message; if not, we’ll display a different message. Insert the lines of Listing 1.1 between the `Private Sub` and `End Sub` statements. (I’m showing the entire listing here; there’s no reason to retype the first and last statements.)

**LISTING 1.1:**  Processing a User-Supplied String

```vbnet
Private Sub EvaluateLanguage_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
    Dim language As String
    language = TextBox1.Text
    If language = "Visual Basic" Then
        MsgBox("We have a winner!")
    Else
        MsgBox(language & " is not a bad language.")
    End If
End Sub
```
Here’s what this code does. First, it assigns the text of the text box control to the variable `language`. A variable is a named location in memory where a value is stored. Variables are where we store the intermediate results of our calculations when we write code. All variables are declared with Dim statement and they have a name and a type. The `language` variable’s type is String because we intend to store text in this variable.

Then the program compares the value of the `language` variable to the literal “Visual Basic”, and depending on the outcome of the comparison, it displays one of two messages. The MsgBox() function displays the specified message in a small window with the OK button, as shown in Figure 1.8. Users can view the message and then click the OK button to close the message box.

Even if you’re not familiar with the syntax of the language, you should be able to understand what this code does. Visual Basic is the simplest of the languages supported by Visual Studio 2005, and we will discuss the various aspects of the language in detail in the following chapters. In the meantime, you should try to understand the process of developing a Windows application: how to build the visible interface of the application and how to program the events to which you want your application to react.

**TIP** The default names of the controls you place on a form are quite generic, and you should change them to something more meaningful. I usually prefix the control names with a few characters that indicate the control’s type (such as txt, lbl, btn, and so on), followed by a meaningful name. Names such as txtLanguage and btnEvaluate make your code far more readable.

The code of our first application isn’t very robust. If the user doesn’t enter the string with the exact spelling shown in the listing, the comparison will fail. We can convert the string to uppercase and then compare it with “VISUAL BASIC” to eliminate differences in case. To convert a string to uppercase, use the ToUpper method of the String class. The following expression returns the string stored in the `language` variable, converted to uppercase:

```vbnet
language.ToUpper
```

We should also take into consideration the fact that the user may enter “VB” or “VB 2005”, and so on. In the following section, we’ll further improve our application. You never know what users may throw at your application, so whenever possible we try to limit their response to the number of available choices. In our case, we can display the names of certain languages (the ones we’re interested in) and force the user to select one of them. One way to display a limited number of choices is to use a ComboBox control. In the following section, we’ll revise our sample application so that users won’t have to enter the name of the language. We’ll force them to select their favorite language from a list so that we won’t have to validate the string supplied by the user.

**Making the Application More User-Friendly**

Start a new project: the WindowsApplication2 project. As soon as the project is created, open the File menu and select the Save All command to save the project. When the Save Project dialog box appears, click the Browse button to select the folder where the project will be saved. On the Project Location dialog box that will appear select an existing folder or create a new folder such as “MyProjects” or “VB.NET Samples.”
Open the Toolbox and double-click the icon of the ComboBox tool. A ComboBox control will be placed on your form. Now place a Button control on the form and position it so that your form looks like the one shown in Figure 1.10. Then set the button’s Text property to “Evaluate My Choice.”

We must now populate the ComboBox control with the valid choices. Select the ComboBox control on the form by clicking it with the mouse and locate its Items property in the Properties window. The setting of this property is “Collection,” which means that the Items property doesn’t have a single value; it’s a collection of items (strings, in this case). Click the ellipsis button and you’ll see the String Collection Editor dialog box, as shown in Figure 1.11.

**Figure 1.10**
Displaying options on a ComboBox control

**Figure 1.11**
Click the ellipsis button next to the Items property of a ComboBox to see the String Collection Editor dialog box.

The main pane on the String Collection Editor dialog box is a TextBox, in which you can enter the items you want to appear in the ComboBox control at runtime. Enter the following strings, one per row and in the order shown here:

- C++
- C#
- Java
- Visual Basic
- Cobol
Click the OK button to close the dialog box. The items will not appear on the control at design time, but you will see them when you run the project. Before running the project, set one more property. Locate the ComboBox control’s Text property and set it to “Select Your Favorite Language.” This is not an item of the list; it’s the string that will initially appear on the control.

You can run the project now and see how the ComboBox control behaves. Press F5 and wait for a few seconds. The project will be compiled, and you’ll see its form on your desktop, on top of the Visual Studio window. I’m sure you know how the ComboBox control behaves in a typical Windows application, and our sample application is no exception. You can select an item on the control, either with the mouse or with the keyboard. Click the button with the arrow to expand the list and then select an item with the mouse. Or press the down and up arrow keys to scroll through the list of items. The control isn’t expanded, but each time you click an arrow button, the next or previous item in the list appears on the control. Press the Tab key to move the focus to the Button control and press spacebar to emulate a Click event (or simply click the Button control).

We haven’t told the application what to do when the button is clicked, so let’s go back and add some code to the project. Stop the application by clicking the Stop button on the toolbar (the solid black square) or by selecting Debug ➤ Stop Debugging from the main menu. When the form appears in design mode, double-click the button, and the code window will open, displaying an empty Click event handler. Insert the statements shown in Listing 1.2 between the Private Sub and End Sub statements.

### Listing 1.2: The Revised Click Event Handler

```vbnet
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
    Dim language As String
    language = ComboBox1.Text
    If language = "Visual Basic" Then
        MsgBox("We have a winner!")
    Else
        MsgBox(language & ", is not a bad language.")
    End If
End Sub
```

When the form is first displayed, a string that doesn’t correspond to a language is displayed in the ComboBox control. We can preselect one of the items from within our code when the form is first loaded. When a form is loaded, the Load event of the Form object is raised. Double-click somewhere on the form, and the editor will open the form’s Load event handler:

```vbnet
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
End Sub
```
Enter the following code to select the item “Visual Basic” when the form is loaded:

```vbnet
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    ComboBox1.SelectedIndex = 3
End Sub
```

`SelectedIndex` is a property of the ComboBox control that determines the selected item. You can use it to retrieve the index of the selected item on the list, or set it to select an item on the list from within your code. You may have noticed that you need not compare the control’s Text property to a string; you can also compare the `SelectedIndex` property to the value that corresponds to the index of the item “Visual Basic,” with a statement such as the following:

```vbnet
If ComboBox1.SelectedIndex = 3 Then ...
```

Of course, if you insert or remove items from the list, you must edit the code accordingly.

The controls on the Toolbox are more than nice pictures we place on our forms. They encapsulate a lot of functionality and they expose properties that allow you to adjust their appearance and their functionality. Most properties are usually set at design time, but quite frequently we change the properties of various controls from within our code.

Now that you’re somewhat familiar with the process of building Windows applications, and before we look into any additional examples, I will go quickly through the components of the Visual Studio IDE.

**The IDE Components**

The IDE of Visual Studio 2005 contains numerous components, and it will take you awhile to explore them. It’s practically impossible to explain in a single chapter what each tool, window, and menu command does. We’ll discuss specific tools as we go along and as the topics we discuss get more and more advanced. In this section, I will go through the basic items of the IDE—the ones we’ll use in the following few chapters to build simple Windows applications.

**The IDE Menu**

The IDE menu provides the following commands, which lead to submenus. Notice that most menus can also be displayed as toolbars. Also, not all options are available at all times. The options that cannot possibly apply to the current state of the IDE are either invisible or disabled. The Edit menu is a typical example. It’s quite short when you’re designing the form and quite lengthy when you edit code. The Data menu disappears altogether when you switch to the code editor—you can’t use the options of this menu while editing code.

**FILE MENU**

The File menu contains commands for opening and saving projects or project items, as well as the commands for adding new or existing items to the current project.

**EDIT MENU**

The Edit menu contains the usual editing commands. Among the commands of the Edit menu are the Advanced command and the IntelliSense command. Both commands lead to submenus, which are discussed next.
**Edit ➤ Advanced Submenu**

The more interesting options of the Edit ➤ Advanced submenu are the following. Notice that the Advanced submenu is invisible while you design a form visually and appears when you switch to the code editor.

**View White Space**  Space characters (necessary to indent lines of code and make it easy to read) are replaced by periods.

**Word Wrap**  When a code line’s length exceeds the length of the code window, it’s automatically wrapped.

**Comment Selection/Uncomment Selection**  Comments are lines you insert between your code’s statements to document your application. Every line that begins with a single quote is a comment; it is part of the code, but the compiler ignores it. Sometimes, we want to disable a few lines from our code, but not delete them (because we want to be able to restore them). A simple technique to disable a line of code is to “comment it out” (insert the comment symbol in front of the line). This command allows you to comment (or uncomment) large segments of code in a single move.

**Edit ➤ IntelliSense Submenu**

The Edit ➤ IntelliSense menu item leads to a submenu with five options, which are described next. IntelliSense is a feature of the editor (and of other Microsoft applications) that displays as much information as possible, whenever possible. When you type the name of a control and the following period, IntelliSense displays a list of the control’s properties and methods, so that you can select the desired one, rather than guessing its name. When you type the name of a function and the opening parenthesis, IntelliSense will display the syntax of the function—its arguments. The IntelliSense submenu includes the following options.

**List Members**  When this option is on, the editor lists all the members (properties, methods, events, and argument list) in a drop-down list. This list will appear when you enter the name of an object or control followed by a period. Then you can select the desired member from the list with the mouse or with the keyboard. Let’s say your form contains a control named TextBox1 and you’re writing code for this form. When you enter the name of the control followed by a period (“TextBox1.”), a list with the members of the TextBox control will appear (as seen in Figure 1.12).

In addition, a description of the selected member is displayed on a ToolTip box, as you can see in the same figure. Select the Text property and then enter the equal sign, followed by a string in quotes like the following:

```csharp
TextBox1.Text = 'Your User Name'
```

If you select a property that can accept a limited number of settings, you will see the names of the appropriate constants in a drop-down list. If you enter the following statement, you will see the constants you can assign to the property:

```csharp
TextBox1.TextAlign =
```

As shown in Figure 1.13, they are the values HorizontalAlignment.Center, HorizontalAlignment.Right, and HorizontalAlignment.Left. Again, you can select the desired value with the mouse. The drop-down list with the members of a control or object (the Members List) remains open until you type a terminator key (the Escape or End key) or switch to another window.
Parameter Info  While editing code, you can move the pointer over a variable, method, or property and see its declaration in a yellow popup box. You can also jump to the variable’s definition or the body of a procedure by selecting Go To Definition from the context menu that will appear if you right-click the variable or method name in the code window.

Quick Info  This is another IntelliSense feature that displays information about commands and functions. When you type the opening parenthesis following the name of a function, for example,
the function’s arguments will be displayed in a ToolTip box (a yellow horizontal box). The first argument appears in bold font; after entering a value for this argument, the next one will appear in bold. If an argument accepts a fixed number of settings, these values will appear in a drop-down list, as explained previously.

**Complete Word**  The Complete Word feature enables you to complete the current word by pressing Ctrl+spacebar. For example, if you type “TextB” and then press Ctrl+spacebar, you will see a list of words that you’re most likely to type (TextBox, TextBox1, and so on).

**Insert Snippet**  This command opens the Insert Snippet window at the current location on the code editor window. Code snippets, which are an interesting feature of Visual Studio 2005, are discussed in the section “Using Code Snippets and the My Object” later in this chapter.

**Edit ➤ Outlining Submenu**

A practical application contains a substantial amount of code in a large number of event handlers and custom procedures (subroutines and functions). To simplify the management of the code window, the Outlining submenu contains commands that collapse and expand the various procedures. Let’s say you’re done editing the Click event handlers of a number of buttons on the form. You can reduce these event handlers to a single line with the name of the procedure and a plus sign in front of them. You can expand a procedure’s listing at any time by clicking the plus sign in front of its name. When you do so, a minus sign appears in front of the procedure’s name, and you can click it to collapse the body of the procedure again. The Outlining submenu contains commands to handle the outlining of the various procedures, or turn off outlining and view the complete listings of all procedures. You will use these commands as you write applications with substantial amounts of code.

**View Menu**

This menu contains commands to display any toolbar or window of the IDE. You have already seen the Toolbars menu (in the “Starting a New Project” section). The Other Windows command leads to submenu with the names of some standard windows, including the Output and Command windows. The Output window is the console of the application. The compiler’s messages, for example, are displayed in the Output window. The Command window allows you to enter and execute statements. When you debug an application, you can stop it and enter VB statements in the Command window.

**Project Menu**

This menu contains commands for adding items to the current project (an item can be a form, a file, a component, or even another project). The last option in this menu is the Project Properties command, which opens the project’s Property Pages. The Add Reference and Add Web Reference commands allow you to add references to .NET (or COM) components and Web components, respectively.

**Build Menu**

The Build menu contains commands for building (compiling) your project. The two basic commands in this menu are the Build and Rebuild All commands. The Build command compiles (builds the executable) of the entire solution, but it doesn’t compile any components of the project that haven’t changed since the last build. The Rebuild All command does the same, but it clears any existing files and builds the solution from scratch.
Debug Menu
This menu contains commands to start or end an application, as well as the basic debugging tools. The basic commands of this menu are discussed briefly later in this chapter. They’re also discussed in detail in Chapter 31.

Data Menu
This menu contains commands you will use with projects that access data. You’ll see how to use this short menu’s commands in the discussion of the visual database tools in Part 5 of the book.

Format Menu
The Format menu, which is visible only while you design a Windows or Web form, contains commands for aligning the controls on the form. The commands of this menu will be discussed later in this chapter. The Format menu is invisible when you work in the code editor—its commands apply to the visible elements of the interface.

Tools Menu
This menu contains a list of tools, and most of them apply to C++. The Macros command of the Tools menu leads to a submenu with commands for creating macros. Just as you can create macros in an Office application to simplify many tasks, you can create macros to automate many of the repetitive tasks you perform in the IDE. The last command in this menu, the Options command, leads to the Options dialog box, in which you can fully customize the environment.

Window Menu
This is the typical Window menu of any Windows application. In addition to the list of open windows, it also contains the Hide command, which hides all toolboxes, and the entire window of the IDE is devoted to the code editor, or the form designer. The toolboxes don’t disappear completely; they’re all retracted, and you can see their tabs on the left and right edges of the IDE window. To expand a toolbox, just hover the mouse pointer over the corresponding tab.

Help Menu
This menu contains the various help options. The Dynamic Help command opens the Dynamic Help window, which is populated with topics that apply to the current operation. The Index command opens the Index window, in which you can enter a topic and get help of the specific topic.

Toolbox Window
Here you will find all the controls you can use to build your application’s interface. The Toolbox window is usually retracted, and you must move the pointer over it to view the Toolbox. The controls on the ToolBox are organized in various tabs, so take a look at the controls in the various tabs to become familiar with the controls and their functions. In the first few chapters, we’ll work with the controls in Common Controls and the Menus & Toolbars tabs. The Common Controls tab contains the icons of the most common Windows controls. The Data tab contains the icons of the objects you will use to build data-driven applications (they’re explored in Part 5 of the book).
Solution Explorer Window
This window contains a list of the items in the current solution. A solution can contain multiple projects, and each project can contain multiple items. The Solution Explorer displays a hierarchical list of all the components, organized by project. You can right-click any component of the project and select Properties in the context menu to see the selected component’s properties in the Properties window. If you select a project, you will see the Project Properties dialog box. You will find more information on project properties in the following chapter.

If the solution contains multiple projects, you can right-click the project you want to become the startup form and select Set As StartUp Project. You can also add items to a project with the Add Item command of the context menu or remove a component from the project with the Exclude From Project command. This command removes the selected component from the project, but doesn’t affect the component’s file on the disk. The Delete command removes the selected component from the project and also deletes the component’s file from the disk.

Properties Window
This window (also known as the Property Browser) displays all the properties of the selected component and its settings. Every time you place a control on a form, you switch to this window to adjust the appearance of the control, and you have already seen how to manipulate the properties of a control through the Properties window.

Many properties are set to a single value, like a number or a string. If the possible settings of a property are relatively few, they’re displayed as meaningful constants in a drop-down list. Other properties are set through a more elaborate interface. Color properties, for example, are set from within a Color dialog box that’s displayed right in the Properties window. Font properties are set through the usual Font dialog box. Collections are set in a Collection Editor dialog box, in which you can enter one string for each item of the collection.

If the Properties window is hidden, or if you have closed it, you can either select the View ➤ Properties Window command or right-click a control on the form and select Properties. Or you can simply press F4 to bring up this window. There will be times when a control might totally overlap another control, and you won’t be able to select the hidden control and view its properties. In this case, you can select the desired control in the ComboBox at the top of the Properties window. This box contains the names of all the controls on the form, and you can select a control on the form by selecting its name on this box.

Output Window
The Output window is where many of the tools, including the compiler, send their output. Every time you start an application, a series of messages is displayed on the Output window. These messages are generated by the compiler, and you need not understand them at this point. If the Output window is not visible, select the View ➤ Other Windows ➤ Output command from the menu.

Command and Immediate Windows
While testing a program, you can interrupt its execution by inserting a so-called breakpoint. When the breakpoint is reached, the program’s execution is suspended, and you can execute a statement in the Immediate window. Any statement that can appear in your VB code can also be executed in the Immediate Window. To evaluate an expression, enter a question mark followed
by the expression you want to evaluate, as in the following samples where result is a variable in
the program you interrupted:

```vba
? Math.Log(35)
? "The answer is " & result.ToString
```

You can also send output to this window from within your code with the `Debug.Write` and
`Debug.WriteLine` methods. Actually, this is a widely used debugging technique—to print the
values of certain variables before entering a problematic area of the code. As you will learn in Chapter 31,
there are more elaborate tools to help you debug your application, but printing a few values to the
Immediate window is a time-honored practice in programming with VB.

In many of the examples of this book, especially in the first few chapters, I use the `Debug.WriteLine`
statement to print something to the Immediate window. To demonstrate the use of the `DateDiff()` func-
tion, for example, I’ll use a statement like the following:

```vba
Debug.WriteLine(DateDiff(DateInterval.Day, #3/9/2005#, #5/15/2006#))
```

When this statement is executed, the value 432 will appear in the Immediate Window. This state-
ment demonstrates the syntax of the `DateDiff()` function, which returns the difference between the
two dates in days. Sending some output to the Immediate window to test a function or display the
results of intermediate calculations is a very common practice.

The Command window allows you to execute any menu command from a command line. If you
enter the string “Edit” followed by a period, you will see on a list the commands of the Edit menu
(including the commands that are not visible at the time) and you can invoke any of these commands
and pass arguments to it. If you enter `Edit.Find "Margin"` in the Command window and then press
Enter, the first instance of the string “Margin” will be located in the open code window. The Command
window is not available at design time, but you can access the various commands directly. It becomes
available at runtime and we use it when the application’s execution breaks.

**TaskList Window**

This window is usually populated by the compiler with error messages, if the code can’t be success-
fully compiled. You can double-click an error message in this window, and the IDE will take you to
the line with the statement in error—which you should fix.

You can also add your own tasks to this window. Just click the first empty line and start typing.
A task can be anything from comments and reminders to URLs of interesting sites. You add tasks to
the list and you’re responsible for removing them. Tasks that correspond to errors are removed auto-
matically as soon as you fix the statement that caused them.

**Environment Options**

The Visual Studio IDE is highly customizable. I will not discuss all the customization options here,
but I will show you how to change the default settings of the IDE. Open the Tools menu and select
Options (the last item in the menu). The Options dialog box will appear, in which you can set all the
options regarding the environment. Figure 1.14 shows the options for the font of the various items of
the IDE. Here you can set the font for various categories of items, such as the Text Editor, dialog boxes,
toolboxes, and so on. Initially, you won’t see all the available options, but if you click the “Show All
Settings” box in the dialog box you will see a list of all the items you can customize. Select an item in
the tree of the left pane list and then set the font for this item in the box below.
Figure 1.15 shows the Projects and Solutions options. The top box is the default location for new projects. The three radio buttons in the lower half of the dialog box determine when the changes to the project are saved. By default, changes are saved when you run a project. If you activate the last option, then you must save your project from time to time with the File > Save All command.

Most of the tabs in the Options dialog box are straightforward, and you should take a look at them. If you don’t like some of the default aspects of the IDE, this is the place to change them. Notice the option “Save New Projects When Created,” which is off by default. If you check this option, every time you create a new project the New Project dialog box will prompt you not only for the project’s name but also for the location of the new project. Another item in the Options dialog box worth mentioning is the “Before Building” item of the Build and Run tab. By default, Visual Studio saves the changes made to the project before compiling it. If you want to be in control as to when project changes are stored to disk on top of the previous version of the project, set it to “Don’t Save Changes To Open Documents.” If you switch to the Basic item under the Text Editor branch of the tree in the left pane of the Options dialog box, you will find the “Line Numbers option.” Check this box to display numbers in front of each line in the code window. The Options dialog box contains a lot of options for customizing your work environment and it’s worth exploring on your own.

**Figure 1.14**
The Fonts and Colors options

**Figure 1.15**
The Projects and Solutions options
Building a Console Application

Apart from Windows applications, you can use the Visual Studio 2005 to build applications that run in a Command Prompt window. The Command Prompt window isn’t really a DOS window, even though it looks like one. It’s a text window, and the only way to interact with an application is to enter lines of text and read the output generated by the application, which is displayed on this text window, one line at a time. This type of application is called a Console application, and I’m going to demonstrate Console applications with a single example. We will not return to this type of application later in the book because it’s not what you’re supposed to do as a Windows developer.

The Console application you’ll build in this section, the ConsoleApplication1, prompts the user to enter the name of his or her favorite language and then it prints the appropriate message on a new line, as shown in Figure 1.16.

![Figure 1.16](image)
A Console application uses the Command Prompt window to interact with the user.

Start a new project. In the New Project dialog box, select the template Console Application. You can also change its default name from ConsoleApplication1 to a more descriptive name. For this example, don’t change the application’s name.

A Console application doesn’t have a user interface, so the first thing you’ll see is the code editor’s window with the following statements:

```vbnet
Module Module1

Sub Main()

End Sub

End Module
```

Unlike a Windows application, which is a class, a Console application is a module. Main() is the name of a subroutine that’s executed automatically when you run a Console application. The code you want to execute must be placed between the statements Sub Main() and End Sub. Insert the statements shown in Listing 1.3 in the application’s Main() subroutine.

Listing 1.3: Console Application

```vbnet
Module Module1

Sub Main()
    Console.WriteLine("Enter your favorite language")
    Dim language As String
    language = Console.ReadLine()

End Sub

End Module
```
This code is quite similar to the code of the equivalent Windows applications we developed earlier, except that it uses the `Console.WriteLine` statement to send its output to the Command Prompt window instead of a message box.

A Console application doesn’t react to events because it has no visible interface. However, it’s easy to add elements of the Windows interface to a Console application. If you change the `Console.WriteLine` method call into the `MsgBox()` function, the message will be displayed in a message box.

The reason to build a Console application is to test a specific feature of the language without having to build a user interface. Many of the examples in the documentation are Console applications; they demonstrate the topic at hand and nothing more. If you want to test the `DateDiff()` function, for example, you can create a new Console application and enter the lines of Listing 1.4 in its `Main()` subroutine.

```
Sub Main()
    Console.WriteLine(DateDiff(DateInterval.Day, #3/9/2000#, #5/15/2004#))
    Console.WriteLine("PRESS ENTER TO EXIT")
    Console.ReadLine()
End Sub
```

The last two lines will be the same in every Console application you write. Without them, the Command Prompt window will close as soon as the End Sub statement is reached, and you won’t have a chance to see the result. The `Console.ReadLine` method waits until the user presses the Enter key.

Console applications are convenient for testing short code segments, but Windows programming is synonymous to designing functional user interfaces, and you won’t find any more Console applications in this book.

The sample applications in this chapter demonstrate very basic programming techniques, such as building user interfaces, event programming, validating user input, and handling errors. The goal is to show you how to write simple applications using the most basic elements of the language. This chapter will explain the methodology for building applications. Although the code of the applications will be rather simple, it will demonstrate the basics of validating data and trapping errors.
If you’re a beginner, you might be thinking, “All I want now is to write a simple application that works—I’ll worry about data validation later.” It’s never too early to start thinking about validating your code’s data and error trapping. As you’ll see, making sure that your application doesn’t crash might require more code than the actual operations it performs! If this isn’t quite what you expected, welcome to the club. A well-behaved application must catch and handle every error gracefully, including user as well as programming (your own!) errors.

Using Code Snippets and the My Object

Visual Basic 2005 comes with a lot of predefined code snippets for selected actions, and you can insert these snippets in your code as needed. Let’s say you want to insert the statements for writing some text to a file, but you have no idea how to access files. Create an empty line in the listing (press the Enter key a couple of times at the end of a code line). Then open the Edit menu and select IntelliSense > Insert Snippet (or right-click somewhere on the code window and select Insert Snippet from the context menu).

You will see on the screen a list of the snippets, organized in folders according to their function. Select the File System folder, as shown in Figure 1.17, and double-click it to see a list of common file-related tasks as shown in Figure 1.18. Locate the item “Write Text To Files” in the list and double-click it to insert the appropriate snippet at the current location in the code window.

The following snippet will be inserted in your code:

```
Try
    Dim filePath As String
    filePath = System.IO.Path.Combine( _
        "test.txt")
    My.Computer.FileSystem.WriteAllText(filePath, "some text", True)
Catch fileException As Exception
    Throw fileException
End Try
```

Note that the 3rd statement in the listing is quite long and I had to break it into three lines to fit on the page. To break a long statement into multiple lines append a space and an underscore character where you want to the statement to break. The Try statement is an error-handling mechanism, which we’ll discuss shortly. The program will attempt to execute the statements following the Try statement (the code between the Try and Catch statements). If their execution fails for any reason, the statement following the Catch statement will be executed and it will throw an exception. If not, the program will continue. The code for writing text to a file is quite short. First, the `filePath` variable is set to the path of the file `test.txt` in the MyDocuments folder. The following statement writes the string “some text” to the file indicated by the `filePath` argument. You can replace the string shown in the snippet with any other string, or with the name of the variable (the variable’s value will be written to the file). The last argument of the `WriteAllText` method determines whether the text will be appended to the file (if False) or whether it will overwrite any existing text (if True).

The snippet shows you the basic statements for performing a common task, and you can edit the code inserted by Visual Studio as needed. If you delete the last few characters of the string MyDocuments in the code, you will see a list of all special directories and you can select another one (such as Desktop, My Pictures, and so on). You can also delete the entire expression `My.Computer.FileSystem.SpecialDirectories.MyDocuments) and replace it with an absolute path name like `C:\Temp Files\MyTestFile.txt`. |
You will soon notice that the code snippets of Visual Studio make use of something called My, which is a peculiar object that was introduced with VB 2005 to simplify many programming tasks. As you saw in the preceding code snippet, the My object allows you to write some text to a file with a single statement. If you’re familiar with earlier versions of Visual Basic, you know that you must first open a file, and then write some text to it, and finally close the file. The My object performs all these operations for you and it basically simplifies the language. The My object is a speed dial into...
the features of the language and the operating system and it hides much of the complexity of the language from beginners. For example, you can use the following statement to play back a WAV file from within your code:

My.Computer.Audio.Play('C:\Sounds\CountDown.wav')

or the following expression to play back a system sound:


You can explore the My object on your own and use it as needed. My is not a substitute for learning the language and the Framework. It can help you initially, but you can’t go far without learning the methods of the Framework for handling files or any other feature. Let’s say you want to locate all the files of a specific type in a folder, including its subfolders. Scanning a folder and its subfolders to any depth is quite a task (and you’ll find the actual code in Chapter 13 later in this book). You can do the same with a single statement using the My object:

Dim files As ReadOnlyCollection(Of String)
files = My.Computer.FileSystem.GetFiles('D:\Dir', True, "*.txt")

The GetFiles method populates the files collection with the pathnames of the text files in the folder D:\Dir and its subfolders. However, it won’t help you if you want to process each file in place. Moreover, this GetFiles method is synchronous: if the folder contains many subfolders with many files, it will block the interface until it retrieves all the files. In Chapter 13 you’ll see the code that retrieves file names and adds them to a control as it goes along.

Building a Loan Calculator

One easy-to-implement, practical application is a program that calculates loan parameters. Visual Basic provides built-in functions for performing many types of financial calculations, and you need only a single line of code to calculate the monthly payment given the loan amount, its duration, and the interest rate. Designing the user interface, however, takes much more effort.

Regardless of the language you use, you must go through the following process to develop an application:

1. Decide what the application will do and how it will interact with the user.
2. Design the application’s user interface according to the requirements of step 1.
3. Write the actual code behind the events you want to handle.

How the Loan Calculator Application Works

Following the first step of the process outlined previously, you decide that the user should be able to specify the amount of the loan, the interest rate, and the duration of the loan in months. You must, therefore, provide three text boxes in which the user can enter these values.

Another parameter affecting the monthly payment is whether payments are made at the beginning or at the end of each month, so you must also provide a way for the user to specify whether the payments will be early (first day of the month) or late (last day of the month). The most appropriate type of control for entering Yes/No or True/False type of information is the CheckBox control. This control is a toggle: If it’s checked, you can clear it by clicking it; if it’s cleared, you can check it by
clicking again. The user doesn’t enter any data in this control (which means you need not anticipate user errors with this control), and it’s the simplest method for specifying values with two possible states. Figure 1.19 shows a user interface that matches our design specifications. This is the main form of the LoanCalculator project, which you will find in this chapter’s folder on the book’s project download site.

Figure 1.19
LoanCalculator is a simple financial application.

The user enters all the information on the form and then clicks the Show Payment button to calculate the monthly payment. The program will calculate the monthly payment and display it in the lower TextBox control. All the action takes place in the button’s Click subroutine. The function for calculating monthly payments is called \( Pmt() \) and must be called as follows:

\[
\text{MonthlyPayment} = Pmt(\text{InterestRate}, \text{Periods}, \text{Amount}, \text{FutureValue}, \text{Due})
\]

The interest rate (argument \( \text{InterestRate} \)) is specified as a monthly rate. If the yearly interest rate is 16.5%, the value entered by the user in the Interest Rate box should be 16.5, and the monthly rate will be \( 0.165 / 12 \). The duration of the loan (argument \( \text{Periods} \)) is specified in number of months, and the \( \text{Amount} \) argument is the loan’s amount. The FutureValue of a loan is zero (it would be a positive value for an investment), and the last argument, \( \text{Due} \), specifies when payments are due. The value of \( \text{Due} \) can be one of the constants \( \text{DueDate.BegOfPeriod} \) and \( \text{DueDate.EndOfPeriod} \). These two constants are built into the language, and you can use them without knowing their exact value. In effect, this is the essence of using named constants: You type a self-descriptive name and leave it to VB to convert it to a numeric value.

The present value of the loan is the amount of the loan with a negative sign. It’s negative because you don’t have the money now. You’re borrowing it—it is money you owe to the bank. Future value represents the value of something at a stated time—in this case, what the loan will be worth when it’s paid off. This is what one side owes the other at the end of the specified period. So the future value of a loan is zero.

\( Pmt() \) is a built-in function that uses the five values in the parentheses to calculate the monthly payment. The values passed to the function are called \( \text{arguments} \), which are the values needed by a function (or subroutine) to carry out an action or calculation. By passing different values to the function, the user can specify the parameters of any loan and calculate its monthly payment.

You don’t need to know how the \( Pmt() \) function calculates the monthly payment. The \( Pmt() \) function does the calculations and returns the result. To calculate the monthly payment on a loan of $25,000 with an interest rate of 14.5%, payable over 48 months, and due the last day of the payment period (which in our case is a month), you’d call the \( Pmt() \) function as follows:

\[
\text{Debug.WriteLine}(Pmt(0.145 / 12, 48, -25000, 0, \text{DueDate.EndOfPeriod}))
\]
The value 689.448821287218 will be displayed in the Output window (you’ll see later how you can limit the digits after the decimal point to two because this is all the accuracy you need for dollar amounts). Notice the negative sign in front of the Amount argument in the statement. If you specify a positive amount, the result will be a negative payment. The payment and the loan’s amount have different signs because they represent different cash flows. The loan’s amount is money you owe to the bank, whereas the payment is money you pay to the bank.

The last two arguments of the Pmt() function are optional. If you omit them, Visual Basic uses their default values, which are 0 for the FutureValue argument and DueDate.BegOfPeriod for the Due argument. You can entirely omit these arguments and call the Pmt() function like this:

```csharp
Console.WriteLine(Pmt(0.145 / 12, 48, -25000))
```

Calculating the amount of the monthly payment given the loan parameters is quite simple. What you need to understand are the parameters of a loan and how to pass them to the Pmt() function. You must also know how the interest rate is specified to avoid invalid values. What you don’t need to know is how the payment is calculated—Visual Basic does it for you. This is the essence of functions: they are “black boxes” that perform complicated calculations on their arguments and return the result. You don’t have to know how they work; just how to supply the values required for the calculations.

**Designing the User Interface**

Now that you know how to calculate the monthly payment, you can design the user interface. To do so, start a new project, name it **LoanCalculator**, and rename its form to **LoanForm**.

Your first task is to decide the font and size of the text you’ll use for most controls on the form. Although we aren’t going to display anything on the form directly, all the controls we place on it will have, by default, the same font as the form. The form is the container of the controls, and they inherit some of the form’s properties, such as the font. You can change the font later during the design, but it’s a good idea to start with the right font. At any rate, don’t try to align the controls if you’re planning to change their fonts. This will, most likely, throw off your alignment efforts.

**TIP** Try not to mix fonts on a form. A form, or a printed page for that matter, that includes type in several fonts looks as if it has been created haphazardly and is difficult to read. However, you can use different sizes for some of the controls on the form.

The loan application you’ll find on the Web site uses the 10-point Verdana font. To change it, select the form with the mouse, double-click the name of the Font property in the Properties window to open the Font dialog box, and select the desired font and attributes.

To design the form shown previously in Figure 1.19, follow these steps:

1. Place four labels on the form and assign the following captions (the Text property of each control) to them:

<table>
<thead>
<tr>
<th>Name</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label1</td>
<td>Amount</td>
</tr>
<tr>
<td>Label2</td>
<td>Duration</td>
</tr>
<tr>
<td>Label3</td>
<td>Interest Rate</td>
</tr>
<tr>
<td>Label4</td>
<td>Monthly Payment</td>
</tr>
</tbody>
</table>
The labels should be large enough to fit their captions. You don’t need to change the default names of the four Label controls on the form because their captions are all we need. You aren’t going to program them.

2. Place a TextBox control next to each label. Set their Name and Text properties to the following values. I used meaningful names for the TextBox controls because we’ll use them in our code shortly to retrieve the values entered by the user on these controls. These initial values correspond to a loan of $25,000 with an interest rate of 14.5% and a payoff period of 48 months.

<table>
<thead>
<tr>
<th>Name</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>txtAmount</td>
<td>25,000</td>
</tr>
<tr>
<td>txtDuration</td>
<td>48</td>
</tr>
<tr>
<td>txtRate</td>
<td>14.5</td>
</tr>
<tr>
<td>txtPayment</td>
<td></td>
</tr>
</tbody>
</table>

3. The fourth TextBox control is where the monthly payment will appear. The user isn’t supposed to enter any data in this box, so you must set its ReadOnly property to True. You’ll be able to change its value from within your code, but users won’t be able to type anything in it. (We could have used a Label control instead, but the uniform look of TextBoxes on a form is usually preferred.)

4. Next, place a CheckBox control on the form. By default, the control’s caption is Check1, and it appears to the right of the check box. Because we want the titles to be to the left of the corresponding controls, we’ll change this default appearance.

5. Select the check box with the mouse (if it’s not already selected), and in the Properties window, locate the CheckAlign property. Its value is MiddleLeft. If you expand the drop-down list by clicking the Arrow button, you’ll see that this property has many different settings, and each setting is shown as a square. Select the button that will center the text vertically and right-align it horizontally. The string MiddleRight will appear in the property’s box when you click the appropriate button. The first component of the CheckAlign property’s value indicates the vertical alignment of the check box, and the second component of the value indicates the horizontal alignment. MiddleRight means that the check box should be centered vertically and right-aligned horizontally.
6. With the check box selected, locate the Name property in the Properties window, and set it to `chkPayEarly`.

7. Change the CheckBox’s caption by entering the string `Early Payment` in its Text property field.

8. Place a Button control in the bottom-left corner of the form. Name it `bttnShowPayment`, and set its caption to `Show Payment`.

9. Finally, place another Button control on the form, name it `bttnExit`, and set its Text property to `Exit`.

ALIGNING THE CONTROLS
Your next step is to align the controls on the form. First, be sure that the captions on the labels are visible. Our labels contain lengthy captions, and if you don’t make the labels long enough, the captions may wrap to a second line and become invisible.

TIP Be sure to make your labels long enough to hold their captions, especially if you’re using a non-standard font. A user’s computer might substitute another font for your nonstandard font, and the corresponding captions might increase in length.

The IDE provides commands to align the controls on the form, all of which can be accessed through the Format menu. To align the controls that are already on the LoanForm, follow these steps:

1. Select the four labels on the form with the mouse and left-align them by choosing Format ➤ Align ➤ Left. The handles of all selected controls will be black, except for one control whose handles will be white. To specify the control that will be used as a reference for aligning the other controls, click it after making the selection. (You can select multiple controls either by drawing a rectangle that encloses them with the mouse, or by clicking each control while holding down the Ctrl button.)

2. With the four text boxes selected, choose Format ➤ Align ➤ Left to left-align them. Don’t include the check box in this selection.

TIP When you select multiple controls to align together, use the control with white handles as a guide for aligning the other controls.

3. With all four text boxes still selected, use the mouse to align them above and below the box of the CheckBox control.

Your form should now look like the one shown in Figure 1.19. Take a good look at it and check to see whether any of your controls are misaligned. In the interface design process, you tend to overlook small problems such as a slightly misaligned control. The user of the application, however, instantly spots such mistakes. It doesn’t make any difference how nicely the rest of the controls are arranged on the form; if one of them is misaligned, it will attract the user’s attention.

Programming the Loan Application
Now run the application and see how it behaves. Enter a few values in the text boxes, change the state of the check box, and test the functionality already built into the application. Clicking the Show Payment button won’t have any effect because we have not yet added any code. If you’re happy
with the user interface, stop the application, open the form, and double-click the Show Payment
Button control. Visual Basic opens the code window and displays the definition of the ShowPayment-
Click event:

Private Sub bttnShowPayment_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles bttnShowPayment.Click
End Sub

**NOTE** I’ve broken the first line with an underline character because it wouldn’t fit on the page. The
underscore character is the line-continuation character, which allows you to break a long code line
into multiple text lines.

This is the declaration of the Button’s Click event handler. This subroutine will be invoked when
the user clicks the Show Payment button. Above the definition of the event handler, you will see the
following statement:

Public Class LoanForm

This statement creates a new class for the project’s form and it was placed there by the IDE, and
you shouldn’t change them. When you learn more about classes and inheritance in the second part of
the book, you’ll be able to better understand the role of these statements.

Place the pointer between the lines Private Sub and End Sub, and enter the rest of the lines of
Listing 1.5 (you don’t have to re-enter the first and last lines that declare the event handler).

**LISTING 1.5: Show Payment Button**

Private Sub bttnShowPayment_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles bttnShowPayment.Click
Dim Payment As Double
Dim LoanIRate As Double
Dim LoanDuration As Integer
Dim LoanAmount As Integer

LoanAmount = Convert.ToInt32(txtAmount.Text)
LoanIRate = 0.01 * Convert.ToDecimal(txtRate.Text) / 12
LoanDuration = Convert.ToInt32(txtDuration.Text)
Dim payEarly As DueDate
If chkPayEarly.Checked Then
    payEarly = DueDate.BegOfPeriod
Else
    payEarly = DueDate.EndOfPeriod
End If
Payment = Pmt(LoanIRate, LoanDuration, -LoanAmount, 0, payEarly)
txtPayment.Text = Payment.ToString("#.00")
End Sub
The code window should now look like the one shown in Figure 1.20. Notice the underscore character at the end of the first part of the long line. The underscore lets you break long lines so that they will fit nicely in the code window. I’m using this convention in this book a lot to fit long lines on the printed page. The same statement you see as multiple lines in the book may appear in a single, long line in the project.

You don’t have to break long lines manually as you enter code in the editor’s window. Open the Edit menu and select Advanced ➤ Word Wrap. The editor will wrap long lines automatically at a word boundary. While the word wrap feature is on, a check mark appears in front of the Edit ➤ Advanced ➤ Word Wrap command. To turn off word wrapping, select the same command again.

In Listing 1.5, the first line of code within the subroutine declares a variable. It lets the application know that Payment is a placeholder for storing a floating-point number (a number with a decimal part)—the Single data type. The second line declares a variable of the DueDate type. This is the type of the argument that determines whether the payment takes place at the beginning or the end of the month. The last argument of the Pmt() function must be a variable of this type, so we declare a variable of the DueDate type. As mentioned earlier in this chapter, DueDate is an enumeration with two members: BegOfPeriod and EndOfPeriod. In short, the last argument of the Pmt() function can be one of the following values:

DueDate.BegOfPeriod
DueDate.EndOfPeriod

The first really executable line in the subroutine is the If statement that examines the value of the chkPayEarly CheckBox control. If the control is checked, the code sets the payEarly variable to DueDate.BegOfPeriod. If not, the code sets the same variable to DueDate.EndOfPeriod. The ComboBox control’s Checked property returns True if the control is checked at the time; False otherwise. After setting the value of the payEarly variable, the code calls the Pmt() function, passing the values of the controls as arguments:

- The first argument is the interest rate. The value entered by the user in the txtRate TextBox is multiplied by 0.01 so that the value 14.5 (which corresponds to 14.5%) is passed to the Pmt() function as 0.145. Although we humans prefer to specify interest rates as integers (8%) or floating-point numbers larger than 1 (8.24%), the Pmt() function expects to read a number
less than 1. The value 1 corresponds to 100%. Therefore, the value 0.1 corresponds to 10%.
This value is also divided by 12 to yield the monthly interest rate.

◆ The second argument is the duration of the loan in months (the value entered in the
\texttt{txtDuration} TextBox).

◆ The third argument is the loan's amount (the value entered in the \texttt{txtAmount} TextBox).

◆ The fourth argument (the loan’s future value) is 0 by definition.

◆ The last argument is the \texttt{payEarly} variable, which is set according to the status of the \texttt{chkPayEarly}
control.

The following two statements convert the numeric value returned by the \texttt{Pmt()} function to a string
and display this string in the fourth TextBox control. The result is formatted appropriately with the
following expression:

\[ \text{Payment}.\text{ToString}('#.00') \]

The \texttt{Payment} variable is numeric, and all numeric variables provide the method \texttt{ToString}, which
formats the numeric value and converts it to a string. The character \# stands for the integer part of the
variable. The period separates the integer from the fractional part, which is rounded to two decimal
digits. The \texttt{Pmt()} function returns a precise number, such as 372.2235687646345, and you must round
and format it nicely before displaying it. Because the bank can’t charge you anything less than a
penny, you don’t need extreme accuracy. Two fractional digits are sufficient. For more information
on formatting numeric (and other) values, see the section “Formatting Numbers” in Chapter 2.

### A Code Snippet for Calculating Monthly Loan Payments

If you didn’t know about the \texttt{Pmt} built-in function, how would you go about calculating loan payments?
Code snippets to the rescue! Right-click somewhere in the code window and from the context menu select
the Insert Snippet command. Double-click the Math folder and then select the snippet “Calculate a
monthly payment on a loan.” The following code will be inserted at the location of the pointer (I inserted
a few line breaks to fit the long statements on the printed page):

\begin{verbatim}
loanAmount = CDbl(InputBox('How much do you want to borrow?'))
annualPercentRate = CDbl(InputBox( 
    'What is the annual percentage rate of your loan? Enter 8% as .08.'))
totalPayments = CDbl(InputBox(_
    'How many monthly payments will you make?'))

payment = Pmt(annualPercentRate / 12, totalPayments, -loanAmount, 
    futureValue, DueDate.EndOfPeriod)

MsgBox('Your payment will be ' & payment.ToString('C') & _
    ' per month.', MsgBoxStyle.OKOnly, 'Payments')
\end{verbatim}

The snippet demonstrates the use of the \texttt{Pmt()} function. It uses the \texttt{InputBox()} function to prompt the user
for the loan's parameters, but you can get an idea of the \texttt{Pmt()} function's arguments and edit the code to
replace the values of the various parameters with the data from the appropriate controls on the form.
To display the result returned by the Pmt() function on the txtPayment TextBox control, use the following statement:

```vbnet
txtPayment.Text = Pmt(0.01 * txtRate.Text / 12, txtDuration.Text, _
-txtAmount.Text, 0, payEarly)
```

This statement assigns the value returned by the Pmt() function directly to the Text property of the control. The monthly payment will be displayed with four decimal digits, but this isn’t a proper dollar amount.

**TIP** You almost always use the ToString method (or the Format() function) when you want to display the results of numeric calculations because most of the time you don’t need Visual Basic’s extreme accuracy. A few fractional digits are all you need. In addition to numbers, the ToString method can format dates and time. The ToString method’s formatting capabilities for the various data types are discussed in Chapter 12, and the Format() function is a VB built-in function.

The code of the LoanCalculator project on the Web site is different and considerably longer than what I have presented here. The statements discussed in the preceding text are the bare minimum for calculating a loan payment. The user can enter any values on the form and cause the program to crash. In the next section, we’ll see how you can validate the data entered by the user, catch errors, and handle them gracefully (that is, give the user a chance to correct the data and proceed), as opposed to terminating the application with a runtime error.

### Validating the Data

If you enter a nonnumeric value in one of the fields, the program will crash and display an error message. For example, if you enter *twenty* in the Duration text box, the program will display the error message shown in Figure 1.21. A simple typing error can crash the program. This isn’t the way Windows applications should work. Your applications must be able to handle all kinds of user errors, provide helpful messages, and in general, guide the user in running the application efficiently. If a user error goes unnoticed, your application will either end abruptly or will produce incorrect results without an indication.

**Figure 1.21**
The Cast Exception message means that you supplied a string where a numeric value was expected.
Click the Break button, and Visual Basic will take you back to the application’s code window, in which the statements that caused the error will be highlighted in green. Obviously, we must do something about user errors. One way to take care of typing errors is to examine each control’s contents; if they don’t contain valid numeric values, display your own descriptive message and give the user another chance. Listing 1.6 is the revised Click event handler that examines the value of each text box before attempting to use it in any calculations.

**Listing 1.6: Revised Show Payment Button**

```vbnet
Private Sub btnShowPayment_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnShowPayment.Click
    Dim Payment As Double
    Dim LoanIRate As Double
    Dim LoanDuration As Integer
    Dim LoanAmount As Integer

    ' Validate amount
    If IsNumeric(txtAmount.Text) Then
        LoanAmount = Convert.ToInt32(txtAmount.Text)
    Else
        MsgBox("Please enter a valid amount")
        Exit Sub
    End If

    ' Validate interest rate
    If IsNumeric(txtRate.Text) Then
        LoanIRate = 0.01 * Convert.ToDouble(txtRate.Text) / 12
    Else
        MsgBox("Invalid interest rate, please re-enter")
        Exit Sub
    End If

    ' Validate loan's duration
    If IsNumeric(txtDuration.Text) Then
        LoanDuration = Convert.ToInt32(txtDuration.Text)
    Else
        MsgBox("Please specify the loan's duration as a number of months")
        Exit Sub
    End If

    ' If all data were validated, proceed with calculations
    Dim payEarly As DueDate
    If chkPayEarly.Checked Then
        payEarly = DueDate.BegOfPeriod
    Else
        payEarly = DueDate.EndOfPeriod
    End If
    Payment = Pmt(LoanIRate, LoanDuration, -LoanAmount, 0, payEarly)
    txtPayment.Text = Payment.ToString("#.00")
End Sub
```
First, we declare three variables in which the loan’s parameters will be stored: LoanAmount, LoanIRate, and LoanDuration. These values will be passed to the Pmt() function as arguments. Each text box’s value is examined with an If structure. If the corresponding text box holds a valid number, its value is assigned to the numeric variable. If not, the program displays a warning and exits the subroutine without attempting to calculate the monthly payment. Before exiting the subroutine, however, the code moved the focus to the text box with the invalid value because this is the control that the user will most likely edit. After fixing the incorrect value, the user can click the ShowPayment button again. IsNumeric() is another built-in function that accepts a variable and returns True if the variable is a number; False otherwise.

If the Amount text box holds a numeric value, such as 21,000 or 21.50, the function IsNumeric(txtAmount.Text) returns True, and the statement following it is executed. That following statement assigns the value entered in the Amount TextBox to the LoanAmount variable. If not, the Else clause of the statement is executed, which displays a warning in a message box and then exits the subroutine. The Exit Sub statement tells Visual Basic to stop executing the subroutine immediately, as if the End Sub line were encountered.

You can run the revised application and check it out by entering invalid values in the fields. Notice that you can’t specify an invalid value for the last argument; the CheckBox control won’t let you enter a value. You can only check or clear it, and both options are valid. The LoanCalculator application you’ll find on the Web site contains this last version with the error-trapping code.

The actual calculation of the monthly payment takes a single line of Visual Basic code. Displaying it requires another line of code. Adding the code to validate the data entered by the user, however, is an entire program. And that’s the way things are.

Writing Well-Behaved Applications

A well-behaved application must contain data-validation code. If an application such as LoanCalculator crashes because of a typing mistake, nothing really bad will happen. The user will try again or else give up on your application and look for a more professional one. However, if the user has been entering data for hours, the situation is far more serious. It’s your responsibility as a programmer to make sure that only valid data are used by the application and that the application keeps working, no matter how the user misuses or abuses it.

NOTE The applications in this book don’t contain much data-validation code because it would obscure the “useful” code that applies to the topic at hand. Instead, they demonstrate specific techniques. You can use parts of the examples in your applications, but you should provide your own data-validation code (and error-handling code, as you’ll see in the following section).

Now run the application one last time and enter an enormous loan amount. Try to find out what it would take to pay off the national debt with a reasonable interest rate in, say, 72 months. The program will crash again (as if you didn’t know). This time the program will go down with a different error message, as shown in Figure 1.22. Visual Basic will complain about an “overflow.” The exact message is “Value was either too large or too small for an Int32” and the program will stop at the line that assigns the contents of the txtAmount TextBox to the LoanAmount variable. Press the Break button, and the offending statement in the code will be highlighted.
BUILDING A MATH CALCULATOR

TIP

An overflow is a numeric value too large for the program to handle. This error is usually produced when you divide a number by a very small value. When you attempt to assign a very large value to an Integer variable, you’ll also get an overflow exception.

Actually, in the LoanCalculator application, any amount greater than 2,147,483,647 will cause an overflow condition. This is largest value you can assign to an Integer variable; it’s plenty for our banking needs, but not nearly adequate for handling government budgets. As you’ll see in the next chapter, Visual Basic provides other types of variables, which can store enormous values (making the national debt look really small). In the meantime, if you want to use the loan calculator, change the declaration of the LoanAmount variable to the following:

Dim LoanAmount As Double

The Double data type can hold much larger values. Besides, the Double data type can also hold noninteger values. You’ll never apply for a loan of $25,000 and some cents, but if you want to calculate the precise monthly payment for a debt you have accumulated, you should be able to specify a noninteger amount. In short, we should have declared the LoanAmount variable with the Double data type in the first place.

An overflow error can’t be caught with data-validation code. There’s always a chance your calculations will produce overflows or other types of math errors. Data validation won’t help here; you just don’t know the result before you carry out the calculations. We need something called error handling, or exception handling. This is additional code that can handle errors after they occur. In effect, you’re telling VB that it shouldn’t stop with an error message, which would be embarrassing for you and wouldn’t help the user one bit. Instead, VB should detect the error and execute the proper statements that will handle the error. Obviously, you must supply these statements (you’ll see examples of handling errors at runtime shortly).

Building a Math Calculator

Our next application is more advanced, but not as advanced as it looks. It’s a math calculator with a typical visual interface that demonstrates how Visual Basic can simplify the programming of fairly advanced operations. If you haven’t tried it, you may think that writing an application such as this one is way too complicated for a beginner, but it isn’t. The MathCalculator application is shown in Figure 1.23, and you’ll find it in this chapter’s folder on the book’s download site. The application emulates the operation of a hand-held calculator and implements the basic arithmetic operations. It has the structure of a math calculator, and you can easily expand it by adding more features. In fact, adding features such as cosines and logarithms is actually simpler than performing the basic arithmetic operations.
Designing the User Interface

The application's interface is straightforward, but it takes a bit of effort. You must align the buttons on the form and make the calculator look as much like a hand-held calculator as possible. Start a new project, the MathCalculator project, and rename its main form from Form1.vb to CalculatorForm.vb.

Designing the interface of the application isn't trivial because it's made up of many buttons, all perfectly aligned on the form. To simplify the design, follow these steps:

1. Select a font that you like for the form. All the Command buttons you'll place on the form will inherit this font. The MathCalculator sample application uses 10-point Verdana font.

2. Add the Label control, which will become the calculator’s display. Set its BorderStyle property to Fixed 3D so that it will have a 3-D look, as shown in Figure 1.23. Change its ForeColor and BackColor properties too, if you want it to look different from the rest of the form. The sample project uses colors that emulate the—now extinct—green CRT monitors.

3. Draw a Button control on the form, change its caption (Text property) to 1, and name it `bttn1`. Size the button carefully so that its caption is centered on the control. The other buttons on the form will be copies of this one, so make sure you’ve designed the first button as best as you can before you start making copies of it. You can also change the button’s style with the FlatStyle property (you can experiment with the Popup, Standard, and System settings of this property).

4. Place the button in its final position on the form. At this point, you’re ready to create the other buttons for the calculator’s digits. Right-click the button and select Copy from the context menu. The Button control is copied to the Clipboard, and now you can paste it on the form (which is much faster than designing an identical button).

5. Right-click somewhere on the form, select Paste to create a copy of the button you copied earlier and the button copied to the Clipboard will be pasted on the form. The copy will have the same caption as the button it was copied from, and its name will be Button1.

6. Now set the button’s Name to `bttn2` and its Text property to 2. This button is the digit 2. Place the new button to the right of the previous button. You don’t have to align the two buttons perfectly now; we’ll use later the Format menu to align the buttons on the form. As you move the control around on the form, one or more lines may appear at times. These lines are called `snap lines` and they appear as soon as a control is aligned (vertically or horizontally) with one or more of the existing controls on the form. The snap lines allow you to align controls with the mouse. Blue snap lines appear when the control’s edge is aligned with the edge of another control. Red snap lines appear when the control’s baseline is aligned with the baseline of another control. The baseline is the invisible line on which the characters of the control’s caption is based.
7. Repeat steps 5 and 6 eight more times, once for each numeric digit. Each time a new Button control is pasted on the form, Visual Basic names it Button1 and sets its caption to 1; you must change the Name and Text properties. You can name the buttons anything you like, but a name that indicates their role in the application is preferred.

8. When the buttons of the numeric digits are all on the form, place two more buttons, one for the C (Clear) operation and one for the Period button. Name them btnClear and btnPeriod, and set their captions accordingly. Use a larger font size for the Period button to make its caption easier to read.

9. When all the digit buttons of the first group are on the form and in their approximate positions, align them with the commands of the Format menu. You can use the snap lines to align horizontally and vertically the various buttons on the form, but you must still space the controls manually, which isn’t a trivial task. Here’s how you can align the buttons perfectly with the Format menu.

   a. First, align the buttons of the top row. Start by aligning the 1 button with the left side of the lblDisplay Label. Then select all the buttons of the top row and make their horizontal spacing equal (select Format ➤ Horizontal Spacing ➤ Make Equal). Then do the same with the buttons in the first column; this time, make sure that their vertical distances are equal (Format ➤ Vertical Spacing ➤ Make Equal).

   b. Now you can align the buttons in each row and each column separately. Use one of the buttons you aligned in the last step as the guide for the rest of them. The buttons can be aligned in many ways, so don’t worry if somewhere in the process you ruin the alignment. You can always use the Undo command in the Edit menu. Select the three buttons on the second row and align their Tops using the first button as reference. To set the anchor control for the alignment, click it with the mouse while holding down the Control key. Do the same for the third and fourth rows of buttons. Then do the same for the four columns of buttons using the top button as reference.

Now, place the buttons for the arithmetic operations on the form—addition (+), subtraction (–), multiplication (*), and division (/). Finally, place the Equals button on the form and make it wide enough to span the space of two operation buttons. Use the commands on the Format menu to align these buttons, as shown in Figure 1.23. The form shown in Figure 1.23 has a few more buttons, which you can align using the same techniques you used to align the numeric buttons.

If you don’t feel quite comfortable with the alignment tools of the IDE, you can still position the controls on the form through the X and Y components of each control’s Location property (they’re the X and Y coordinates of the control’s upper left corner on the form). The various alignment tools are among the first tools of the IDE you’ll master and you’ll be creating forms with perfectly aligned controls in no time at all.

### Programming the MathCalculator

Now you’re ready to add some code to the application. Double-click one of the digit buttons on the form, and you’ll see the following in the code window:

```vbnet
Private Sub btn1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btn1.Click
End Sub
```
This is the Click event’s handler for a single digit button. Your first attempt is to program the Click event handler of each digit button, but repeating the same code 10 times isn’t very productive (not to mention that if we decide to edit the code later, the process must be repeated 10 times). We’re going to use the same event handler for all buttons that represent digits. All you have to do is append the names of the events to be handled by the same subroutine after the Handles keyword. You should also change the name of the event handler to something that indicates its role. Because this subroutine handles the Click event for all the digit buttons, let’s call it DigitClick(). Here’s the revised declaration of a subroutine that can handle all the digit buttons:

```vbnet
Private Sub DigitClick(ByVal sender As System.Object, ByVal e As System.EventArgs) 
Handles bttn0.Click, bttn1.Click, bttn2.Click, 
    bttn3.Click, bttn4.Click, bttn5.Click, bttn6.Click, 
    bttn7.Click, bttn8.Click, bttn9.Click
End Sub
```

You don’t have to type all the event names; as soon as you insert the first comma after `bttn0.Click`, a drop-down list with the names of the controls will open, and you can select the name of the next button with the arrow down. Press the space bar to select the desired control (`bttn1, bttn2`, and so on) and then type the period. This time, you’ll see another list with the names of the event for the selected control. Locate the Click event and select it by pressing the spacebar. Enter the next comma and repeat the process for all the buttons. This extremely convenient feature of the language is the IntelliSense: the IDE presents the available and valid keywords as you type. If you use IntelliSense consistently, you’ll minimize syntax errors.

When you press a digit button on a hand-held calculator, the corresponding digit is appended to the display. To emulate this behavior, insert the following line in the Click event handler:

```vbnet
lblDisplay.Text = lblDisplay.Text + sender.Text
```

This line appends the digit clicked to the calculator’s display. The sender argument of the Click event represents the control that was clicked (the control that fired the event). The Text property of this control is the caption of the button that was clicked. For example, if you have already entered the value 345, clicking the digit 0 displays the value 3450 on the Label control that acts as the calculator’s display.

The expression `sender.Text` is not the best method of accessing the Text property of the button that was clicked, but it will work as long as the Strict option is off. We’ll return to this topic later in the book, but for now you should convert the `sender` object to a Button object and then access its Text property with the following statement:

```vbnet
CType(sender, Button).Text
```

The CType() function is discussed in the following chapter. For now, keep in mind that it converts an object to an object of a different type. The `sender` object represents the object that fired the event, which is a Button control. You will also notice that after typing the period following the closing parenthesis, IntelliSense will display all the members of the Button control as if you had entered the name of a Button control followed by a period.

The code behind the digit buttons needs a few more lines. After certain actions, the display should be cleared. After pressing one of the buttons that correspond to math operations, the display should be cleared in anticipation of the second operand. Actually, the display must be cleared as soon as the first digit of the second operand is pressed, and not as soon as the math operator button is pressed. Likewise,
the display should also be cleared after the user clicks the Equals button. Revise the Digit_Click event handler, as shown in Listing 1.7.

**Listing 1.7: Digit_Click Event**

```vbnet
Private Sub Digit_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btn1.Click, btn2.Click, btn3.Click, btn4.Click, btn5.Click, btn6.Click, btn7.Click, btn8.Click, btn9.Click
    If clearDisplay Then
        lblDisplay.Text ="
        clearDisplay = False
    End If
    lblDisplay.Text = lblDisplay.Text + sender.text
End Sub
```

The `clearDisplay` variable is declared as Boolean, which means it can take a True or False value. Suppose that the user has performed an operation and the result is on the calculator’s display. The user now starts typing another number. Without the If clause, the program would continue to append digits to the number already on the display. This is not how calculators work. When the user starts entering a new number, the display must be cleared. And our program uses the `clearDisplay` variable to know when to clear the display.

The Equals button sets the `clearDisplay` variable to True to indicate that the display contains the result of an operation. The DigitClick() subroutine examines the value of this variable each time a new digit button is pressed. If the value is True, DigitClick() clears the display and then prints the new digit on it. The subroutine also sets `clearDisplay` to False so that when the next digit is pressed, the program won’t clear the display again.

What if the user makes a mistake and wants to undo an entry? The typical hand-held calculator has no Backspace key. The Clear key erases the current number on the display. Let’s implement this feature. Double-click the C button and enter the code of Listing 1.8 in its Click event.

**Listing 1.8: Clear Button**

```vbnet
Private Sub btnClear_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnClear.Click
    lblDisplay.Text ="
End Sub
```

Now we can look at the Period button. A calculator, no matter how simple, should be able to handle fractional numbers. The Period button works just like the digit buttons, with one exception. A digit can appear any number of times in a numeric value, but the period can appear only once. A number such as 99.991 is valid, but you must make sure that the user can’t enter numbers such as 23.456.55. After a period is entered, this button mustn’t insert another one. The code in Listing 1.9 accounts for this.
Listing 1.9: Period Button

```vbnet
Private Sub btnPeriod_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnPeriod.Click
    If lblDisplay.Text.IndexOf(“.”) >= 0 Then
        Exit Sub
    Else
        lblDisplay.Text = lblDisplay.Text & “.”
    End If
End Sub
```

IndexOf is a method that can be applied to any string. The expression `lblDisplay.Text.IndexOf(“.”)` returns the location of the first instance of the period in the caption of the Label control. If this number is zero or positive, the number entered contains a period already, and another can’t be entered. In this case, the program exits the subroutine. If the method returns –1, the period is appended to the number entered so far, just like a regular digit.

Check out the operation of the application. We have already created a functional user interface that emulates a hand-held calculator with data-entry capabilities. It doesn’t perform any operations yet, but we have already created a functional user interface with only a small number of statements.

**Math Operations**

Now we can move to the interesting part of the application: considering how a calculator works. Let’s start by defining three variables:

- **Operand1** The first number in the operation
- **Operator** The desired operation
- **Operand2** The second number in the operation

When the user clicks one of the math symbols, the value on the display is stored in the variable `Operand1`. If the user then clicks the Plus button, the program must make a note to itself that the current operation is an addition and set the `clearDisplay` variable to True so that the user can enter another value. The symbol of the operation is stored in the `Operator` variable. The user enters another value and then clicks the Equals button to see the result. At this point, our program must do the following:

1. Read the value on the display into the `Operand2` variable.
2. Perform the operation indicated by the `operator` variable with the two operands.
3. Display the result and set the `clearDisplay` variable to True.
The Equals button must perform the following operation:

\[ \text{Operand1 } \text{Operator} \text{Operand2} \]

Suppose that the number on the display when the user clicks the Plus button is 3342. The user then enters the value 23 and clicks the Equals button. The program must carry out the addition:

\[ 3342 + 23 \]

If the user clicked the Division button, the operation is as follows:

\[ 3342 / 23 \]

Variables are local in the subroutines in which they are declared. Other subroutines have no access to them and can’t read or set their values. Sometimes, however, variables must be accessed from many places in a program. If the \text{Operand1}, \text{Operand2}, and \text{Operator} variables in this application must be accessed from within more than one subroutine, they must be declared outside any subroutine. The same is true for the \text{clearDisplay} variable. Their declarations, therefore, must appear outside any procedure, and they usually appear at the beginning of the code with the following statements:

```vbnet
Dim clearDisplay As Boolean
Dim Operand1 As Double
Dim Operand2 As Double
Dim Operator As String
```

We’ll have a lot to say about the scope of variables in the following chapter, but here’s the basic rule about variables: a variable is visible in the module in which it was declared. A variable declared in a subroutine like the DigitClick() subroutine can be accessed by the code in the same subroutine and no other part of the application. If you need a variable that can be accessed from within all the procedures of a form, declare it outside any subroutine (or function). These variables are called \textit{form-wide variables}, or simply \textit{form variables}, because they are visible from within any subroutine on the form.

Let’s see how the program uses the \text{Operator} variable. When the user clicks the Plus button, the program must store the value “+” in the \text{Operator} variable. This takes place from within the Plus button’s Click event. But later, the Equals button must have access to the value of the \text{Operator} variable in order to carry out the operation (in other words, it must know what type of operation the user specified). Because these variables must be manipulated by the code of several subroutines, they were declared outside any subroutine.

All variables that store numeric values are declared as variables of the Double type, which can store values with the greatest possible precision. (Numeric variables and their types are discussed in detail in the next chapter.) The Boolean type takes two values: True and False. You have already seen how the \text{clearDisplay} variable is used.

With the variable declarations out of the way, we can now implement the Operator buttons. Double-click the Plus button and, in the Click event’s handler, enter the lines shown in Listing 1.10.
LISTING 1.10: Plus Button

Private Sub bttnPlus_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles bttnPlus.Click
Operand1 = Convert.ToDouble(lblDisplay.Text)
Operator = "+"
clearDisplay = True
End Sub

The variable Operand1 is assigned the value currently on the display. The Convert.ToDouble() method converts its argument to a double value. The Text property of the Label control is a string. For example, you can assign the value “My Label” to a label’s Text property. The actual value stored in the Text property is not a number. It’s a string such as “428”, which is different from the numeric value 428. That’s why we use the Convert.ToDouble method to convert the value of the Label’s caption to a numeric value. The remaining buttons do the same, and I won’t show their listings here.

So far, we have implemented the following functionality in our application: When an operator button is clicked, the program stores the value on the display in the Operand1 variable and the operator in the Operator variable. It then clears the display so that the user can enter the second operand. After the second operand is entered, the user can click the Equals button to calculate the result. When this happens, the code of Listing 1.11 is executed.

LISTING 1.11: Equals Button

Private Sub bttnEquals_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles bttnEquals.Click
Dim result As Double
Operand2 = Convert.ToDouble(lblDisplay.Text)
Select Case Operator
Case "+"
    result = Operand1 + Operand2
Case "-"
    result = Operand1 - Operand2
Case "*"
    result = Operand1 * Operand2
Case "/"
    If Operand2 <> "0" Then
        result = Operand1 / Operand2
    End Select
End Select
lblDisplay.Text = result.ToString
clearDisplay = True
End Sub
The resultant variable is declared as Double so that the result of the operation will be stored with maximum precision. The code extracts the value displayed in the Label control and stores it in the variable Operand2. It then performs the operation with a Select Case statement. This statement compares the value of the Operator variable to the values listed after each Case statement. If the value of the Operator variable matches one of the Case values, the following statement is executed.

- If the operator is “+”, the resultant variable is set to the sum of the two operands.
- If the operator is “–”, the resultant variable is set to the difference of the first operand minus the second.
- If the operator is “*”, the resultant variable is set to the product of the two operands.
- If the operator is “/”, the resultant variable is set to the quotient of the first operand divided by the second operand, provided that the divisor is not zero.

NOTE: Division takes into consideration the value of the second operand because if it’s zero, the division can’t be carried out. The last statement carries out the division only if the divisor is not zero. If Operand2 happens to be zero, nothing happens.

Now run the application and check it out. It works just like a hand-held calculator, and you can’t crash it by specifying invalid data. We didn’t have to use any data-validation code in this example because the user doesn’t get a chance to type invalid data. The data-entry mechanism is foolproof. The user can enter only numeric values because there are only numeric digits on the calculator. The only possible error is to divide by zero, and that’s handled in the Equals button.

Simple Debugging Tools

Our application works nicely and is quite easy to test and fix if you discover something wrong with it (but only because it’s a very simple application). As you write code, you’ll soon discover that something doesn’t work as expected, and you should be able to find out why and then fix it. The process of eliminating errors is called debugging, and Visual Studio provides the tools to simplify the process of debugging (these tools are discussed in detail in Chapter 31). There are a few simple operations you should know, even as you work with simple projects like this one.

Open the MathCalculator project in the code editor and place the pointer in the line that calculates the difference between the two operands. Let’s pretend there’s a problem with this line, and we want to follow the execution of the program closely to find out what’s going wrong with the application. Press F9, and the line will be highlighted in brown color. This line has become a breakpoint: As soon as it is reached, the program will stop.

Press F5 to run the application and perform a subtraction. Enter a number; then click the minus button, and then another number, and finally the Equals button. The application will stop, and the code editor will open. The breakpoint will be highlighted in yellow color. Hover the pointer of the Operand1 and Operand2 variables in the code editor’s window. The value of the corresponding variable will appear in a small ToolTip box. Move the pointer over any variable in the current event handler to see its value. These are the values of the variables just prior to the execution of the highlighted statement.
The `result` variable is zero because the statement hasn’t been executed yet. If the variables involved in this statement have their proper values (if they don’t, you know that the problem is prior to this statement and perhaps in another event handler), you can execute this statement by pressing F10. By pressing F10 you’re executing the highlighted statement only. The program will stop at the next line. The next statement to be executed is the `End Select` statement.

Find an instance of the `result` variable in the current event handler, rest the point over it, and you will see the value of the variable after it has been assigned a value. Now you can press F10 to execute another statement or press F5 to return to normal execution mode.

You can also evaluate expressions involving any of the variables in the current event handler by entering the appropriate statement in the Command window. The Command window appears at the bottom of the IDE. If it’s not visible, open the View menu and select Other Windows ➤ Command Window. The current line in the Command window is prefixed with the greater-than symbol (reminiscent of the DOS days). Place the cursor next to it and enter the following statement:

```
? Operand1 / Operand2
```

The quotient of the two values will appear in the following line. The question mark is just a shorthand notation for the `Print` command. If you want to know the current value on the calculator’s display, enter the following statement:

```
? lblDisplay.Text
```

This statement requests the value of a property of a control on the form. The current value of the Label control’s `Text` property will appear in the following line. You can also evaluate math expressions with statements such as the following:

```
```

`Log()` is the logarithm function and it’s a method of the `Math` class. To create a random value between 0 and 1, enter this statement:

```
? Rnd()
```

With time, you’ll discover that the Command window is a very handy tool for debugging applications. If you have a statement with a complicated expression, you can request the values of the individual components of the expression and make sure they can be evaluated.

Now move the pointer of the breakpoint and press F9 again. This will toggle the breakpoint status, and the execution of the program won’t halt the next time this statement is executed.

If the execution of the program doesn’t stop at a breakpoint, it means that the statement is never reached. In this case, you must search for the bug in statements that are executed before the breakpoint is reached. If you didn’t assign the proper value to the `Operator` variable, the `Case` clause for the subtraction operation will never be reached. You should place the breakpoint at the first executable statement of the Equal button’s Click event handler to examine the values of all variables the moment this subroutine starts its execution. If all variables have the expected values, you will continue testing the code forward. If not, you’d have to test the statements that lead to this statement—the statements in the event handlers of the various buttons.
Another simple technique for debugging applications is the Immediate window. Although this isn’t a debugging tool, it’s very common among VB programmers (and very practical, I might add). Many programmers print the values of selected variables after the execution of some complicated statements. To do so, use the statement:

```vbnet
Debug.WriteLine
```
followed by the name of the variable you want to print, or an expression:

```vbnet
Debug.WriteLine(Operand1)
```

This statement sends its output to the Immediate window. This is a very simple technique, but it works. You can also use it to test a function or method call. If you’re not sure about the syntax of a function, pass an expression that contains the specific function to the `Debug.WriteLine` statement as argument. If the expected value appears in the Immediate window, you can go ahead and use it in your code.

### Adding More Features

Now that we have implemented the basic functionality of a hand-held calculator, we can add more features to our application. Let’s add two more useful buttons:

- The +/–, or Negate, button, which inverts the sign of the number on the display
- The 1/x, or Inverse, button, which inverts the number on the display

Open the code window for each of the Command buttons and enter the code from Listing 1.12 in the corresponding Click event handlers. For the +/– button, enter the event handler named `bttnNegate_Click`; for the 1/x button, enter the one named `bttnInverse_Click`.

#### Listing 1.12: Negate and Inverse Buttons

```vbnet
Private Sub bttnNegate_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles bttnNegate.Click
    lblDisplay.Text = -Convert.ToDouble(lblDisplay.Text)
clearDisplay = True
End Sub

Private Sub bttnInverse_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles bttnInverse.Click
    If Convert.ToDouble(lblDisplay.Text) <> 0 Then
        lblDisplay.Text = (1 / Convert.ToDouble(lblDisplay.Text)).ToString
    clearDisplay = True
    End If
End Sub
```
As with the Division button, we don’t attempt to invert a zero value. The operation 1 / 0 is undefined and causes a runtime error. Notice also that I use the value displayed on the Label control directly in the code. I could have stored the `Display.Text` value to a variable and used the variable instead:

```
TempValue = Convert.ToDouble(lblDisplay.Text)
If TempValue <> 0 Then lblDisplay.Text = 1 / TempValue
```

This is also better coding, but in short code segments, we all tend to minimize the number of statements.

You can easily expand the Math application by adding Function buttons to it. For example, you can add buttons to calculate common functions, such as Cos, Sin, and Log. The Cos button calculates the cosine of the number on the display. The code behind this button’s Click event is a one-liner:

```
lblDisplay.Text = Math.Cos(Convert.ToDouble(lblDisplay.Text)).ToString
```

It doesn’t require a second operand, and it doesn’t keep track of the operation. You can implement all math functions with a single line of code.

The preceding statement is fairly complicated, because it combines several statements in a single line of code. This statement retrieves the text on the display and converts it to a Double value. Then it calculates the cosine of this value and finally it converts the result, which is another Double value, to a string and displays it on the calculator’s display. Here’s a simpler, but not common, code segment to perform the same calculations:

```
Dim angle As Double
angle = Convert.ToDouble(lblDisplay.Text)
Dim cos As Double
cos = Math.Cos(angle)
lblDisplay.Text = cos.ToString
```

You may find some of the code segments in this chapter “unnecessarily complicated,” but this isn’t the case. Why shouldn’t we be able to assign the value on the calculator’s display directly to a variable, and we have to convert it to another data type (or cast it to another data type, to use proper programming terms)? You can certainly do it and let the compiler perform the necessary conversions, but this isn’t the best programming practice. It’s actually shortcuts like these that gave BASIC such a bad reputation for years, and you should avoid them. You’ll learn a lot more about variable types in Chapter 2 and you’ll be able to appreciate better the so-called strictly-typed code.

Of course, you should add some error trapping, and in some cases, you can use data-validation techniques. For example, the `Sqr()` function, which calculates the square root of a number, expects a positive argument. If the number on the display is negative, you can issue a warning:

```
If Convert.ToDouble(lblDisplay.Text) < 0 Then
    MsgBox("Can’t calculate the square root of a negative number")
Else
    lblDisplay.Text = 
        Math.Sqr(Convert.ToDouble(lblDisplay.Text)).ToString
End If
```
All math functions are methods of the Math class; that's why they're prefixed by the name of the class. You can also import the Math class to the project with the following statement and thus avoid prefixing the math functions:

```csharp
Imports System.Math
```

The Log() function can calculate the logarithms of positive numbers only. If you add a button to calculate logarithms and attempt to calculate the logarithm of a negative number, the result will be the string "NaN." This value is similar to infinity, and it says that the result is not a valid number (NaN stands for not a number and is discussed in detail in Chapter 2). Of course, displaying a value such as NaN on the calculator’s display isn’t the most user-friendly method of handling math errors. I would validate the data and pop up a message box with the appropriate description, as shown in Listing 1.13.

### Listing 1.13: Calculating the Logarithm of a Number

```csharp
Private Sub btnLog_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnLog.Click
    If Convert.ToDouble(lblDisplay.Text) < 0 Then
        MsgBox("Can't calculate the logarithm of a negative number")
    Else
    End If
    clearDisplay = True
End Sub
```

Because you can’t foresee everything that might go wrong with your application, you should catch errors as they occur at runtime and handle them accordingly. This is the topic of the following section, where you’ll learn how to catch any error that may occur at runtime and stop them from crashing your application. One more feature you could add to the calculator is a limit to the number of digits on the display. Most calculators can display only a limited number of digits. To add this feature to the Math application (if you consider this a “feature”), use the Len() function to find out the number of digits on the display and ignore any digits entered after the number has reached the maximum number of allowed digits.

### Exception Handling

Crashing this application won’t be as easy as crashing the Loan application. If you start multiplying very large numbers, you won’t get an overflow exception. Enter a very large number by typing repeatedly the digit 9; then multiply this value with another equally large value. When the result appears, click the multiplication symbol and enter another very large value. Keep multiplying the result with very large numbers until you exhaust the value range of the Double data type (that is, until the result is so large that it can’t be stored to a variable of the Double type). When this happens, the string “infinity” will appear in the display.
Our code doesn’t include statements to capture overflows, so where did the string “infinity” come from? As you will learn in Chapter 2, it is possible for numeric calculations to return the string “infinity.” It’s Visual Basic’s way of telling you that it can’t handle very large numbers. This isn’t a limitation of VB; it’s the way computers store numeric values: they provide a limited number of bytes for each variable. You will find out more about oddities such as infinity in Chapter 2.

You can’t create an overflow exception by dividing a number with zero, either, because the code will not even attempt to carry out this calculation. In short, the Calculator application is pretty robust. However, we can’t be sure that users won’t cause the application to generate an exception, so we must provide some code to handle all types of errors.

Errors are now called exceptions. You can think of them as exceptions to the normal (or intended) flow of execution. If an exception occurs, the program must execute special statements to handle the exception—statements that wouldn’t be executed normally. I think they’re called exceptions because “error” is a word nobody likes, and most people can’t admit they wrote code that contains errors. The term exception can be vague. What would you rather tell your customers: that the application you wrote has errors or that your code has raised an exception? You may not have noticed it, but the term bug is not used as frequently any more; bugs are now called “known issues.” The term debugging, however, hasn’t changed yet.

VB 6 programmers used the term error to describe something wrong in their code, and they used to write error-trapping code. With VB 2005, your code is error-free—it just raises exceptions every now and then. Both the error-trapping code of VB 6 and the exception-handling features of VB 2005 are supported. The error-trapping code of VB 6 could get messy, so Microsoft added what it calls structured exception handling. It’s a more organized method to handle runtime errors—or exceptions. The basic premise is that when an exception occurs, the program doesn’t crash with an error message. Instead, it executes a segment of code that you, the developer, provide.

How do you prevent an exception raised by a calculation? Data validation won’t help. You just can’t predict the result of an operation without actually performing the operation. And if the operation causes an overflow, you can’t prevent it. The answer is to add a structured exception handler. Most of the application’s code is straightforward, and you can’t easily generate an exception for demonstration purposes. The only place where an exception may occur is the handler of the Equals button, where the calculations take place. This is where we must add an exception handler. The outline of the structured exception handler is the following:

```
Try
    { statements block }
Catch Exception
    { handler block }
Finally
    { clean-up statements block }
End Try
```

The program will attempt to perform the calculations, which are coded in the statements block. If it succeeds, it continues with the clean-up statements. These statements are mostly clean-up code, and the Finally section of the statement is optional. If missing, the program execution continues with the statement following the End Try statement. If an error occurs in the first block of statements, the Catch Exception section is activated, and the statements in the handler block are executed.
The Catch block is where you handle the error. There’s not much you can do about errors that result from calculations. All you can do is display a warning and give the user a chance to change the values. There are other types of errors, however, which can be handled much more gracefully. If your program can’t read a file from a CD drive, you can give the user a chance to insert the CD and retry. In other situations, you can prompt the user for a missing value and continue. If the application attempts to write to a read-only file, for example, chances are that the user specified a file on a CD drive, or a file with its Read-Only attribute set. You can display a warning, exit the subroutine that saves the data and give the user a chance to either select another file name, or change the Read-Only attribute of the selected file. In general, there’s no unique method to handle all exceptions. You must consider all types of exceptions your application may cause and handle them on an individual basis. What’s important about error handlers is that your application doesn’t crash; it simply doesn’t perform the operation that caused the exception (this is also known as the offending operation or offending statement) and continues.

The error handler for the Math application must inform the user that an error occurred and abort the calculations—not even attempt to display a result. If you open the Equals button’s Click event handler, you will find the statements detailed in Listing 1.14.

### Listing 1.14: Revised Equals Button

```vbnet
Private Sub btnEquals_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnEquals.Click
Dim result As Double
Operand2 = Convert.ToDouble(lblDisplay.Text)
Try
    Select Case Operator
    Case "+"
        result = Operand1 + Operand2
    Case "-"
        result = Operand1 - Operand2
    Case "*"
        result = Operand1 * Operand2
    Case "/"
        If Operand2 <> "0" Then result = Operand1 / Operand2
    End Select
    lblDisplay.Text = result
Catch exc As Exception
    MsgBox(exc.Message)
    result = "ERROR"
Finally
    clearDisplay = True
End Try
End Sub
```
Most of the time, the error handler remains inactive and doesn’t interfere with the operation of the program. If an error occurs, which most likely will be an overflow error, the error-handling section of the Try...Catch...End Try statement will be executed. This code displays a message box with the description of the error, and it also displays the string “ERROR” on the display. The Finally section is executed regardless of whether an exception occurred or not. In this example, the Finally section sets the clearDisplay variable to True so that when another digit button is clicked, a new number will appear on the display.

**NOTE** The exc variable represents an exception; it exposes a few properties in addition to the Message property, which is the description of the exception. For more information on the members of the Exception class and how to handle exceptions, see Chapter 31.

**Executable Files**

So far, you have been executing applications within Visual Basic’s environment. However, you can’t expect the users of your application to have Visual Studio installed on their systems. If you develop an interesting application, you won’t feel like giving away the code of the application (the source code, as it’s called). Applications are distributed as executable files, along with their support files. The users of the application can’t see your source code, and your application can’t be modified or made to look like someone else’s application (that doesn’t mean it can’t be copied, of course).

**NOTE** An executable file is a binary file that contains instructions only the machine can understand and execute. The commands stored in the executable file are known as machine language.

Applications designed for the Windows environment can’t fit in a single file—it just wouldn’t make sense. Along with the executable files, your application requires support files, and these files may already exist on many of the machines on which your application will be installed. That’s why it doesn’t make sense to distribute huge files. Each user should install the main application and only the support files that aren’t already installed on their computer.

The executable will run on the system on which it was developed because the support files are there. Under the project’s file, you will find two folders named Bin and Obj. Open the Bin folder, and you will see that it contains a subfolder named Debug. This is where you will find the executable, which is named after the project and has the extension .exe. Make sure that no instance of VS is running on your computer and then double-click the icon of the MathCalculator.exe or LoanCalculator.exe file. The corresponding application will start outside the Visual Studio IDE, and you can use it like any other application on your PC. You can create desktop shortcuts to the two applications.

You can distribute the EXE file to other machines. The application will work only if the .NET Framework 2.0 is already installed on the target machine. If not, users will have to download version 2.0 of the Framework and install it on their machine before they can execute your application. In the last part of this book, you will learn about the various methods of distributing an application, including the generation of a Setup application that installs your application, along with the Framework if needed, to another machine.
Quick Chapter Review

Visual Studio 2005 is an integrated environment for developing, testing, and debugging applications. Visual Basic 2005 is one of the languages supported by Visual Studio. A Windows application consists of a visual interface and code. The visual interface is what users see at runtime: a form with controls with which the user can interact—enter strings, check or clear check boxes, click buttons, and so on. The visual interface of the application is designed with visual tools: you drop controls from the Toolbox window onto the form, size and align the controls on the form, and finally set their properties through the Properties window. The controls include quite a bit of functionality right out of the box. This functionality is readily available to your application without a single line of code. A TextBox control with its MultiLine property set to True and its Scrollbars property set to Vertical is a complete, self-contained text editor. It can even exchange text with other text-based applications in the Windows environment with no code at all. Being able to put together a semi-functional application with simple mouse operations without writing code is why Visual Studio is known as a RAD tool (RAD stands for Rapid Application Development).

Once the visual interface has been designed, you can start coding the application. Windows applications follow an event-driven model: We code the events to which we want our application to respond: The Click event of the various buttons are typical events to which an application reacts. The Button control fires many more events, such as the DoubleClick and the Enter/Leave events. We rarely program these events. The TextBox control fires some 60 events, but most applications don’t react to a single one of them. You select the actions to which you want your application to react and program these events accordingly.

When an event is fired, the appropriate event handler is automatically invoked. Event handlers are subroutines that pass two arguments to the application: the sender object (which is an object that represents the control that fired the event) and the e argument (which carries additional information about the event). Most controls fire multiple events when a key is pressed while they have the focus: the KeyDown, KeyPress, and KeyUp events. The e argument of these events reports to the application the key that was pressed, as well as the state of the Control, Alt, and Shift keys at the time the key was pressed. To program a specific event for a control, double-click the control on the design surface and the editor will come up with the default event for the control. You can select any other event to program in the Events combo box at the top of the editor’s window. Every time you program an event handler, you should type the name of the event handler’s second argument (it’s always the e argument) followed by a period to see the properties exposed by this argument. Most likely, you will use one or more of these properties in your code.

Most developers write code as if the application will be used by extremely careful users, but this isn’t the case. Users make mistakes (they enter invalid amounts and dates, for example). Numerous conditions can cause an application to crash: to terminate with an error message. A professional application should be able to detect these abnormal conditions and handle them gracefully. To begin with, you should always validate your data before you attempt to use them in your code. A well known computer term is the “garbage-in garbage-out” term, which means that you shouldn’t perform any calculations on invalid data. Before you attempt to calculate the monthly payment of a loan, make sure that the user has supplied valid (if not reasonable) values for the loan’s parameters. Even so, errors may still occur. Multiplying two extremely large numbers with one another, or dividing a very large by a very
small number may cause overflow conditions. To handle these conditions use the Visual Basic error-catch-
catching mechanism, the \texttt{Try...Catch} statement. This statement allows you to execute a statement that
may cause a \textit{runtime exception}. If the statement doesn't produce an exception, the program continues
as usual. If it does, the application executes the statements you provide to handle the error condition.
You usually can't remedy runtime exceptions from within your code, but you can provide a good indi-
cation as to what might have gone wrong and give users a chance to correct the situation (supply a dif-
ferent value, insert a missing CD, connect to a network printer, and so on). At the very least, you should
save the user data and then terminate the application.

\section*{Moving On}

This chapter was a quick introduction to the environment you'll be using to design your applica-
tions. It's a very rich environment, and it will take you a while to become quite comfortable with
it. Keep in mind that you won't need most of the menus and toolbars when building simple Win-
dows applications.

You also wrote your first few applications and you should be able to create new projects, design
visual interfaces, and program simple events such as the click of a button control. In the following two
chapters, you'll read about the language itself. After you understand the nature of variables, the var-
ious data types you can work with, and the statements of the language, you'll be able to write real
applications.