

Introduction to Visual Basic .NET

- 2.1 Elements of a Visual Basic Application
- 2.2 Getting Started in Visual Basic
- 2.3 Adding an Event Procedure
- 2.4 Adding Controls
- 2.5 Adding Additional Event Procedures
- 2.6 Focus on Program Design and Implementation:
Creating a Main Menu
- 2.7 Knowing About: The Help Facility
- 2.8 Common Programming Errors and Problems
- 2.9 Chapter Review

Goals

In this chapter we begin learning about the fundamentals of programming and Visual Basic .NET. First we examine the two elements that are required by every practical Visual Basic program: the screens and instructions seen by the user, and the “behind the scenes” processing that is done by the program. We then present the basic design windows that you must be familiar with to produce such programs. Finally, we show you how to use these design windows to create the visual user interface, or GUI, and then add processing instructions.

2.1 Elements of a Visual Basic Application

Visual Basic was initially introduced in 1991 as the first programming language that directly supported programmable graphical user interfaces using language-supplied objects. From that time until 2002, there were five other versions released, each version having features that increased the power of the language. In 2001, Microsoft released the .NET (pronounced “dot net”) platform. Visual Basic .NET, or VB.NET, is an upgrade to the last version of VB (version 6.0) that conforms to the .NET platform. As you will see in subsequent chapters, the changes in VB.NET allow programmers to write Web or desk-top applications within the same language. In addition, VB.NET is fully object-oriented as opposed to prior versions that had many, but not all, of the elements of an object-oriented language. This book is based on VB.NET. In the balance of the book we will sometimes refer to Visual Basic as VB, omitting .NET.

From a programming viewpoint, Visual Basic is an object-oriented language that consists of two fundamental parts: a visual part and a language part. The visual part of the language consists of a set of objects, while the language part consists of a high-level procedural programming language. These two elements of the language are used together to create applications. An *application* is simply a Visual Basic program that can be run under the Windows operating system. The term *application* is preferred to the term *program* for two reasons: one, it is the term selected by Microsoft to designate any program that can be run under its Windows Operating System (all versions) and two, it is used to avoid confusion with older procedural programs that consisted entirely of only a language element. Thus, for our purposes we can express the elements of a Visual Basic application as:

```
Visual Basic Application = Object-Based Visual Part +  
Procedural-Based Language Part
```

Thus, learning to create Visual Basic applications requires being very familiar with both elements, visual and language.

The Visual Element

From a user’s standpoint, the visual part of an application is provided within a window. This is the graphical interface that allows the user to see the input and output provided

by the application. This user interface is referred to as the *graphical user interface* (GUI). From a programmer's perspective the GUI is constructed by placing a set of visual objects on a blank window, or form, when the program is being developed. For example, consider Figure 2-1, which shows how a particular application would look to the user. From a programmer's viewpoint, the application shown in Figure 2-1 is based on the design form shown in Figure 2-2. The points displayed on the form are a design grid used to arrange objects on the form and are only displayed during design time.

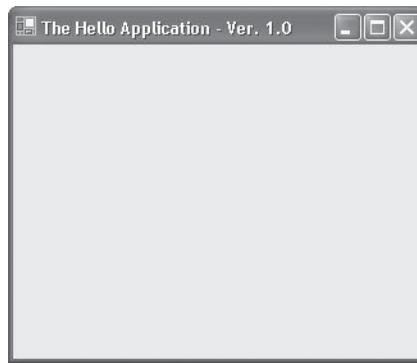


Figure 2-1 A User's View of an Application

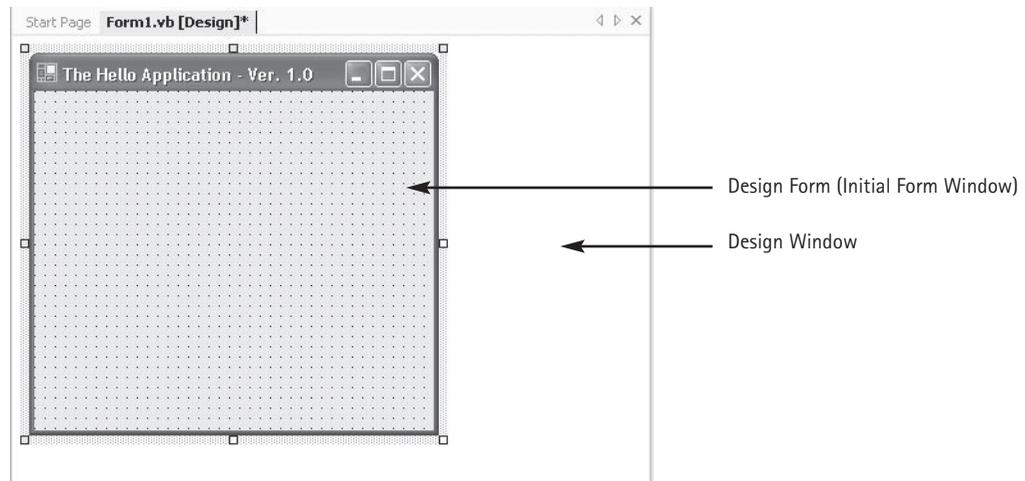


Figure 2-2 The Design Form on which Figure 2-1 is Based

The programmer can place various objects on this form, which is itself a Visual Basic object. When an application is run, the form becomes a window that provides the background for the various objects placed on the form by the programmer. The objects on the window become the controls used to direct program events. Let's take a moment to look at the objects provided in the Visual Basic Toolbox. The standard object Toolbox, which is illustrated in Figure 2-3, contains the objects we will use in constructing each graphical user interface.

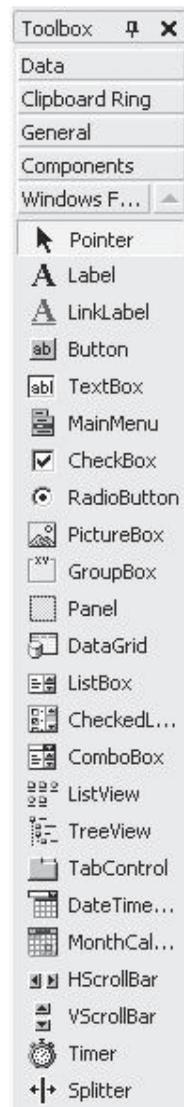


Figure 2-3 The Standard Visual Basic Toolbox

Programmer Notes

Forms and Controls

When an application is being designed, a *form* is a container upon which controls are placed. When an application is executed, the form becomes either a window or a dialog box. Forms can be of two types: SDI or MDI. The acronym SDI stands for Single Document Interface, which means that only one window at a time can be displayed by an application. SDI applications can have multiple windows, but a user can only view one window at a time. The acronym MDI refers to Multiple Document Interface, which means the application consists of a single “parent” or main window that can contain multiple “child” or internal windows. For example, the Notepad application supplied with the Windows operating system is an SDI application, while Excel and Access are both MDI applications.

A control is an object that can be placed on a form, and has its own set of recognized properties, methods, and events. Controls are used to receive user input, display output, and trigger event procedures.

A majority of applications can be constructed using a minimal set of objects provided by the standard object Toolbox. This minimal set consists of the Label, TextBox, and Button objects. The next set of objects that are more frequently found in applications include the CheckBox, RadioButton, ListBox, and ComboBox. Finally, the Timer and PictureBox can be used for constructing interesting moving images across the window. Table 2-1 lists these object types and describes what each object is used for. The remaining sections of the text will describe the use of objects in the toolbox, with special emphasis on the four objects (Label, TextBox, Button, and ListBox) that you will use in almost every application that you develop.

In addition to the basic set of controls provided in VB, a great number of objects can be purchased either for special purpose applications or to enhance standard applications.

Table 2-1 Fundamental Object Types and Their Uses

Object Type	Use
Label	Create text that a user cannot directly change.
TextBox	Enter or display data.
Button	Initiate an action, such as a display or calculation.
CheckBox	Select one option from two mutually exclusive options.
RadioButton	Select one option from a group of mutually exclusive options.
ListBox	Display a list of items from which one can be selected.
ComboBox	Display a list of items from which one can be selected, as well as permit users to type the value of the desired item.
Timer	Create a timer to automatically initiate program actions.
PictureBox	Display text or graphics.

Don't be overwhelmed by all of the available controls. At a minimum, you will always have the objects provided by the standard Toolbox available to you, and these are the ones we will be working with. Once you learn how to place the basic control objects on a form, you will also understand how to place the additional objects, because every object used in a Visual Basic application, whether it is selected from a standard or purchased control, is placed on a form in the same simple manner. Similarly, each and every object contains two basic characteristics: properties and methods.

An object's *properties* define particular characteristics of the object. For example, the properties of a text box include the location of the text box on the form, the color of the box (the background color), the color of text that will be displayed in the box (the foreground color), and whether it is read-only or can also be written to by the user.

Methods are predefined procedures that are supplied with the object for performing specific tasks. For example, you can use a method to move an object to a different location or change its size.

Additionally, each object from the Toolbox recognizes certain actions. For example, a button recognizes when the mouse pointer is pointing to it and the left mouse button is clicked. These types of actions are referred to as *events*. In our example, we would say that the button recognizes the mouse-click event. However, once an event is activated, we must write our own procedures to do something in response to the event. This is where the language element of Visual Basic comes into play.

The Language Element

Before the advent of GUIs, computer programs consisted entirely of a sequence of instructions. Programming was the process of writing these instructions in a language to which the computer could respond. The set of instructions and rules that could be used to construct a program were called a programming language. Frequently, the word *code* was used to designate the instructions contained within a program. With the advent of graphical user interfaces the need for code (program instructions) has not gone away—rather, it forms the basis for responding to the events taking place on the GUI. Figure 2-4 illustrates the interaction between an event and a program code.

As illustrated in Figure 2-4, an event, such as clicking the mouse on a button, sets in motion a sequence of actions. If code has been written for the event, the code is exe-

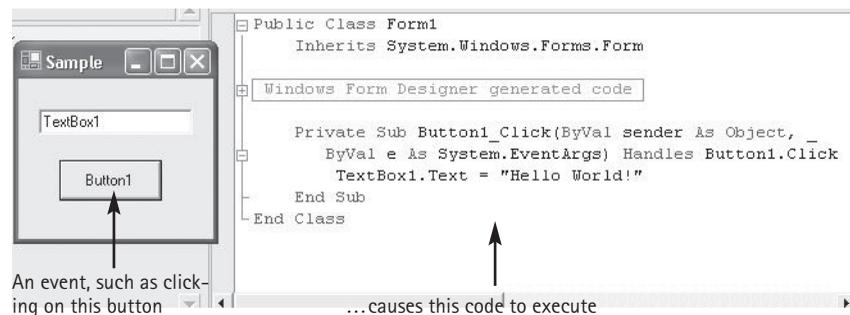


Figure 2-4 An Event "Triggers" the Initiation of a Procedure

cuted; otherwise the event is ignored. This is the essence of GUIs and event-driven applications—the selection of executed code depends on what events occur, which ultimately depends on what the user does. The programmer must still write the code that performs the desired action.

Visual Basic is a high-level programming language that supports all of the procedural programming features found in other modern languages. These include statements to perform calculations, permit repetitive instruction execution, and allow selection between two or more alternatives.

With these basics in mind, it is now time to create our first Visual Basic application. In the next section, we introduce the Visual Basic programming environment and create an application that uses only a single object: the form itself. We will then add additional objects and code to create a more complete Visual Basic application.

Exercises 2.1

1. List the two elements of a Visual Basic Application.
2. What is the purpose of a GUI and what elements does a user see in a GUI?
3. What does a Visual Basic toolbox provide?
4. Name and describe the four most commonly used Toolbox objects.
5. When an application is run, what does a design form become?
6. What is executed when an event occurs?

2.2 Getting Started in Visual Basic

It's now time to begin designing and developing Visual Basic programs. To do this, you will have to bring up the opening Visual Basic screen and understand the basic elements of the Visual Basic development environment. Visual Studio is the integrated development environment (IDE, pronounced as both I-D-E, and IDEE) used to create, test, and debug projects. Developers can also use Visual Studio to create applications using languages other than Visual Basic, such as C# and Visual C++. To bring up the opening Visual Basic screen, either click the Microsoft Visual Studio .NET icon (see Figure 2-5), which is located within the Microsoft Visual Studio .NET Group, or, if you have a shortcut to Visual Basic .NET on the desktop, double-click this icon.

When you first launch Visual Basic .NET, the Start Page similar to the one shown in Figure 2-6 will appear. While this page provides links to Web pages to help developers find useful information, we will be concerned only with the following three areas: the central rectangle displaying recent programs, the Open Project button, and the New Project button. Clicking on any of the recent programs causes VB.NET to retrieve the program and load it into the IDE. Clicking the Open Project button opens a standard Windows file dialog box permitting you to retrieve a previously saved Visual Basic program and load it into the IDE. Clicking the New Project button opens the dialog box shown in Figure 2-7. This dialog box provides a choice of eleven project types, shown

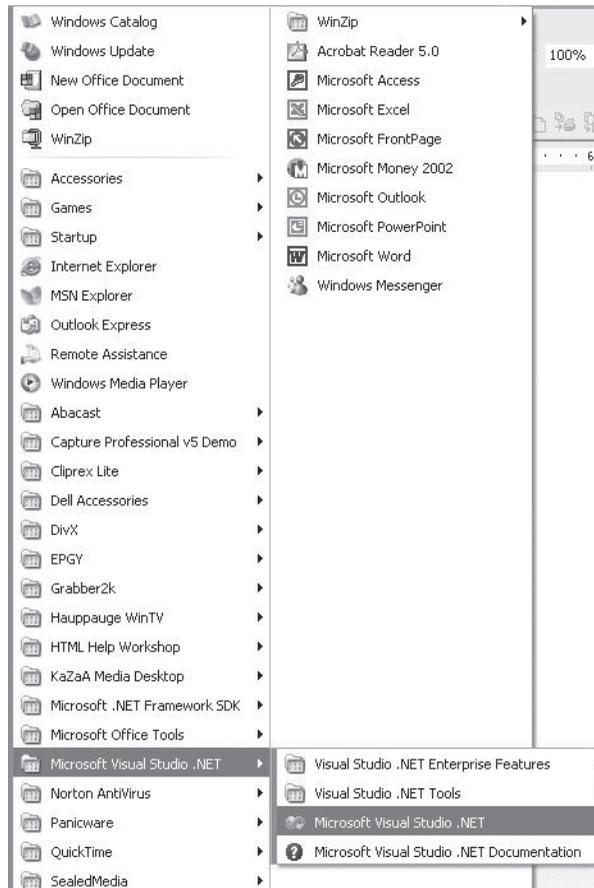


Figure 2-5 The Microsoft Visual Studio .NET Icon within the Visual Studio .NET Group

in Table 2-2. In this text, we will be concerned with Windows Applications and ASP.NET Web Applications.

Click the New project button to open the New Project Dialog box displayed in Figure 2-7. Click the OK button to create a new project. Don't be concerned with the Name and Location, as the goal here is to display the IDE screen as shown in Figure 2-8

The four windows shown in Figure 2-8 are, as marked, the Toolbox window, the Initial Form window, the Solution window, and the Properties window. Additionally, directly under the Title bar at the top of the screen sits a Menu bar and a Toolbar, which should not be confused with the Toolbox window. Table 2-3 lists a description of each of these components. Before examining each of these components in depth, it will be useful to consider the IDE as a whole and how it uses standard Windows keyboard and mouse techniques.

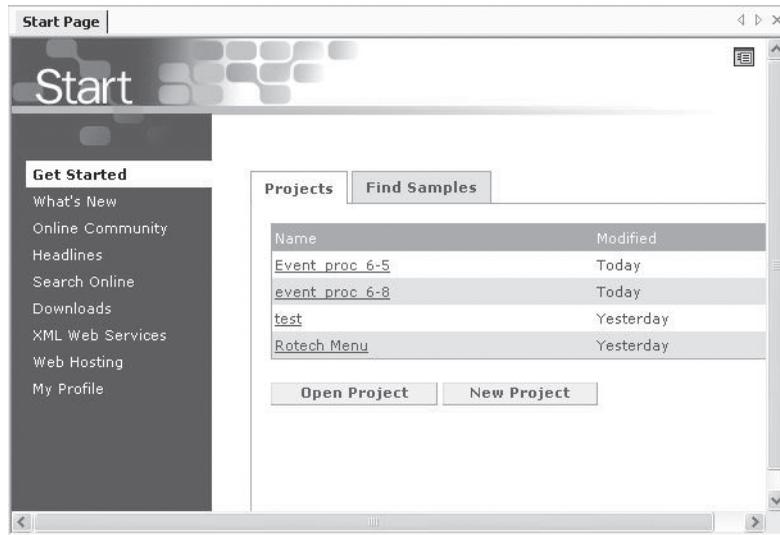


Figure 2-6 The Visual Basic .NET Start Page

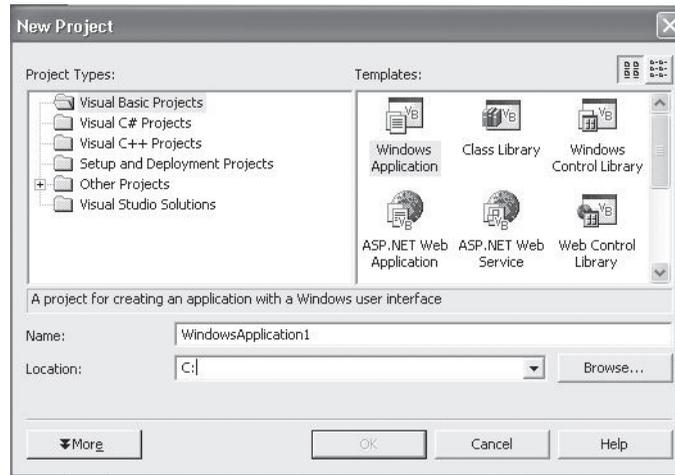


Figure 2-7 New Project Dialog

The IDE as a Windows Workspace

The IDE consists of three main components: a GUI designer, a code editor, and a debugger. In the normal course of developing a Visual Basic program, you will use each of these components. Initially, we will work with GUI designer, which is the screen shown in Figure 2-8. The screen is actually composed of a main “parent” window containing multiple “child” windows.

Table 2–2 Eleven Project Types

Windows Application	Class Library
Windows Control Library	ASP.NET Web Application
ASP.NET Web Service	Web Control Library
Console Application	Windows Service
Empty Project	Empty Web Project
New Project in Existing Folder	

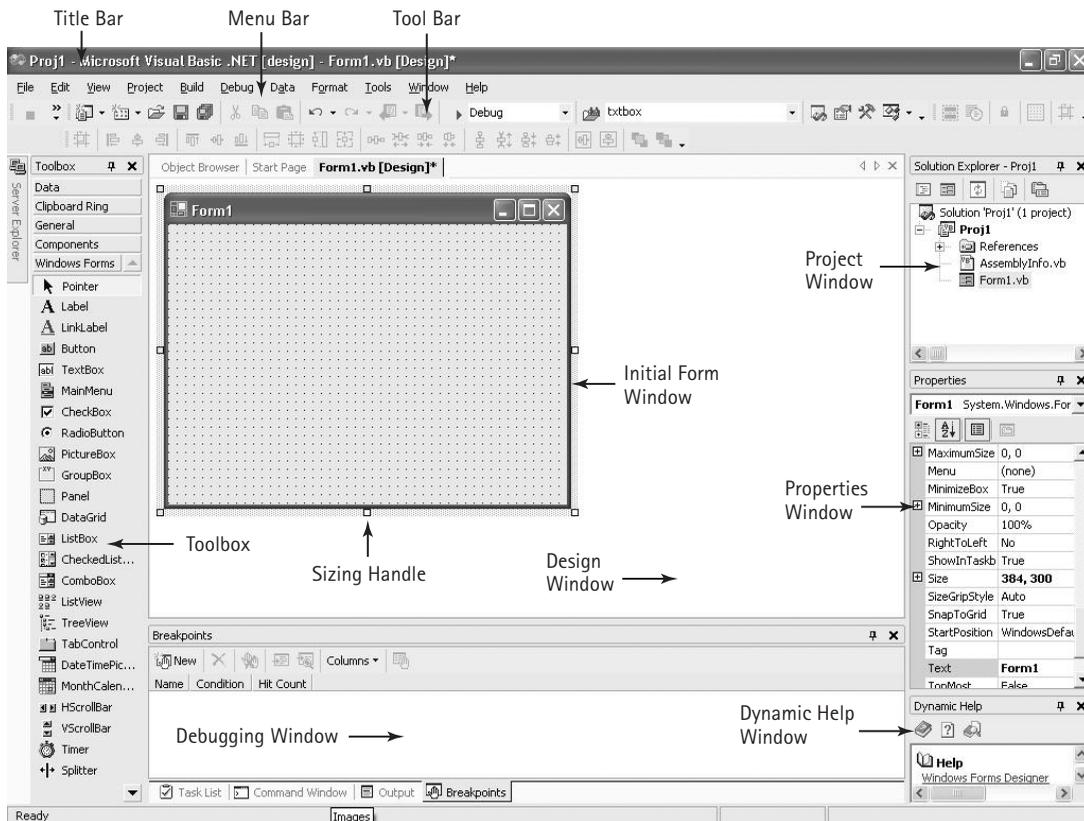


Figure 2–8 The Integrated Development Environment's Initial Screen

As a Windows-based application, each child window within the overall parent window, as well as the parent window itself, can be resized and closed in the same manner as all windows. To close a window you can double-click the X in the upper right-hand corner of each window. Windows can be resized by first moving the mouse pointer to a window's border. Then, when the pointer changes to a double-headed arrow, click and drag the border in the desired direction. You can move each window by clicking the mouse within the window's Title bar, and then dragging the window to the desired position on the screen.

Table 2-3 Initial Development Screen Components

Component	Description
Title Bar	The colored bar at the top edge of a window that contains the window name.
Menu Bar	Contains the names of the menus that can be used with the currently active window. The menu bar can be modified, but cannot be deleted from the screen.
Toolbar	Contains icons that provide quick access to commonly used Menu Bar commands. Clicking an icon, which is referred to as a button, carries out the designated action represented by that button.
Layout Toolbar	Contains buttons that enable you to format the layout of controls on a form. These buttons enable you to control aligning, sizing, spacing, centering, and ordering controls.
Toolbox	Contains a set of controls that can be placed on a Form window to produce a graphical user interface (GUI).
Initial Form Window	The form upon which controls are placed to produce a graphical user interface (GUI). By default, this form becomes the first window that is displayed when a program is executed.
Properties Window	Lists the property settings for the selected Form or control and permits changes to each setting to be made. Properties such as size, name, and color, which are characteristics of an object, can be viewed and altered either from an alphabetical or category listing.
Solution Window	Displays a hierarchical list of projects and all of the items contained in a project. Also referred to as both the Solution Resource Window and the Solution Explorer.
Form Layout Window	Provides a visual means of setting the Initial Form window's position on the screen when a program is executed.

As with any other Windows application, Visual Basic makes use of a menu bar to provide an interface to the programmer. For example, if you wish to save a program you have been working on and start a new one, you would choose the File item from the menu bar, which will bring up the File submenu shown in Figure 2-9. From this menu you can save the current project by using the Save All option, then click the New option and click Project (Figure 2-10). The New Project dialog box appears. To access an existing program, you can also use the menu bar File item, except you would then click Open and click Project to reopen a previously saved program. Similarly, these two options can also be activated by clicking the appropriate icons on the Toolbar located immediately under the Menu bar.

Once a program has been opened, you can always use the View item on the menu bar to display any windows that you need. For example, if either the Properties or Toolbox windows are not visible on the development screen, select the View item from the

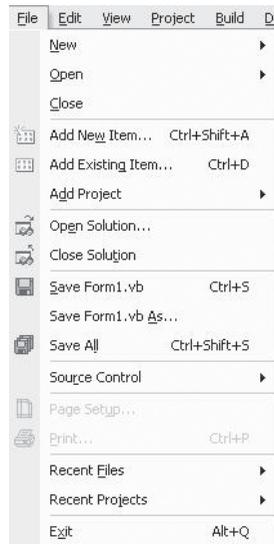


Figure 2-9 The File SubMenu

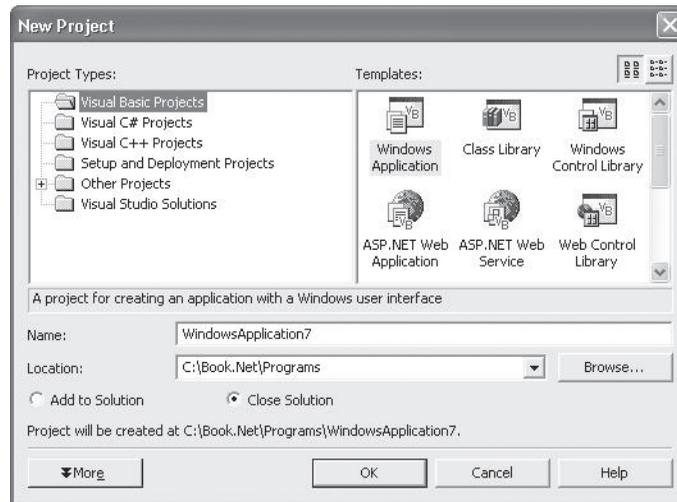


Figure 2-10 The New Project Dialog Box

menu bar. This will open the View submenu illustrated in Figure 2-11. From this submenu, click the Properties Window or click Toolbox and then click a Toolbox item to open the desired window. Note in Figure 2-11 that all Visual Basic's windows are listed in the View submenu.

Having examined the Menu bar and how it is used to configure the development screen, make sure that you go back to the initial development screen shown in Figure 2-8. If any additional windows appear on the screen, close them by clicking each win-

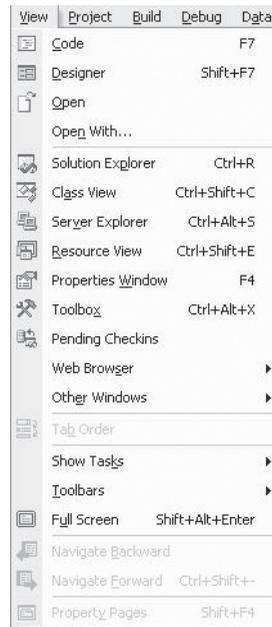


Figure 2–11 The *V*iew SubMenu

dow's close button (the box with the X in the upper right corner). The window does not have to be active to do this.

Note that the caption within the top title bar of the screen shown in Figure 2–8 contains the words *Microsoft Visual Basic* [design]. The word [design] in the top Title bar caption is important because it indicates that we are in the design phase of a Visual Basic program. At any point within our development, we can run the program and see how it will look to the user.

Once the design windows are visible, creating a Visual Basic application requires the following three steps:

1. Create the graphical user interface (GUI).
2. Set the properties of each object on the interface.
3. Write the code.

The foundation for creating the GUI (Step 1) is the Initial Form window. It is on this design form that we place various objects to produce the interface we want users to see when the program is executed. When the program is run, the design form becomes a window and the objects that we place on the design form become visual controls that are used to input data, display output, and activate events. The objects that we can place on the design form are contained within the Toolbox. The Toolbox drop-down list is shown in Figure 2–12. The visual controls we will be using are under the Windows Forms drop-down list.

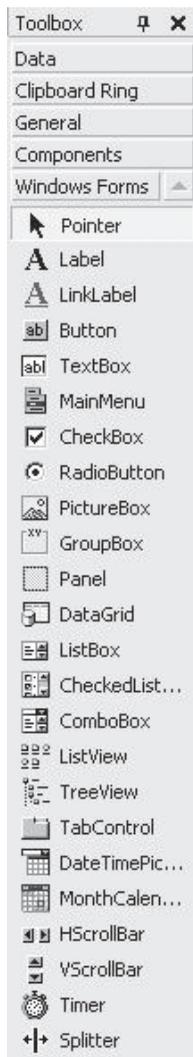


Figure 2-12 The Standard Object Toolbox

Programmer Notes

Opening the Basic Design Windows

To create a Visual Basic program, you will need the following three windows: the Toolbox window for selecting objects, a Form window for placing objects, and a Properties window for altering an object's properties. Additionally, the Solution Explorer window should be vis-

ible when you begin. If any of these windows are in the background, clicking on them will activate them and bring them to the foreground, or you may use the following procedures:

For a Form window:

For a new project, first click File from the menu bar, click New from the submenu, and click Project (or use the hot key sequence Alt+F, then N, then P). This will open a new Form window.

For an existing project, click File from the menu bar, click Open from the submenu, and then click Project (or use the hot key sequence Alt+ F, then O, then P). This will open a project window. Then select the project file.

For a Toolbox window:

To either activate an existing Toolbox window or open one if it is not on the screen, do either of the following:

- Click View and then click Toolbox.
- Use the hot key sequence Alt+V and then press the X key (Alt+V / X).

For a Properties window:

To activate an existing Properties window or open one if it is not on the screen, do one of the following:

- Click View and then click Properties Window.
- Use the hot key sequence Alt+V, and then press the W key (Alt+V / W).
- Press the F4 function key.

For a Solution Explorer window:

To activate an existing Solutions window or open one if it is not on the screen, do one of the following:

- Click View and then click Solution Explorer.
- Use the hot key sequence Alt+V and then press the P key (Alt+V / P).

Don't be confused by all of the available objects. Simply realize that Visual Basic provides a basic set of object types that can be selected to produce a graphical user interface. The balance of this book explains what some of the more important objects represent and how to design a Visual Basic application using them. To give you an idea of how simple it is to design such an interface, move the mouse pointer to the Button object in the Toolbox and double-click the Button icon. Note that a Button object appears in the form. Placing any object from the Toolbox onto the form is this simple. Now click the newly created button within the form and press the Delete key to remove it.

Setting an Object's Properties

As previously stated, all objects on a form have properties, which define where on the form the object will appear (the object's vertical and horizontal position relative to the left-hand corner of the form), the color of the object, its size, and various other attributes. To understand these properties, we will now examine the most basic object in Visual Basic: the form. Like any other object, the Form object has properties that define how it will appear as a screen when the program is run. As an introduction to the ease with which properties are set, we will first explore the Form object's properties. To do this, you need to have a basic design screen open (see Figure 2-8).

First, click the Properties window to activate it. The Properties window, which should appear as shown in Figure 2-13, is used for setting and viewing an object's properties.

The Properties window allows properties to be listed in alphabetic order, by property name, or by property category. By default, the properties are sorted by category. To change to property categories, click the first button on the Properties window toolbar. To switch back to alphabetic sort, click the second button on the Properties window toolbar. When viewed by category, individual properties are grouped according to appearance, font, position, behavior, and so on.

No matter the order of properties selected, the first box within a Properties window is the *Object Identification box*, located immediately under the window's Title bar. This box lists the name of the object and its object type. In Figure 2-13 the name of the object is Form1 and its type is Form.

The two columns within the Properties window are where individual object properties are identified. The column on the left is the properties list, which provides the names of all the properties of the object named in the object box. The column to the right is the settings list, which provides the current value assigned to the property on the left. The currently selected property is the one that is highlighted. For example, the

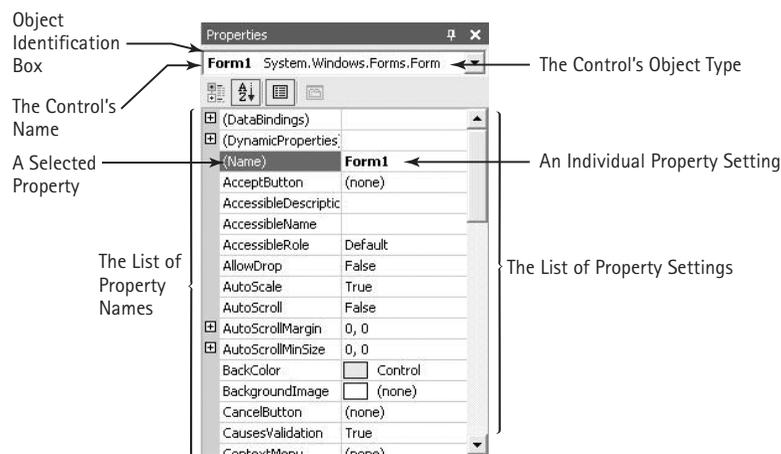


Figure 2-13 The Properties Window

Name property is highlighted in Figure 2-13. The value assigned to a highlighted property can be changed directly in the property settings list.

Take a moment now and, using the keyboard arrows, move down through the Properties window. Observe that each property is highlighted as it is selected, and that the description of the highlighted property is displayed in the description box at the bottom of the window.¹ Now move back up until the Name property at the top of the alphabetical list is highlighted. The name Form1 shown in the figure is the default name that Visual Basic gives to the first Form object provided for a new program. If a second form were used, it would be given the default name Form2, the third form would be named Form3, and so on.

The Name Property There is nothing inherently wrong with keeping the default name that Visual Basic provides for each Form object that you use. However, good programming practice requires that all Form and other object names be more descriptive and convey some idea about what the object is used for. The names that are allowed for all objects, of which a Form is one, are also used to name other elements in the Visual Basic programming language and are collectively referred to as *identifiers*. Identifiers can be made up of any combination of letters, digits, or underscores (`_`) selected according to the following rules:

1. The first character of an identifier must be a letter.
2. Only letters, digits, or underscores may follow the initial letter. Blank spaces, special characters, and punctuation marks are not allowed. Use the underscore or capital letters to separate words in an identifier consisting of multiple words.
3. An identifier can be no longer than 1016 characters.
4. An identifier cannot be a keyword. (A *keyword* is a word that is set aside by the language for a special purpose.)²

Using these rules, development teams may then use whatever naming conventions they choose. In this book, form names begin with `frm` and our first form will always be given the name `frmMain`. To assign this name to our current form, do the following: if the name property is not already highlighted, click the name property and change the name to `frmMain` by directly typing in the settings list to the right of the Name property. The name change takes effect when you either press the Enter key, move to another property, or activate another object.

¹The description box can be toggled on or off by clicking the right mouse button from within the Properties window.

²More specifically, an identifier cannot be a restricted keyword. A restricted keyword is a word that is set aside by the language for a specific purpose and can only be used in a specified manner. Examples of such words are If, Else, and Loop. Other languages refer to such words as reserved words. Use the Help Facility and search for "Word Choice" to find a table of keywords.

Programmer Notes

The Properties Window

The Properties window is where you set an object's initial properties. These are the properties that the object will exhibit when the application is first run. These properties can be altered later, using procedural code.

To Activate the Properties Window:

To activate a particular object's Properties window, first click the object to select it and then press the F4 function key. You can also click **V**iew and then click Properties **W**indow (or use the hot-key sequence Alt+V, then W). This will activate the Properties window for the currently active object. Once the Properties window is active, clicking the down-facing arrowhead to the right of the object identification box will activate a drop-down list that can be used to select any object on the form, including the form itself.

To Move to a Specific Property:

First, make sure that the Properties window is active. You can then cursor up or down through the properties by using the up and down arrow keys or by simply clicking the desired property with the mouse.

The Text Property A form's name property is important to the programmer when developing an application. However, it is the form's Text property that is important to the user when a program is run, because it is the Text property that the user sees within the window's Title bar when an application is executing.

To change the Text property, select it from the Properties window. To do this, make sure the Properties window is selected and use the arrow keys to position the cursor on the Text property. Now change the caption to read:

```
The Hello Application - Version 1 (pgm2-1).
```

If the caption is larger than the space shown in the settings box, as is shown in Figure 2-14, the text will scroll as you type it in. When you have changed the text, the design screen should appear as shown in Figure 2-14.

Before leaving the Properties window to run our application, let's take a moment to see how properties that have restricted values can also be changed.

We changed both the Name and Text properties by simply typing new values. Certain properties have a fixed set of available values. For example, the Cursor property, which determines the type of cursor that will appear when the program runs, can be selected from a list of defined cursors. Similarly, the Font property, which determines the type of font used for an object's displayed text can only be selected from a list of available fonts. Likewise, the BackColor and ForeColor properties, which determine the background color and the color of text displayed in the foreground, can only be selected

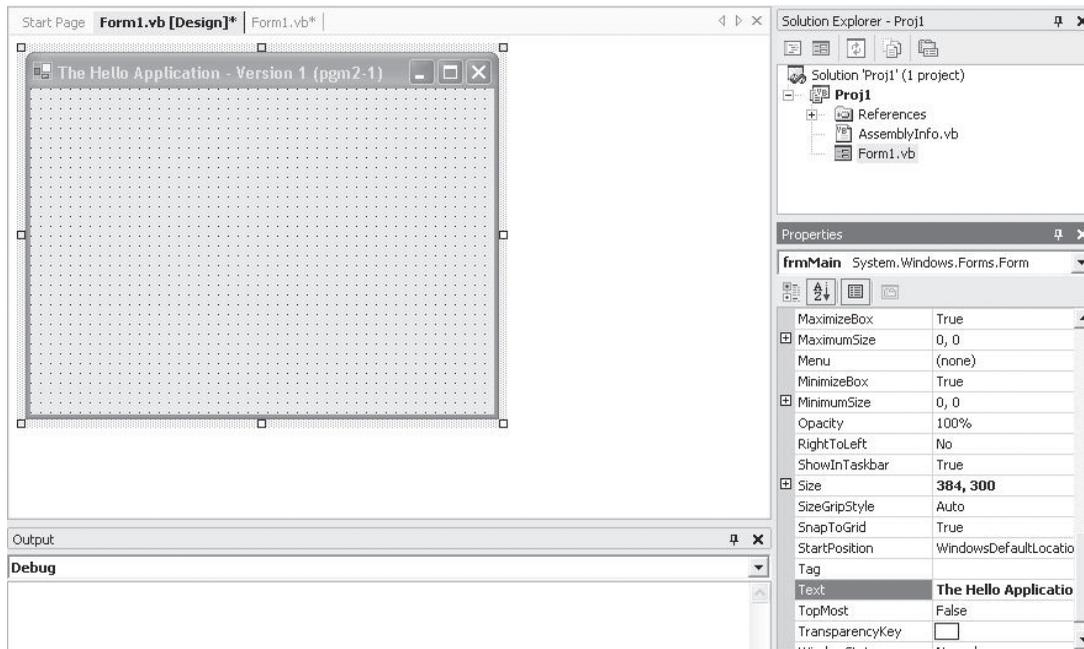
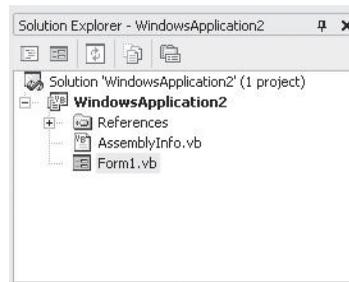


Figure 2–14 The Design Screen after the Text Property Change

from a predefined palette of colors. When one of these properties is selected, either a down-facing arrowhead property button (▼) or an ellipses (...) property button will appear to the right of the selected setting. Clicking this button will show you the available settings. You can then make a selection by clicking the desired value. In the case of colors, a palette of available colors is displayed, and clicking on a color sets the numerical code for the chosen color as the property's value.

The Solution Explorer Window Although Visual Basic was initially designed as a tool to build smaller desktop applications, it has expanded in scope to allow for the construction of enterprise-wide systems. To support this capability, Visual Basic .NET has adopted a more complex, but logical, way to associate different elements of an application. At the top level is a *solution* file. A solution consists of a set of projects. Only in cases of a complex application will there be more than one project. For all examples and assignments in this book, there will be only one project. A project consists of all the programming components we write. We have just seen one of these components: a form. There are many others that we can use to build an application. All of these components are associated together under the project.

The Solution Explorer window displays a hierarchical list of projects and components contained in a project, as shown in Figure 2–15. As files are added or removed from a project, Visual Basic reflects all of your changes within the displayed hierarchical tree.

Figure 2–15 *Solution Explorer Window*

The hierarchical tree uses the same folder tree structure found in Windows, which means that you can expand and contract tree sections by clicking on the plus (+) and minus (–) symbols, respectively. As always, sections of the tree that are hidden from view due to the size of the window can be displayed using the attached scroll bars.

The Solution Explorer window is extremely useful in providing a visual picture of the project files and in providing a rapid means of accessing, copying, and deleting files associated with a project. For example, if a Form object is not displayed on the design screen, you can make it visible by double-clicking the desired Form object from within the hierarchical tree. In a similar manner, you can expand a folder or bring up both code and visible objects by clicking one of the five icons shown in Figure 2–15. Two further items relating to the Solution Explorer window are worth noting. First, Visual Basic assigns default names to solutions (Solution1), projects (Project1), and forms (Form1). These names can be changed by selecting the solution, project, or form with the Solution Explorer and right-clicking to open a menu. Select the Rename menu option and enter the new name.

Second, it is worth noting that the first two items in the View submenu, Code and Object, have icons identical to those displayed in the Solution Explorer window shown in Figure 2–15. Thus, both code and objects can be displayed by using either the View submenu or the Solution Explorer window.

Running an Application

At any time during program development, you can run your program using one of the following three methods:

1. Select the Debug Menu and click Start.
2. Press the F5 function key.
3. Use the hot key sequence Alt+D, then press the S key.

If you do this now for Program 2–1, the program will appear as shown in Figure 2–16. Before doing so, change the Name property of the Form back to Form1. (You had changed it to frmMain.) We will explain later why this is necessary.

Notice that when the program is run, the form becomes a standard window. Thus, even though we have not placed any objects on our form or added any code to our pro-

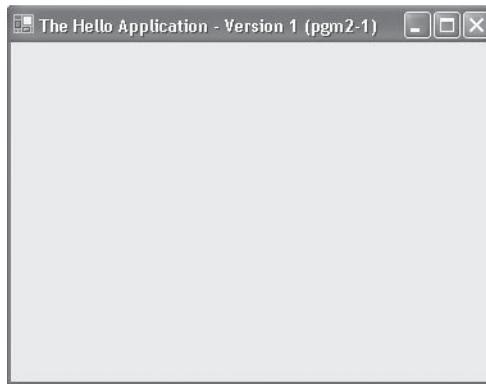


Figure 2-16 The Form as a Window When the Application is Run

gram, we can manipulate the window using standard window techniques. For example, you can click the Maximize or Minimize buttons, move or resize the window, and close the application by double-clicking the Close (X) button.

A useful feature of Visual Basic is that you can run your program at any point within its development process. This permits you to check both the look of the graphical user interface and the operation of any code that you write while the program is being developed, rather than at the end of the design process. As you write more involved programs, it is a good idea to get into the habit of checking program features as they are added by running the program.

To clearly distinguish between when a program is being developed and when it is being executed, Visual Basic uses the terms *design time* and *run time*. *Design time* is defined as the time when a Visual Basic application is being developed. During design time, objects are placed on a form, their initial properties are set, and program code is written. *Run time* is defined as the time a program is running. During run time, each form becomes a window, and the windows and controls respond to events, such as a mouse-click, by invoking the appropriate procedural code. Run time can be initiated directly from design time by pressing the F5 function key (or any of the other methods listed in the accompanying Programmer Notes on Running an Application). Although in this section we have changed object properties in design time, we will see in Section 2.4 that an object's properties can also be changed at run time.

Saving and Recalling a Project

In the next section, we will add three Button objects and one Text box to our form. Then, in Section 2.4, we will complete our application by adding program code. Before doing so, let's make sure that you can save and then retrieve the work we have completed so far.

Programmer Notes

Running an Application

While creating a Visual Basic application, you can run the application at any time using one of the following procedures:

1. Select the **D**ebug Menu and select **S**tart.
2. Use the hot key sequence Alt+D, then press the S key (Alt+D / S).
3. Press the F5 function key.

Unlike our current program which consists of a single form, a program can consist of many forms, additional code modules containing program code, and third-party supplied objects. A *form* contains the data for a single Form object, information for each object placed on the form (in this case there are none), all event code related to these objects, and any general code related to the form as a whole. A *code module* contains procedural code (no objects) that will be shared between two or more forms. It is for this reason that a separate project file, with its own name, is used. The project file keeps track of all forms, and any additional code and object modules.

To save an application, first click the **F**ile menu and then click **S**ave **A**ll. At this point all the forms, code modules and ancillary files will be saved in a folder. The name of the folder will be the project name. You can also click the SaveAll icon in the Standard Toolbar (see Figure 2-17). It is recommended that you save your solution often to prevent accidental loss of work.

To retrieve a project, select **O**pen **S**olution from the **F**ile menu, at which point an Open Solution dialog box similar to the one shown in Figure 2-17 is displayed. Select the folder with the correct solution name. A second file dialog box will appear. Within

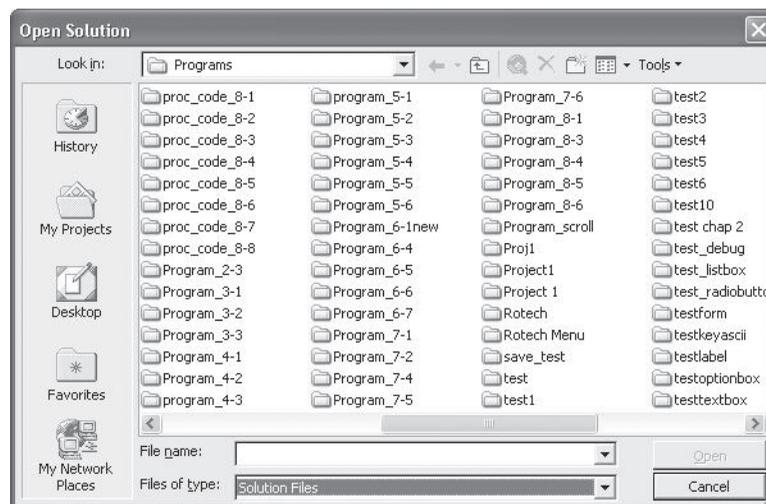


Figure 2-17 The Open Solution Dialog Box



Figure 2-18 Visual Basic's Standard Toolbar

this dialog box a file with the project name and a file type of Visual Studio Solution will appear. After this file is selected, the forms that comprise your project will reappear.

Using the Toolbar

Once you have become comfortable with the menu bar items and see how they operate and interconnect, you should take a closer look at the Standard Toolbar. For the most commonly used features of Visual Basic, such as opening a solution file, saving a solution, and running or stopping an application, click on the appropriate toolbar icon to perform the desired operation. Figure 2-18 illustrates the Standard Toolbar and identifies the icons that you will use as you progress in designing Visual Basic applications. To make sure the Standard Toolbar is visible, select the **T**oolbar item from the **V**iew menu. When this item is selected, a menu listing the available toolbars is displayed. Make sure that a check mark (v) appears to the left of the Standard item. The most useful Standard Toolbar buttons are represented by the Save All, Start, and Stop Debugging icons.

Exercises 2.2

1. Describe the difference between design time and run time.
2.
 - a. Name the three windows that should be visible during an application's design.
 - b. What are the steps for opening each of the windows listed in your answer to Exercise 2a?
 - c. In addition to the three basic design windows, what two additional windows may also be visible on the design screen?
3. What two **Form** properties should be changed for every application?
4. What does a form become during run time?
5. List the steps for creating a Visual Basic application.
6. Determine the number of properties that a **Form** object has. (*Hint*: Activate a form's property window and count the properties.)
7.
 - a. Design a Visual Basic application that consists of a single form with the heading *Test Form*. The form should not have a minimize button nor a maximize button, but should contain a close control button. (*Hint*: Locate these properties in the Properties window and change their values from **True** to **False**.)
 - b. Run the application you designed in Exercise 7a.
8. By looking at the screen, how can you tell the difference between design time and run time?

2.3 Adding an Event Procedure

In the previous section, we completed the first two steps required in constructing a Visual Basic application:

1. Create the GUI.
2. Set initial object properties.

Now we will finish the application by completing the third step:

3. Adding procedural code.

At this point our simple application, `pgm2-1`, produces a blank window when it is executed. If you then click anywhere on the window, nothing happens. This is because no event procedures have been included for the form. We will complete the application by providing a mouse click event procedure that displays a message whenever the application is running and the mouse is clicked anywhere on the application's window.

In a well-designed program, each procedure will consist of a set of instructions necessary to complete a well-defined task. Although a procedure can be initiated in a variety of ways, a procedure that is executed (called into action, or *invoked*) when an event occurs is referred to as an *event procedure* or *event handler*. The general structure of an event procedure is illustrated in Figure 2-19.

The first line of a procedure is always a header line. A *header line* begins with the optional keyword `Private`³ and must contain the keyword `Sub` (which derives from the word Subprogram), the name of the procedure, and a set of parentheses. For event procedures, the name consists of an object identification, an optional underscore character (`_`), a valid event for the object, the parameters in parentheses, the keyword `Handles` followed by the object identification, an underscore character, and a valid event. If the object is the form itself, the object name `Form` is used. For example, the header line

```
Private Sub Form1_Click(ByVal sender As Object, ByVal e As _
    System.EventArgs) Handles MyBase.Click
```

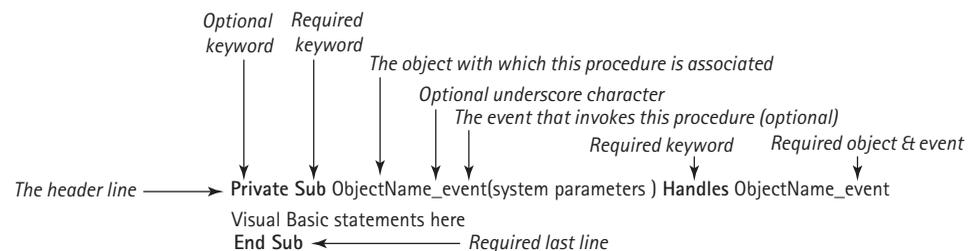


Figure 2-19 The Structure of an Event Procedure

³The significance of the keyword `Private` is explained in Chapter 7.

denotes an event procedure that will be activated when the mouse is clicked on the form. The values between the parentheses are used for transmitting data to and from the procedure when it is invoked. Data transmitted in this fashion are referred to as *arguments* of the procedure. Note that for forms, the object identification after the **Handles** keyword is `MyBase`, while for controls on the form the object identification is the name of the control (e.g., `txtBox1`). The last line of each procedure consists of the keywords **End Sub**. Finally, all statements from the header line up to and including the terminating **End Sub** statement are collectively referred to as the procedure's *body*.

For a form's mouse click event, the required procedure's structure is:

```
Private Sub Form1_Click(ByVal sender As Object, ByVal e As _
    System.EventArgs) Handles MyBase.Click
    ' Visual Basic statements in here
End Sub
```

The first and last lines of a procedure, consisting of the header line and terminating body line **End Sub**, are referred to as the procedure's *template*. Note that if the form is named `frmMain`, the procedure is named `frmMain_Click`. As shown in Figure 2-20, event procedure templates need not be manually typed because they are automatically provided in Visual Basic's Code window.

Before activating the Code window, we need to decide what Visual Basic statements will be included in the body of our event procedure. In this section, we present an easy way for displaying an output message—the `MessageBox.Show` method.

The `MessageBox.Show` Method⁴

Visual Basic provides a number of built-in methods in addition to methods for constructing event procedures. The `MessageBox.Show` method is used to display a box with a user-supplied message inside. The message box also contains a title and an icon. For example, the boxes illustrated in the next several figures all were created using the

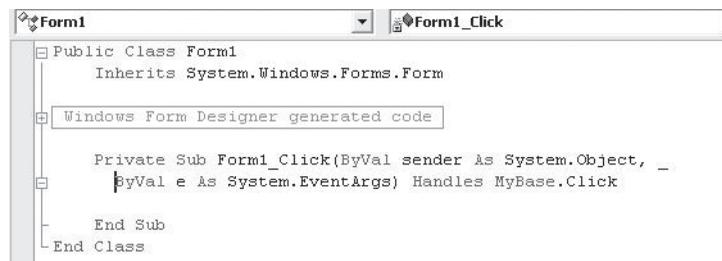


Figure 2-20 The Code Window, Showing a Click Event Procedure Template

⁴In previous versions of Visual Basic, the `MsgBox` function was used to display a message box. Although this function may still be used, the `MessageBox` class replaces this function with `MessageBox.Show` as the method used to display a message box.

MessageBox.Show method. The **MessageBox.Show** method has the following general formats:

```
MessageBox.Show(text)
MessageBox.Show(text, caption)
MessageBox.Show(text, caption, buttons)
MessageBox.Show(text, caption, buttons, icon)
MessageBox.Show(text, caption, buttons, icon, defaultbutton)
```

Messages, such as those displayed in message boxes, are called *strings* in Visual Basic. A *string* consists of a group of characters made up of letters, numbers, and special characters, such as the exclamation point. The beginning and end of a string of characters are marked by double quotes (“string in here”).⁵ The argument *text* appears in the message box window and it may be a literal string enclosed in quotes or a string variable. The string *caption* is displayed in the message box’s title bar. If *caption* is not specified, as in the preceding form, the title bar is empty. Figure 2-21 shows the message box displayed as a result of the following statement:

```
MessageBox.Show("This is the text")
```

This is the simplest form of the message box. Figure 2-22 displays a message from the following statement, which includes a title (caption):

```
MessageBox.Show("This is the text", "This is the caption")
```

The word *buttons* specifies the types of buttons that are displayed. The value for *buttons* can be one of the following:

```
MessageBoxButtons.AbortRetryIgnore
MessageBoxButtons.OK
MessageBoxButtons.OKCancel
MessageBoxButtons.RetryCancel
MessageBoxButtons.YesNo
MessageBoxButtons.YesNoCancel
```

If the value is **MessageBoxButtons.AbortRetryIgnore**, then the Abort, Retry, and Ignore buttons are all displayed in the message box. The string following the period indicates which buttons are displayed (e.g., Yes and No for **YesNo**). If this argument is not specified, the OK button is displayed as shown in the previous two figures. The following statement displays a message box with two buttons, as shown in Figure 2-23.

```
MessageBox.Show("Are you sure you want to delete the record?", _
    "Customer Records", MessageBoxButtons.YesNo)
```

⁵Strings are discussed in detail in Chapter 3.



Figure 2-21 A Simple Message Box



Figure 2-22 A Message Box with Title



Figure 2-23 A Message Box with Title and Yes/No Buttons

Table 2-4 MessageBox.Show Icons

Values for Icon	Icon
MessageBoxIcon.Asterisk	The Letter i
MessageBoxIcon.Information	The Letter i
MessageBoxIcon.Error	The Letter X
MessageBoxIcon.Hand	The Letter X
MessageBoxIcon.Stop	The Letter X
MessageBoxIcon.Exclamation	Exclamation Point
MessageBoxIcon.Warning	Exclamation Point
MessageBoxIcon.Question	Question Mark

Table 2-4 lists the values for the *icon* argument and shows what the icons look like. Note that some values display the same icon as other values. An icon is displayed to the left of the message.

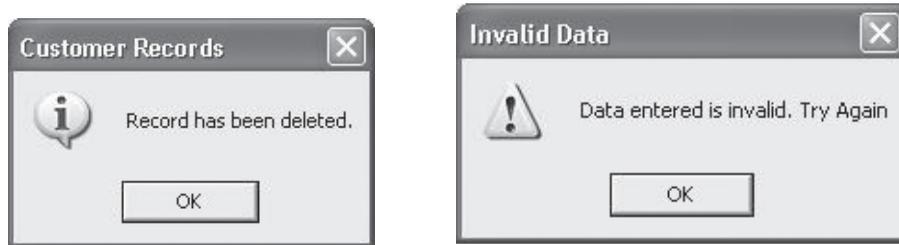
For example, the statements:

```
MessageBox.Show("Record has been deleted.", "Customer Records", _
    MessageBoxButtons.OK, MessageBoxIcon.Information)
```

```
MessageBox.Show("Data entered is invalid. Try Again", "Invalid Data", _
    MessageBoxButtons.OK, MessageBoxIcon.Warning)
```

produced the message boxes shown in Figures 2-24 and 2-25. These message boxes include an information icon.

The information icons (Asterisk and Information) should be used when you are displaying an information message box with only an OK button. The stop icons (Error, Hand, and Stop) should be used when the message displayed is indicating a serious



Figures 2–24 and 2–25 Message Boxes with Title, OK Button, and Information Icon

problem that needs to be corrected before the program can continue. The exclamation icons (Exclamation and Warning) should be used when the user must make a decision before the program can continue; this would not be used with only an OK button. The question icon can be used when a question needs to be answered.

The last optional argument to the `MessageBox.Show` method, as shown in the fifth format above, is the `defaultbutton`. This argument specifies which button to select as the default button, as there may be three buttons displayed in the message box (e.g., Yes, No, Cancel). The default button is the button that has focus when the message box is displayed, and is the button that is clicked when the user presses the Enter key. If this argument is not specified, the default is that the leftmost button is the default. The values for this argument are:

```
MessageBoxDefaultButton.Button1
MessageBoxDefaultButton.Button2
MessageBoxDefaultButton.Button3
```

where `MessageBoxDefaultButton.Button1` specifies the leftmost button (and is the default), `MessageBoxDefaultButton.Button2` specifies the second button from the left, and `MessageBoxDefaultButton.Button3` specifies the third button from the left.

After the user clicks on a button in the message box, the message box is closed. The return value of the call to the `MessageBox.Show` method indicates which button the user clicked. This is useful in code to determine what action to take. The return values may be one of the following:

```
DialogResult.Abort
DialogResult.Cancel
DialogResult.Ignore
DialogResult.No
DialogResult.OK
DialogResult.Retry
DialogResult.Yes
```

The following code gives an example of how the return value may be used:

```
Dim msgresult as Integer

msgresult = MessageBox.Show("Press OK to Delete", "Confirm Delete", _
    MessageBoxButtons.OKCancel, MessageBoxIcon.Stop, _
    MessageBoxDefaultButton.Button2)

If msgresult = DialogResult.OK Then
    . . .
```

Figure 2-26 shows the message box that is displayed by this call to `MessageBox.Show`. It is important to note that, in this example, the second button, Cancel, is the default button. The user has to move focus to the OK button to confirm the deletion. If the OK button is clicked, the return value is `DialogResult.OK`.

The message boxes shown are all special cases of a more general type of box referred to as a dialog box. A *dialog box* is any box that appears which requires the user to supply additional information to complete a task. In the case of the message boxes illustrated in this section, the required additional information is simply that the user must either click the OK box or press the Enter key to permit the application to continue.

Now let's include a `MessageBox.Show` method on our form so that the statement will be executed when the mouse is clicked. The required procedure is:

```
Private Sub Form1_Click(ByVal sender As Object, ByVal e As _
    System.EventArgs) Handles MyBase.Click
    MessageBox.Show("Hello World!", "Sample")
End Sub
```

To enter this code, first make sure that you are in design mode and have a form named `Form1` showing on the screen, as illustrated in Figure 2-27.

To open the Code window, do any one of the following:

- If the Code window is visible, click on it.

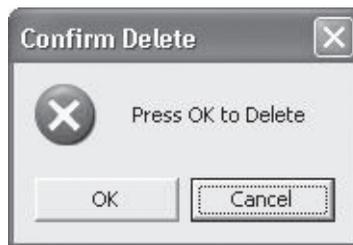


Figure 2-26 A Message Box with Default Button Argument Changed

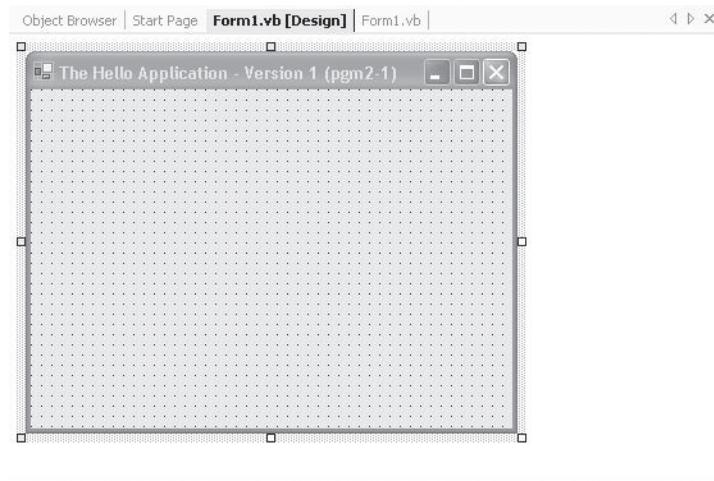


Figure 2-27 The Form1 Form in Design Time

- Double-click anywhere on the Form window.
- Select the Code option from the View menu.
- Press the F7 method key anywhere on the design form.
- Select the View Code icon from the Project Window.

Any of these actions will open the Code window shown in Figure 2-28.

The class drop-down should display `Form1` and the method drop-down should display `Form1_Load`. This indicates that the current class is `Form1` and the method is `Load`, which in this case is an event. Note that a code stub (template) for the `Form1_Load` procedure is auto-

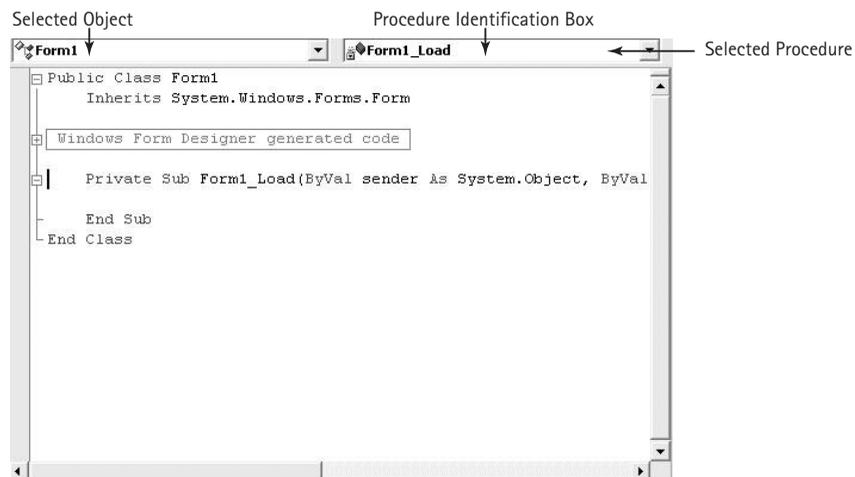


Figure 2-28 The Code Window

matically supplied within the Code window. Note, too, that the Code window displays all procedures and declarations that have been written, with each procedure separated by a line. When the Code window is not large enough to display either all procedures or even all of a single procedure, the scroll bars can be used to bring sections of code within the visible window area.

When you have the Code window shown in Figure 2-28 visible, click the down arrowhead (▼) to the right of the selected class box. Then select *(Base Class Events)*. Click the down-facing arrowhead in the method box. This produces the window shown in Figure 2-29. Here the drop-down list can be scrolled to provide all of the events associated with the selected object.

To select the Click procedure, do either of the following:

- Using the list's scroll bar, locate the word **Click**, and then click on this keyword.
- Using the up-arrow cursor key, highlight the word **Click**, and press Enter.

Either of these actions will add the following lines to the Code window:

```
Private Sub Form1_Click(ByVal sender As Object, ByVal e As _
    System.EventArgs) Handles MyBase.Click
```

```
End Sub
```

In your code window, the first statement beginning with `Private` fits on one line. However, this line will not fit on a page of this book. To continue a Visual Basic statement on the next line, type a space followed by the underscore symbol (“_”).

You are now ready to add code to the Form1 Click event. Type the line,

```
MessageBox.Show("Hello World!", "Sample")
```

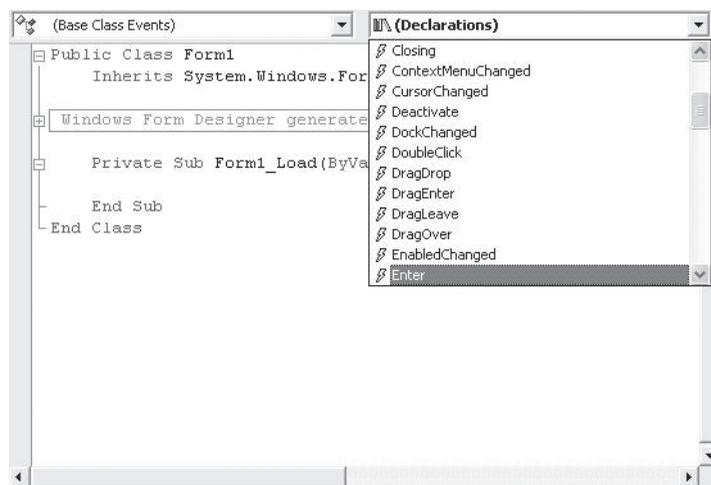


Figure 2-29 The List of Events Associated With a Form

between the header line `Private Sub Form1_Click (parameters)` and the terminating line `End Sub`. When this task is completed, the procedure should appear as shown below:

```
Private Sub Form1_Click(ByVal sender As Object, ByVal e As _  
    System.EventArgs) Handles MyBase.Click  
    MessageBox.Show("Hello World!", "Sample")  
End Sub
```

Note that we have indented the single Visual Basic statement using three spaces. Although this is not required, indentation is a sign of good programming practice. Here it permits the statements within the procedure to be easily identified.

Our event procedure is now complete and you can close the Code window. When you run the program, the application should appear as shown in Figure 2-30. Clicking anywhere on the window will create the window shown in Figure 2-31. To remove the

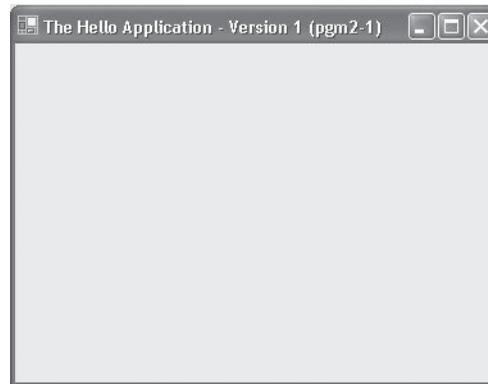


Figure 2-30 The Initial Run Time Application Window

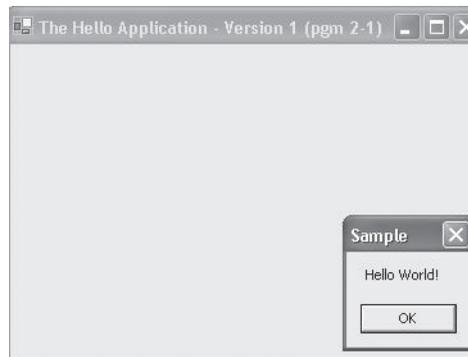


Figure 2-31 The Effect of the Mouse Click Event

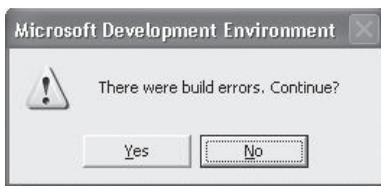


Figure 2-32a Notification of an Error

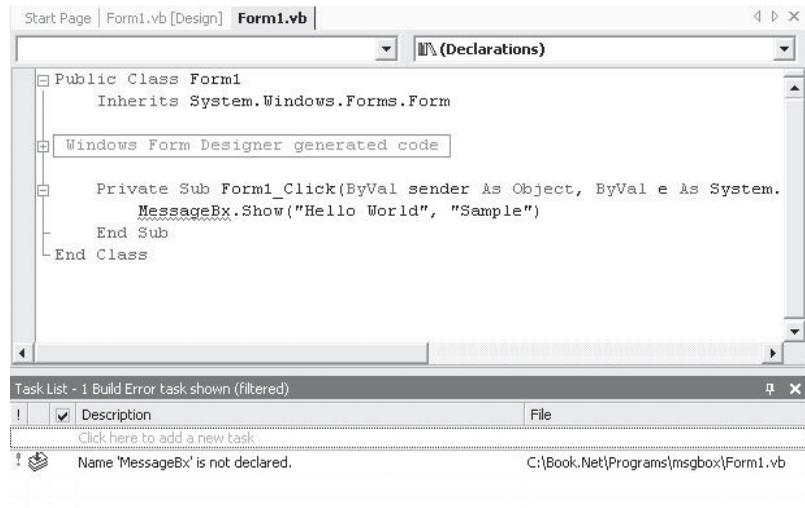


Figure 2-32b Identification of the Invalid Statement and its Procedure

message box, press either the escape (Esc) or Enter key, or click the OK Button. After performing this test, save the application using the Save All option from the File menu.

Correcting Errors

If you incorrectly typed the message box statement in your procedure, when the program is run the code box containing this procedure would automatically be displayed with the highlight placed on the incorrect statement. For example, if you inadvertently spelled `MessageBox` as `MessageBx`, the window shown in Figure 2-32a would appear. If you then clicked No in the message box, Figure 2-32b would appear. Below the Code window, a new window would appear listing all the errors detected. In this case only one error was found: Visual Basic could not interpret `MessageBx`. If you double-clicked `MessageBx` in the Task List window, Visual Basic would position the cursor in the Code window beside the statement in question. Note that even before you tried running the program, Visual Basic would underline, with a blue saw-toothed line, parts of statements that it could not interpret. Once you corrected the error, you could re-run the program.

Programmer Notes

Code Editor Options

The Visual Basic Editor provides a number of options that are useful when you are entering code into the Code window. These include the following:

Color Coded Instructions

The Visual Basic Editor displays procedural code in a variety of user-selected colors. By default, the following color selections are used:

- Keywords—Blue
- Comments—Green
- Errors—Blue, saw-toothed underline
- Other Text—Black

Completing a Word

Once you have entered enough characters for Visual Basic to identify a work, you can have the Editor complete the word.

Quick Syntax Information:

If you are trying to complete a statement, such as a `MessageBox.Show` statement, and forget the required syntax, you can ask the editor to provide it. You activate this option by placing the insertion cursor (the vertical insert line) over the piece of code in question.

Exercises 2.3

1. Define the following terms:
 - a. event-procedure
 - b. dialog box
 - c. method
 - d. header line
 - e. argument
 - f. template
2. a. What window do you use to enter the code for an event procedure?
b. List two ways of activating the window you listed as the answer for Exercise 2a.
3. Using the Code window, determine how many event procedures are associated with a form.
4. Design and run the application presented in this section using the `MessageBox.Show` method in the form's click event procedure.

2.4 Adding Controls

Although the application presented in the previous section is useful in introducing us to the basic design-time windows needed for developing Visual Basic applications, it is not a very useful application in itself. To make it useful, we will have to add additional

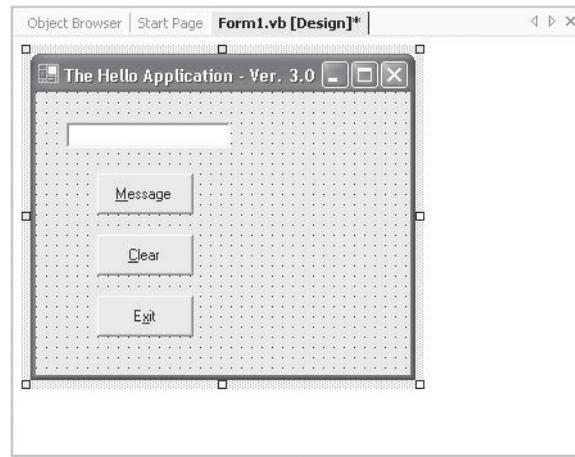


Figure 2-33 Program 2-3's Interface

objects and event procedures to the form. Adding objects to the form creates the final graphical interface that the user will see and interact with when the program is run. Adding event procedures to the objects then brings them “to life,” so that when they are selected something actually happens. In this section, we present the basic method for placing objects on a form, and in the next section we will attach specific event procedures to these objects.

Objects selected from the Toolbox and placed on a form are referred to as *controls*. Placing objects on a Form is quite simple, and the same method is used for all objects.

The simplest procedure is to double-click the desired Toolbox object. This causes an object of the selected type to be automatically placed on the Form. Once this is done you can change its size or position, and set any additional properties such as its name, text, or color. These latter properties are modified from within the Properties window or in the Design window, and determine how the object appears when it is first displayed during run time.

By far the most commonly used Toolbox objects are the Button, TextBox, and Label. For our second application, we will use the first two of these object types—the Button and TextBox—to create the design-time interface shown in Figure 2-33.

Adding a Button

To place a button on the form, double-click the Button icon. Double-clicking this icon causes a button with eight small squares, referred to as *sizing handles*, to be placed on the form, as shown in Figure 2-34. The fact that the sizing handles are visible indicates that the object is *active*, which means that it can be moved, resized, and have its other

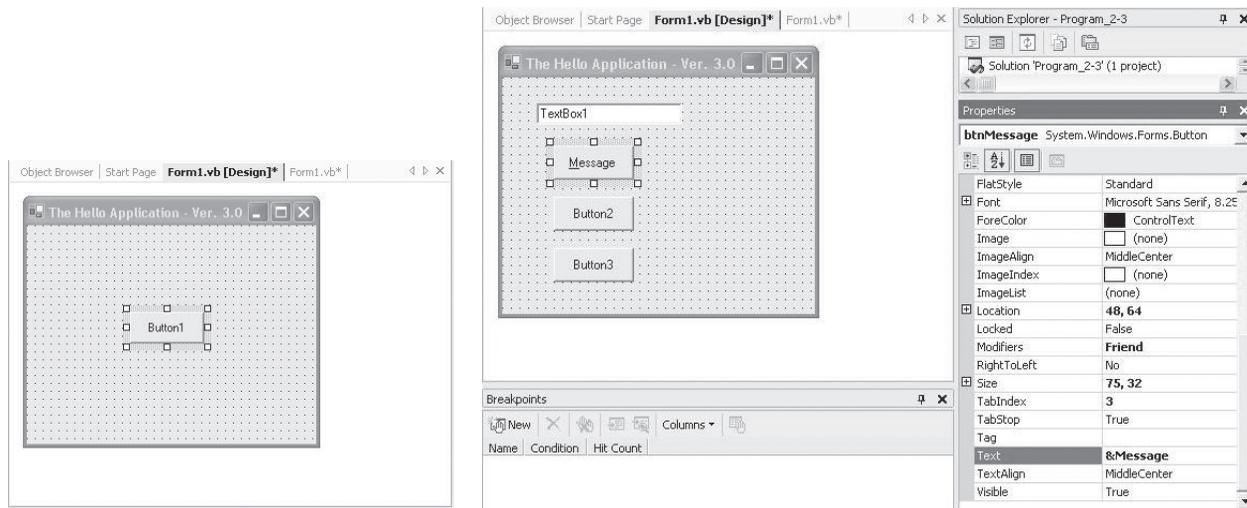


Figure 2-34 The First Button Placed on the Form

properties changed. To deactivate the currently active object, use the mouse to click anywhere outside of it. Clicking on another object will activate the other object, while clicking on an area of the form where no object is located activates the Form object itself.

The active object, which should now be the button just placed on the form, can be moved by placing the mouse pointer anywhere inside the object (but not on the sizing handles), holding down the mouse's left button, and dragging the object to its desired new position. Do this now and place this first button in the position of the Message button shown in Figure 2-33. Your form should now look like the one shown in Figure 2-35.

Once you have successfully placed the first button on the form, either use the same procedure to place two more buttons in the positions shown in Figure 2-36, or use the alternative procedure given in the Programmer Notes box on page 82. Included in this box are additional procedures for resizing, moving, and deleting an object. Controls do not have to line up perfectly, but should be placed neatly on the form.

Adding a TextBox Control

Text boxes can be used for both entering data and displaying results. In our current application, we will use a text box for output by displaying a message when one of the form's buttons is clicked.

To place a TextBox object on a form, double-click the TextBox icon, as we did when we placed the three-button object. If you happen to double-click the wrong icon, simply activate it and press the Delete key to remove it. Once you have placed a text box on the form, move and resize it so that it appears as shown in Figure 2-37.

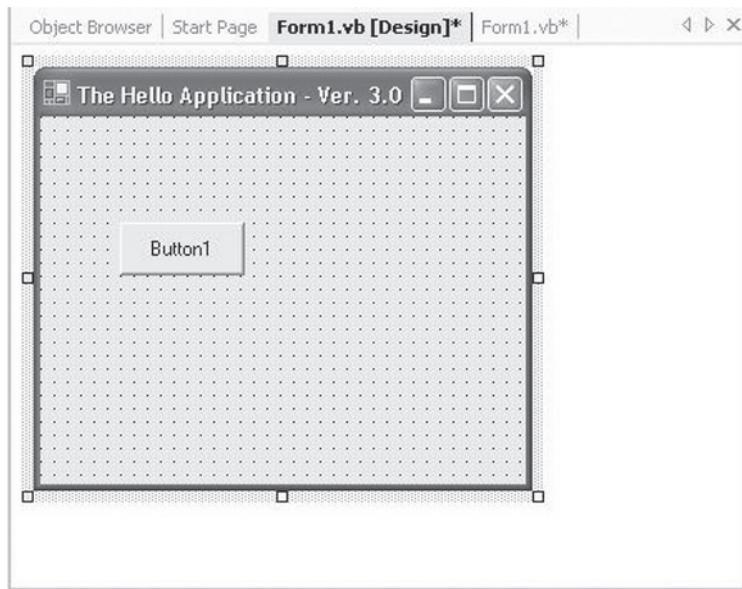


Figure 2–35 The Final Placement of the First Button

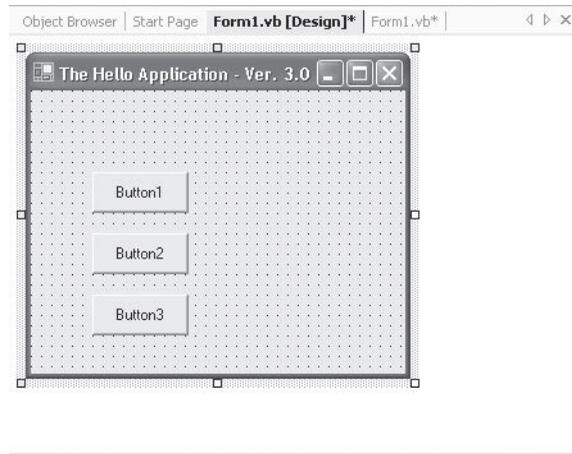


Figure 2–36 Placement of Three Buttons on the Form

Programmer Notes

Creating and Deleting Objects

To Add an Object:

Either:

Double-click the desired object in the toolbox. Doing this places a presized object on the form.

Or:

Click the desired object in the Toolbox and then move the mouse pointer onto the form. When the mouse pointer moves onto the form, it will change to a crosshair cursor. Hold the left mouse button down when the crosshairs are correctly positioned for any corner of the object and drag the mouse diagonally away from this corner, in any direction, to generate the opposite corner. When the object is the desired size, release the left mouse button.

To Resize an Object:

Activate the object by clicking inside it. Place the mouse pointer on one of the sizing handles, which will cause the mouse pointer to change to a double-sided arrow, <=>. Hold down the left mouse button and move the mouse in the direction of either arrowhead. Release the mouse button when the desired size is reached. You can also hold down the Shift key while pressing any of the four arrow keys to resize the object. Pressing the up and down arrow keys will decrease and increase the height; pressing the right and left arrow keys will increase and decrease the width.

To Move an Object:

Whether the object is active or not, place the mouse pointer inside the object and hold down the left mouse button. Drag the object to the desired position and then release the mouse button. You can also press one of the four arrow keys to move the object in the direction of the arrow. Holding down the Control key while pressing an arrow key will move the object in smaller increments.

To Delete an Object:

Activate the object by clicking inside it, and then press the Delete key.

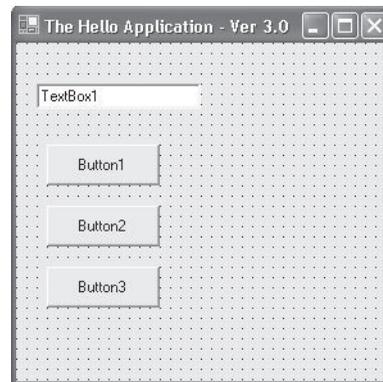


Figure 2-37 Placement of the TextBox

Table 2-5 Program 2-3's Initial Properties Table

Object	Property	Setting
Form	Name	frmMain ⁶
	Text	The Hello Application—Ver. 3.0
Button	Name	btnMessage
	Text	&Message
Button	Name	btnClear
	Text	&Clear
Button	Name	btnExit
	Text	E&xit
TextBox	Name	txtDisplay
	Text	(blank)

Setting the Initial Object Properties

At this point we have assembled all of the Form controls that are required for our application. We still need to change the default names of these objects and set the Text properties of the buttons to those previously shown in Figure 2-33. After that, we can add the code so that each button performs its designated task when it is clicked. Let's now change the initial properties of our four objects to make them appear as shown in Figure 2-33. Table 2-5 lists the desired property settings for each object, including the Form object.

Programmer Notes

Changing the Name of a Form

When executing an application, Visual Basic needs to know the name of the form with which to start the application. So far we have only created small programs with one form. In subsequent chapters we will build applications with multiple forms.

By default, Visual Basic assumes that the name of the first form of the application is Form1. If we change the name of the form in the Property Window, even in a one-form application, Visual Basic will generate the following error message when we try to run the program:

'Sub Main' was not found in 'Project_1.Form1'.

To fix this error, there are two options.

1. Double-click the above error. This will cause the dialog box shown in Figure 2-38 to appear, prompting us to confirm that frmMain is the first form to be executed. Double

⁶Be sure to read the Programmer Notes on changing Form names. If the actions in the Note are not taken, the program may never run.

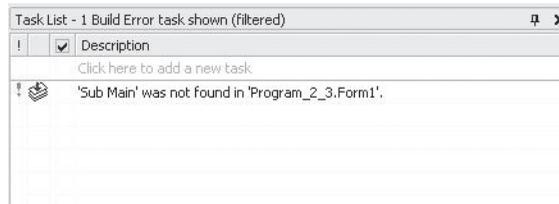


Figure 2-38 Error Dialog Caused by New Form Name

click Project_1.frmMain and then click OK. This will only have to be done once because the specification that frmMain is the initial form will be saved with other project information.

2. In the Solution Explorer, highlight Project 1, right-click it, and then choose Properties. The form shown in Figure 2-39 will appear. From the Startup Object, select the new form name, in this case frmMain.

Before we set the properties listed in Table 2-5, two comments are in order. The first concerns the ampersand (&) symbol that is included in the Text property of all of the buttons. This symbol should be typed exactly as shown. Its visual effect is to cause the character immediately following it to be underlined. Its operational effect is to create an accelerator key. An *accelerator key*, which is also referred to as a hot key sequence (or hot key, for short), is simply a keyboard shortcut for a user to make a selection. When used with a button it permits the user to activate the button by simultaneously pressing the Alt key and the underlined letter key, rather than either clicking with the mouse or activating the button by first selecting it and then pressing the Enter key.

The second comment concerns the Text property for a text box, shown in the last line in Table 2-5. For a text box, the Text setting determines what text will be displayed in the text box. As shown in Figure 2-37, the initial data shown in the text box is

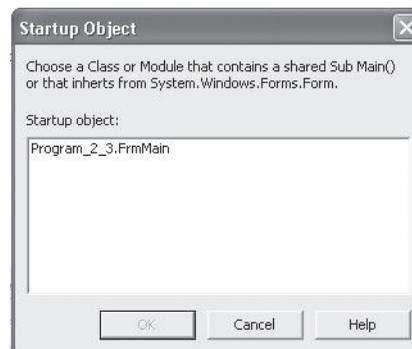


Figure 2-39 Dialog Box to Change the Startup Form Name

TextBox1, which is the default value for this property. The term (blank) means that we will set this value to a blank.

Also note that, although we are setting the initial properties for all of the objects at the same time, this is not necessary. We are doing so here to show the various setting methods in one place. In practice, we could just as easily have set each object's properties immediately after it was placed on the form.

Recall from the Programmer Notes box on page 59 that a Properties window can be activated in a variety of ways: by pressing the F4 function key or by selecting Properties from the Window menu (which can also be obtained by the hot key sequence ALT+V, followed by S). Now, however, we have five objects on the design screen: the form, three buttons, and a text box. To select the properties for a particular object, you can use any of the options listed in the Programmer Notes box on page 59.

The simplest method is to first activate the desired object by clicking it, then press the F4 function key, and then scroll to the desired property. Because an object is automatically activated just after it is placed on a form, this method is particularly useful for immediately changing the object's properties. This sequence of adding an object and immediately changing its properties is the preferred sequence for many programmers.

An alternative method is to open the Properties window for the currently active object, no matter what it is, and then click on the downward-facing arrowhead key (▼) to the right of the object's name (see Figure 2–40). The pull-down list that appears contains the names of all objects associated with the form. Clicking the desired object name

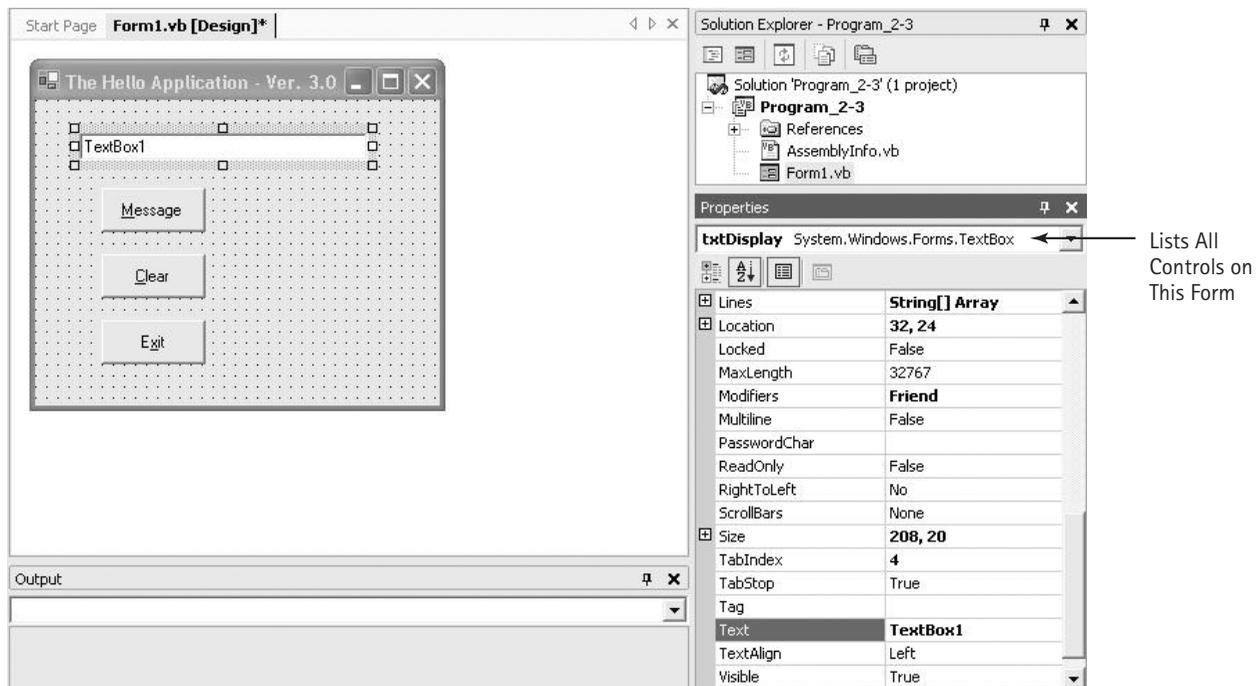


Figure 2–40 Changing the Properties Settings

in the list both activates the desired object and opens its Properties window. This method is particularly useful when changing the properties of a group of objects by sequencing through them after all objects have been placed on the form. Using either of these methods, alter the initial properties to those listed in Table 2-5.

At this stage, you should see the design screen shown in Figure 2-40. Within the context of developing a complete program, we have achieved the first two steps in our three-step process. They are:

1. Create the GUI.
2. Set the properties of each object on the interface.

We will complete the third and final step of writing the code in the next section. However, before doing so, run the application by pressing the F5 function key. Despite the fact that clicking on any of the buttons produces no effect (because we have not yet attached any code to these buttons), we can use the application to introduce two important concepts connected with any form: focus and tab sequence.

Programmer Notes

Activating the Properties Window for a Specific Object

1. Activate the object by clicking it, and then press the F4 function key.
2. Activate the Properties window for the currently selected object or form, whatever it may be, by either pressing the F4 key or selecting the Properties option from the Windows menu (Alt+V / S). Then change to the desired object from within the Properties window by clicking the underlined down arrow to the right of the object's name and then selecting the desired object from the pull-down list.

Looking at the Focus and Tab Sequence

When an application is run and a user is looking at the form, only one of the form's controls will have *input focus*, or focus, for short. The control with focus is the object that will be affected by pressing a key or clicking the mouse. For example, when a button has the focus, its caption will be surrounded by a dotted rectangle, as shown in Figure 2-41. Similarly, when a text box has the focus, a solid cursor appears, indicating that the user can type in data.

An object can only receive focus if it is capable of responding to user input through either the keyboard or mouse. As a result, controls such as labels can never receive the focus. In order to receive the focus a control must have its **Enabled**, **Visible**, and **TabStop** properties set to **True**. As the default settings for all three properties are **True**, they do not usually have to be checked for normal tab operation. By setting a control's **Enabled** property to **True**, you permit it to respond to user-generated events, such as pressing a key or clicking a mouse. The **Visible** property determines whether an object will actually be visible on the form during run time (it is always available for view during design time). A **True TabStop** setting forces a tab stop for the object, while a **False**



Figure 2-41 A Button With and Without Focus

value causes the object to be skipped over in the tab stop sequence. A control capable of receiving focus, such as a button, can receive the focus in one of three ways:

1. A user clicks the object.
2. A user presses the tab key until the object receives the focus.
3. The code activates the focus.

To see how the first method operates, press the F5 function key to execute the Hello Application (Program 2-3). Once the program is executing, click on any of the form objects. As you do, notice how the focus shifts. Now, press the tab key a few times and see how the focus shifts from control to control. The sequence in which the focus shifts from control to control as the tab key is pressed is called the *tab sequence*. This sequence is initially determined by the order in which controls are placed on the form. For example, assume you first created buttons named `btnCom1`, `btnCom2`, and `btnCom3`, respectively, and then created a text box named `txtText1`. When the application is run, the `btnCom1` button will have the focus. As you press the tab key, focus will shift to the `btnCom2` button, then to the `btnCom3` button, and finally to the text box. Thus, the tab sequence is `btnCom1` to `btnCom2` to `btnCom3` to `txtText1`. (This assumes that each control has its **Enabled**, **Visible**, and **TabStop** properties all set to True.)

You can alter the default tab order obtained as a result of placing controls on the form by modifying an object's **TabIndex** value. Initially, the first control placed on a form is assigned a **TabIndex** value of 0, the second object is assigned a **TabIndex** value of 1, and so on. To change the tab order, you have to change an object's **TabIndex** value and Visual Basic will renumber the remaining objects in a logical order. For example, if you have six objects on the form with **TabIndex** values from 0 to 5, and you change the object with value 3 to a value of 0, the objects with initial values of 0, 1, and 2 will have their values automatically changed to 1, 2, and 3, respectively. Similarly, if you change the object with a **TabIndex** value of 2 to 5, the objects with initial values of 3, 4, and 5 will have their values automatically reduced by one. Thus, the sequence from one object to another remains the same for all objects, except for the insertion or deletion of the altered object. However, if you become confused, simply reset the complete sequence in the desired order by manually starting with a **TabIndex** value of 0 and then assigning values in the desired order. A control whose **TabStop** property has been set to

False maintains its `TabIndex` value, but is simply skipped over for the next object in the tab sequence. Be sure to check all the `TabIndex` values after any changes to the form have been made. In Chapter 3, we will describe another way to set the `TabIndex` values.

The Format Menu Option⁷

The `Format` menu option provides the ability to align and move selected controls as a unit, as well as lock controls and make selected controls the same size. This is a great help in constructing a consistent look on a form that contains numerous controls. In this section, we will see how this menu option is used.

As a specific example using the `Format` menu, consider Figure 2-42, showing two buttons on a design form. To align both controls and make them the same size, first you must select the desired controls. This can be done by clicking the form and dragging the resulting dotted line to enclose all of the controls that you wish to format, as illustrated in Figure 2-42, or by holding the `Shift` key down and clicking the desired controls.

Once you have selected the desired controls for formatting, the last selected object will appear with solid grab handles. For example, in Figure 2-43 it is `Button2`. The solid grab handles designate the control that is the *defining control*, setting the pattern for both sizing and aligning the other selected controls. If this control is not the defining control that you want, select another by clicking it.

Having selected the desired defining control, click on the `Format` menu bar and then select the desired `Format` option. For example, Figure 2-44 illustrates the selection for making all controls within the dotted lines the same size. Within this submenu, you have the further choice of making either the width, height, or both dimensions of all controls equal to the defining control's respective dimensions. The choice shown in this figure would make all selected controls equal in both width and height to the defining control. You can also use the `Layout` Toolbar, as shown in Figure 2-8, instead of using the `Align` submenu.

You can also change the size of a control by selecting the control in design mode and using the `Shift` and arrow keys. In addition to sizing controls, you may also want to

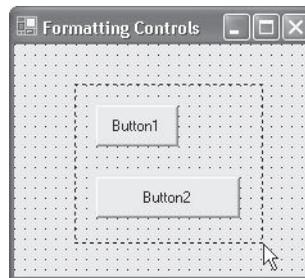


Figure 2-42 Preparing Two Controls for Formatting

⁷This topic may be omitted on first reading with no loss of subject continuity.

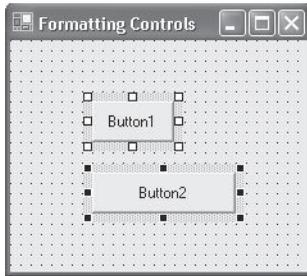


Figure 2-43 Locating the Defining Control

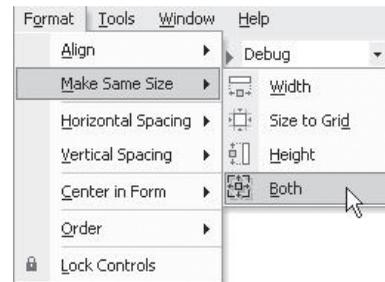


Figure 2-44 Making Controls the Same Size

align a group of controls within a form. Figure 2-45 illustrates the options that are provided for the Align submenu. As shown, controls may be aligned in seven different ways, the first six of which are aligned relative to the position of the defining control. Choosing any one of these first six options will move all other formatted controls in relation to the defining control. The position of the defining control *is not* altered.

An additional and very useful feature of the Format selection process is that all selected controls can be moved as a unit. To do this, click within one of the selected controls and drag the control. As the control is dragged, all other selected controls will move as a group while maintaining their relative positions to each other.

Finally, as shown in Figures 2-44 and 2-45, the Format submenu provides a number of other Format choices, the effects of which are obvious, except perhaps for the Lock control. This control locks all controls on the form in their current positions and prevents you from inadvertently moving them once you have placed them. Because this control works on a form-by-form basis, only controls on the currently active form are locked, and controls on other forms are unaffected.

The Label Control

When you create a GUI application, you need to make clear what the purpose of a form is and what sort of data should be entered into a text box. The Label control is used to provide the user with information. As such, labels appear as headings within a form or next to a control to let the user know the control's purpose. For example, in Figure

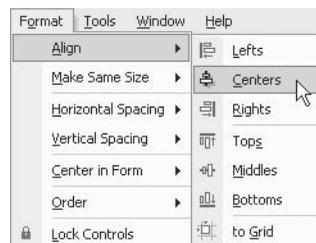


Figure 2-45 Aligning Controls to the Defining Control

Figure 2-46 A Form with Labels

2-46, the heading *Disk Order Sales Form* is a label. Additionally, the text located to the left of each text box is also a label. As the text within a button is provided by the button's Text property, buttons rarely have labels associated with them.

Creating a label is very simple; all that is required is selecting the Label icon from the Toolbox and setting both its Text and Font properties. By definition, a label is a read-only control that cannot be changed by a user directly. The text displayed by a label is determined by its Text property, which can be set at design time or at run time under program control. The Text property's value is displayed in a style using the information provided by the Font property. For example, Figure 2-47 shows the Font property's setting box as it appears when the Font property was selected for the heading used in Figure 2-46. In particular, notice that the highlighted Font property has an ellipsis box (the box with the three dots) to the right of the MS Sans Serif setting. The

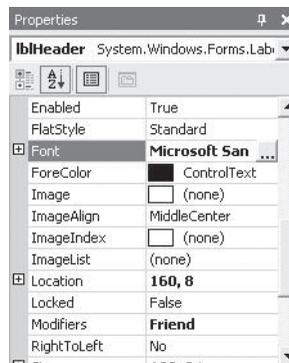


Figure 2-47 Selecting the Font Property

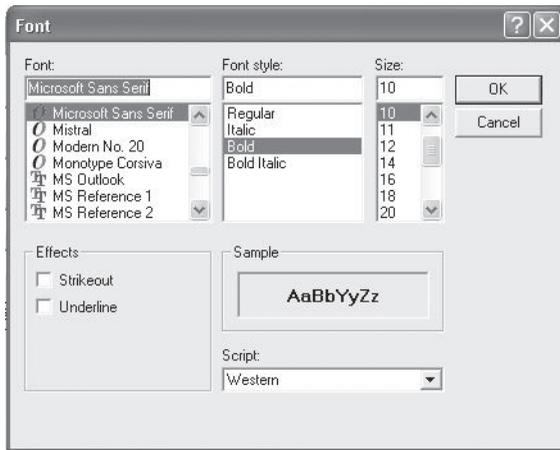


Figure 2-48a The Font Dialog Box

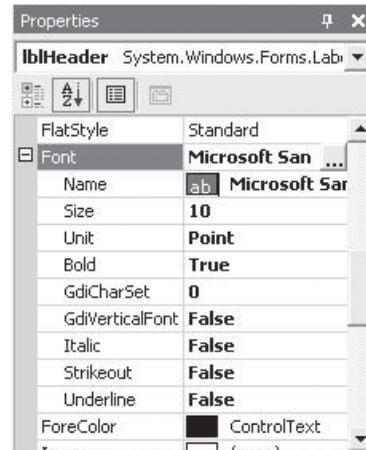


Figure 2-48b The Font Properties Expanded

ellipsis box appears when the **Font** property is selected, and clicking this box causes the **Font** dialog box shown in Figure 2-48a to appear.

In the **Font** dialog box, you set the label's font type, style, and size. The size is specified in points, and there are 72 points to an inch. Note that the label used as a heading in Figure 2-46 uses a Microsoft Sans Serif font type and is displayed in bold with a point size of 10. The type size used for all of the other labels is 8.25 points, the default. To make the form easily readable, you should try to avoid using a point size smaller than 8 points on any form. Figure 2-48b shows what is displayed in the properties window if you click on the + to the left of the **Font** property.

Although it is the label's **Text** property that is displayed on an application's interface, an additional useful property is the **AutoSize**. If the **AutoSize** property is set to **False**, you must manually adjust the physical size of the label at design time, using the sizing handles to fit the text. It is generally easier to set the **AutoSize** property to **True**. Then, as you type in the text, the label will automatically expand its width to the right to fit the text. However, you should take care that the size of the control is not too large and does not impact the other controls.

Exercises 2.4

1. Determine how many event procedures are available for a **Button** control. (*Hint:* Activate the **Code** window for a form that has a **Button** control and count the available procedures.)
2. Determine how many properties can be set for a text box control.
3. Determine if a label has a **TabStop** property.
4. What is the difference between the **Name** and **Text** properties?
5. How is a hot (accelerator) key created for a button?
6. Create a button named `btnInput` having a **Text** setting of `U`alues.

7. Create a button named `btnDisplay` having a `Text` setting of `Display`.
8. Assume that one button has a `Text` setting of `Message`, and a second button has a `Text` setting of `Display`. Determine what happens when the hot key sequence `Alt+A` is pressed twice. Do this by creating the two buttons and running the program.
9. Create a text box named `txtOne` that has a red foreground color and a blue background color. The initial text displayed in the box should be `Welcome to Visual Basic`. (*Hint: Use the `ForeColor` and `BackColor` properties—click on the ellipsis (...) to bring up the available colors.*)
10. Create a text box named `txtTwo` that has a blue foreground color and a gray background color. The initial text displayed in the box should be `High-Level Language`. (*Hint: Use the `ForeColor` and `BackColor` properties—click on the ellipsis (...) to bring up the available colors.*)
11. What are the three ways that an object can receive focus?
12. To receive focus in the tab sequence, what three properties must be set to `True`?
13.
 - a. Create a graphical user interface that contains two buttons and two text boxes. The names of these controls should be `btnOne`, `btnTwo`, `txtFirst`, and `txtSecond`. Set the tab sequence so that tab control goes from `txtFirst` to `txtSecond` to `btnOne` to `btnTwo`.
 - b. For the tab sequence established in Exercise 13a, set the `TabStop` property of `btnOne` to `False` and determine how the tab sequence is affected. What was the effect on the objects' `TabIndex` values?
 - c. For the tab sequence established in Exercise 13a, set the `TabStop` property of `btnOne` to `True` and its `Visible` property to `False`. What is the run time tab sequence now? Did these changes affect any object's `TabIndex` values?
 - d. For the tab sequence established in Exercise 13a, set the `TabStop` property of `btnOne` to `True`, its `Visible` property to `True`, and its `Enabled` property to `False`. What is the run time tab sequence now? Did these changes affect any object's `TabIndex` values?
 - e. Change the tab sequence so that focus starts on `txtFirst`, and then goes to `btnOne`, `txtSecond`, and finally `btnTwo`. For this sequence, what are the values of each object's `TabIndex` property?

2.5 Adding Additional Event Procedures

Now that we have added four objects to Program 2-3 (The Hello Application, Version 3.0, shown in Figure 2-40), we will need to supply these objects with event code. Although each object can have many events associated with it, one of the most commonly used events is the clicking of a button. For our Hello Application, we will initially create three mouse click event procedures, each of which will be activated by clicking one of the three buttons. Two of these event procedures will be used to change the text displayed in the text box, and the last will be used to exit the program.

To change an object's property value while a program is running, a statement is used that has the syntax:

```
Object.Property = value
```

The term to the left of the equals sign identifies the desired object and property. For example, `btnMessage.Name` refers to the `Name` property of the control named `btnMessage`, and `txtDisplay.Text` refers to the `Text` property of the control named `txtDisplay`. The period between the object's name and its property is required. The value to the right of the equal sign provides the new setting for the designated property. The equal sign is used as an assignment operator where the value on the right is 'assigned' to the object on the left.

For our program, we want to display the text `Hello World!` when the button named `btnMessage` is clicked. This requires that the statement

```
txtDisplay.Text = "Hello World!"
```

be executed for the click event of the `btnMessage` control. Notice that this statement will change a property of one object, the `TextBox` object, using an event procedure associated with another object, a `Button` object. Now let's attach this code to the `btnMessage` button so that it is activated when this control is clicked. The required event procedure code is:

```
Private Sub btnMessage_Click(ByVal sender As System.Object, ByVal e As _
    System.EventArgs) Handles btnMessage.Click

    txtDisplay.Text = "Hello World!"
End Sub
```

To enter this code, double-click the `btnMessage` control. (Make sure you have the design form shown in Figure 2-33 on the screen). This will open the Code window shown in Figure 2-49. As always, the stub for the desired event is automatically supplied for you, requiring you to complete the procedure's body with your own code. You might also notice that the keywords `Private`, `Sub`, `ByVal`, `As`, and `End` are displayed in a different color than the procedure's name.⁸

Also, note that the first statement above beginning with `Private` is too long to fit on one line in this book. To continue a Visual Basic statement on the next line, type a space followed by the underscore symbol ("`_`") and indent the continued statement.

The object identification box should display `btnMessage` and the procedure identification box should display `Click`. This indicates that the current object is the `btnMessage` control and that the procedure we are working on is for the `Click` event. If either of

⁸Typically, the color for keywords is blue and is automatically supplied when a keyword is typed.

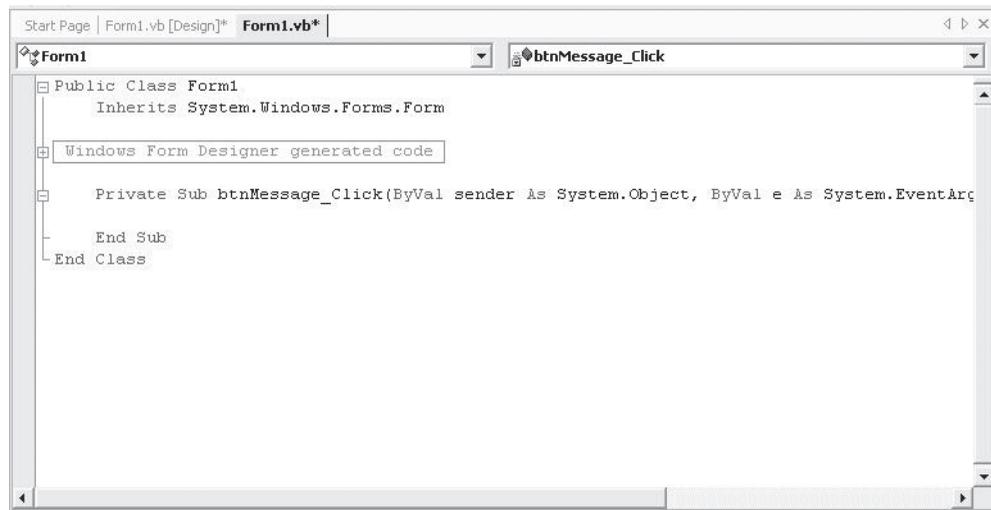


Figure 2-49 The Code Window

these boxes does not contain the correct data, click the underlined arrow to the right of the box and then select the desired class and method. Note that when you click on the arrow to the right of the class name box, a drop-down list appears which lists all of the form's objects, including the form itself.

When the Code window looks like the one shown in Figure 2-49, type in the line

```
txtDisplay.Text = "Hello World!"
```

between the header line, `Private Sub btnMessage_Click()`, and terminating `End Sub` line, so that the complete procedure appears as

```
Private Sub btnMessage_Click(ByVal sender As System.Object, ByVal e As _
    System.EventArgs) Handles btnMessage.Click
    txtDisplay.Text = "Hello World!"
End Sub
```

After your procedure is completed, press the F5 function key to run the program. When the program is running, activate the `btnMessage` control by either clicking it, tabbing to it and pressing the Enter key, or using the hot key combination Alt+M. When any one of these actions is performed, your screen should appear as shown in Figure 2-50.

One of the useful features of Visual Basic is the ability to run and test an event procedure immediately after you have written it, rather than having to check each feature after the whole application is completed. You should get into the habit of doing this as you develop your own programs.



Figure 2-50 The Interface Produced by Clicking the Message Button

Now let's finish our second application by attaching event code to the click events of the remaining two buttons, and then fixing a minor problem with the TextBox control.

Bring up the Code window for the btnClear button by double-clicking this control after you have terminated program execution and are back in the design mode. When the Code window appears, add the single line

```
txtDisplay.Text = ""
```

between the procedures header (Private Sub) and terminating line (End Sub).

When this is completed, the procedure should appear as follows:

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

The string "", with no spaces, is called the *empty string*. This string consists of no characters. Setting the Text property of the text box to this string value will have the effect of clearing the text box of all text. Note that a value such as " ", which consists of one or more blank spaces, will also clear the text box. However, a string with one or more blank spaces is not an empty string, which is defined as a string having *no* characters.

When this procedure is completed, use the arrow to the right of the class name box in the Code window to switch to the btnExit control. (You can also double-click the btnExit control to open the Code window.) The event procedure for this event should be:

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

`Beep` is an instruction that causes the computer to make a short beeping sound. The keyword `End` terminates an application.

You are now ready to run the application by pressing the F5 function key. Running the application should produce the window shown in Figure 2-51. Note that when the program is initially run, focus is on the `btnMessage` button and the text box is empty. The empty text box occurs because we set this control's `Text` property to a blank during design time. Similarly, focus is on the `btnMessage` box because this was the first control added to the Form (its `TabIndex` value is 0).

Now click the `Message` control to activate the `btnMessage_Click()` procedure and display the message shown in Figure 2-52.



Figure 2-51 The Initial Run Time Window



Figure 2-52 The Run Time Window after the Message Button is Clicked

Clicking the Clear button invokes the `btnClear_Click()` procedure, which clears the text box, whereas clicking the Exit button invokes the `btnExit_Click()` procedure. This procedure causes a short beep and terminates program execution.

Comments

Comments are explanatory remarks made within a program. When used carefully, comments can be helpful in clarifying what the complete program is about, what a specific group of statements is meant to accomplish, or what one line is intended to do.

Comments are indicated by using an apostrophe or the keyword `Rem`, which is short for Remark. Using an apostrophe is the preferred method and it will be used in this book. For example, the following are comment lines:

```
' this is a comment
' this program calculates a square root
```

With one exception, comments can be placed anywhere within a program and have no effect on program execution. Visual Basic ignores all comments—they are there strictly for the convenience of anyone reading the program. The one exception is that comments cannot be included at the end of a statement that is continued on the next line.

A comment can always be written either on a line by itself or on the same line as a program statement that is not continued on the next line. When written on a line by itself, either an apostrophe or the keyword `Rem` may be used. When a comment is written on the same line as a program statement, the comment must begin with an apostrophe. In all cases, a comment only extends to the end of the line it is written on. For example, the following event procedure illustrates the use of comments.

```
' This is the click event procedure associated with the Exit
Private Sub btnExit_Click(ByVal sender As System.Object, ByVal e As _
    System.EventArgs) Handles btnExit.Click
    Beep() 'This causes a short beep
End Sub ' This ends the application
```

If you need to create multiline comments, each line must begin with an apostrophe. Typically, many comments are required when using unstructured programming languages. These comments are necessary to clarify the purpose of either the program itself or individual sections and lines of code within the program. In Visual Basic, the program's inherent modular structure is intended to make the program readable, making the use of extensive comments unnecessary. However, if the purpose of a procedure or any of its statements is still not clear from its structure, name, or context, include comments where clarification is needed.

Statement Categories

You will have many statements at your disposal while constructing your Visual Basic event procedures. All statements belong to one of two broad categories: executable

statements and nonexecutable statements. An *executable statement* causes some specific action to be performed by the compiler or interpreter. For example, a `MessageBox.Show` statement or a statement that tells the computer to add or subtract a number is an executable statement. A nonexecutable statement is a statement that describes some feature of either the program or its data but does not cause the computer to perform any action. An example of a nonexecutable statement is a comment statement. As the various Visual Basic statements are introduced in the upcoming sections, we will point out which ones are executable and which are nonexecutable.

A Closer Look at the TextBox Control

Text boxes form a major part of most Visual Basic programs, because they can be used for both input and output purposes. For example, run the Program 2-3 (see Figure 2-51) again, but this time click on the text box. Note that a cursor appears in the text box. At this point, you can type in any text that you choose. The text that you enter will stay in the text box until you click on one of the buttons, either changing the text to Hello World!, clearing the Text box of all text, or terminating the program.

Because we have constructed the program to use the text box for output display purposes only, we would like to alter the operation of the text box so that a user cannot enter data into it. To do this, we set the text box's Enter event to immediately put focus on one of the buttons. The following procedure accomplishes this:

```
Private Sub txtDisplay_Enter(ByVal sender As Object, ByVal e As _
    System.EventArgs) Handles txtDisplay.Enter

    btnMessage.Focus()
End Sub
```

Enter this procedure now in the text box's Code window. When you first open the Code window for the TextBox object (by either pressing the Shift+F4 keys or using the View menu), the Code window may appear as shown in Figure 2-53. If this happens, click the underlined arrow to the right of the class name box and select the `txtDisplay` object. Then click the Method Name underlined arrow and choose the Enter event.

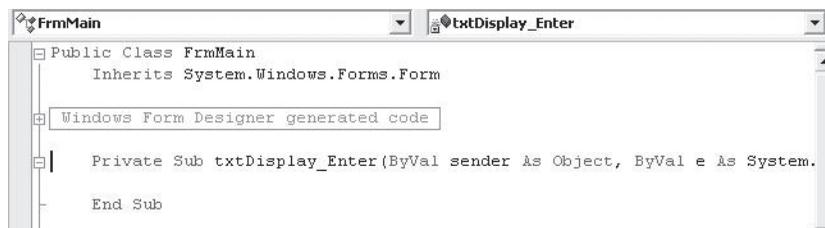


Figure 2-53 The Code Window

Focus is a method in which the Button object sets the focus on its object.⁹ Thus, when a user clicks on the text box, it will trigger the `txtDisplay_Enter()` procedure, which in turn will call the **Focus** method of the `btnMessage` button. This method will set the focus on the Message Button.

Note that when you initially entered characters in the text box, before we deactivated it for input, the entered characters were actually accepted as a string of text. This is always true of a text box—all data entered or displayed is considered a string. As we will see in the next chapter, when we want to use a text box to input a number, such as 12356, we will have to carefully check that a string representing an invalid number, such as 123a56, is not inadvertently entered. This type of validation is necessary because the text box does not filter out unwanted characters from an input or displayed string.

Exercises 2.5

1. a. Determine the events that can be associated with a button. (*Hint*: Create a button and use the Code window to view its various events.)
 - b. List the event names for each event that can be associated with a button.
2. Repeat Exercise 1a for a Text box.
3. Repeat Exercise 1a for a Label.
4. List the objects and the events that the following procedures refer to:
 - a. `Private Sub btnDisplay_Click(ByVal sender As Object, ByVal e As _ System.EventArgs) Handles btnDisplay.Click`
 - b. `Private Sub btnBold_Leave(ByVal sender As Object, ByVal e As _ System.EventArgs) Handles btnBold.Leave`
 - c. `Private Sub txtInput_Enter(ByVal sender As Object, ByVal e As _ System.EventArgs) Handles txtInput.Leave`
 - d. `Private Sub txtOutput_Leave(ByVal sender As Object, ByVal e As _ System.EventArgs) Handles txtOutput.Leave`
5. Using the correspondence shown here, follow the instructions below:

Event Name	Event
Click	Click
DbClick	Double click
Enter	Enter
Leave	Leave

- a. Write the header line for the double-click event associated with a Label control named `lblFirstName`.
- b. Write the header line for the lost focus event of a text box named `txtLastName`.
- c. Write the header line for the got focus event of a text box named `txtAddress`.

⁹An alternative solution is to set the `Locked` property of the text box to `True`. With this property set to `True`, the text box is locked from receiving any input and effectively becomes a read-only box. However, it still can be clicked on and receive focus.

6. Write instructions that will display the following message in a text box named txtTest:
- Welcome to Visual Basic
 - Now is the time
 - 12345
 - 4 * 5 is 20
 - Vacation is near

7. For each of the following procedures, determine the event associated with the button btnDisplay and what is displayed in the text box named txtOut:

a. `Private Sub btnDisplay_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles btnDisplay.Click`
`txtOut.Text = "As time goes by"`
`End Sub`

b. `Private Sub btnDisplay_Enter(ByVal sender As Object, ByVal e As System.EventArgs) Handles btnDisplay.Enter`
`txtOut.Text = "456"`
`End Sub`

c. `Private Sub btnDisplay_Leave(ByVal sender As Object, ByVal e As System.EventArgs) Handles btnDisplay.Leave`
`txtOut.Text = "Play it again Sam"`
`End Sub`

d. `Private Sub btnDisplay_Enter(ByVal sender As Object, ByVal e As System.EventArgs) Handles btnDisplay.Enter`
`txtOut.Text = " "`
`End Sub`

8. a. **TextAlign** is the name of a property that can be set for a text box. What do you think this property controls?
- b. What display do you think the following procedure produces when the button btnOne is clicked?

```
Private Sub btnOne_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles BtnOne.Click

    txtBox1.TextAlign = HorizontalAlignment.Center
    txtBox1.Text = "Computers"

End Sub
```

For Exercises 9 and 10, create the given interface and initial properties. Then complete the application by writing code to perform the stated task.

9. Object	Property	Setting
Form	Name	frmMain
	Text	Messages
Button	Name	btnGood
	Text	&Good
Button	Name	btnBad
	Text	&Bad
Text box	Name	txtMessage
	Text	(blank)

When a user clicks the btnGood button, the message *Today is a good day!* should appear in the text box, and when the btnBad button is clicked, the message *I'm having a bad day today!* should be displayed.

- Write a Visual Basic application having three buttons and one text box that is initially blank. Clicking the first button should produce the message *See no evil*. Clicking the second button should produce the message *Hear no evil*, and clicking the third button should produce the message *Speak no evil* in the text box.

2.6 Focus on Program Design and Implementation: Creating a Main Menu

Most commercial applications perform multiple tasks, with each task typically assigned its own form. For example, one form might be used for entering an order, a second form for entering the receipt of merchandise into inventory, and a third form used to specify information needed to create a report.

Although each additional form can easily be added to an existing project using the techniques presented later in this section, there is the added requirement for activating each form in a controlled and user-friendly manner. This activation can be accomplished using an initial main menu form that is displayed as the application's opening window. This menu of choices provides the user with a summary of what the application can do, and is created as either a set of buttons or as a menu bar. Here, we show how to rapidly prototype a main menu consisting of buttons, using the Rotech Systems case (see Section 1.5) as an example. The procedure for constructing a menu bar is presented in Section 1.5.

Button Main Menus

The underlying principle involved in creating a main menu screen is that a user can easily shift from one form to a second form by pressing a button. How this works for two screens is illustrated in Figure 2-54. Here, pressing the button shown on the Main Menu form causes the second form to be displayed. Similarly, pressing the second form's button redisplay the Main Menu form.

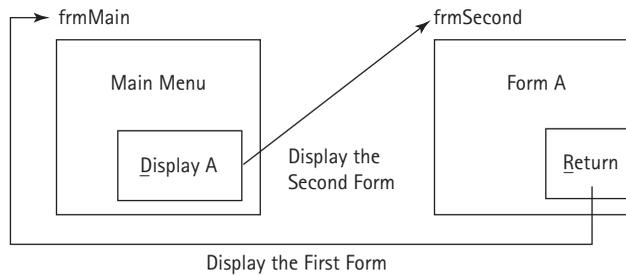


Figure 2-54 Switching Between Two Screens

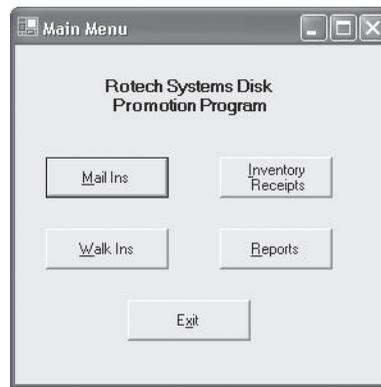


Figure 2-55 A Sample Main Menu Screen

The switching between screens that is provided by the scheme shown in Figure 2-54 can easily be extended to produce a main menu screen, a sample of which is shown in Figure 2-55. For this screen there would be four additional screens, with each screen displayed by clicking one of the buttons on the main menu. In a completed application, each additional screen is different, but each screen would have a Return button to display the main menu screen again.

The menu options shown in this figure would be provided as the first operational form presented to the user. The term *operational form* means that the form is used to interact directly with the operational part of the system to perform a task, unlike an information form. *Information forms* are purely passive forms that provide information about the application, and which, if omitted from the system, would not stop the system from performing its intended functions. Examples of information forms are an opening splash screen containing general advertising information that is displayed for a few seconds before the first operational form is presented, an About Box providing information about the application (such as author, copyright holder, and version number), and Help forms providing assistance on how to use various application features. An application's initial main menu screen is constructed from the information contained in the system's

Table 2-6 The Initial Rotech Systems TTL Table

Form: Task	Trigger Object	Event
Enter Inventory Receipts		
Enter & Process a Mail-in Response		
Enter & Process a Walk-in Request		
Produce Reports		
Exit the System		

initial TTL table. We will construct the Rotech Systems main menu to provide access to the tasks listed in the TTL table developed in Section 1.5, and reproduced here for convenience as Table 2-6.

Because each menu option in our main menu is to be activated by the clicking of a button, we can now easily complete Table 2-6. The form that we will use to activate these tasks is a Main Menu form. Each object used to trigger the listed task will be a button, and each event trigger will be a button's Click event. After names have been assigned arbitrarily to each button, the completed initial TTL table is listed as Table 2-7.

Tips From The Pros

Creating a Main Menu

The development of a menu system is quite easy, provides one of the most visually impressive functioning parts of a system, and actually demands the least in terms of programming competency. To develop a menu, follow these steps:

1. Make the opening application's form a Main Menu form, which is a form that contains buttons used to display other forms.
2. Initially, add a single new form to the project, which ultimately will be replaced by an operational form. (Later on, additional new forms will be added, one for each operational form required by the project.)
3. Provide the new form with a button having a caption such as Return to Main Menu form. This button is referred to as the Return button.
4. Attach three lines of code to the new form's Return button's Click event.

These three lines of code appear as:

```
Dim frmMainRef As New frmMain()
Me.Hide           ' this is a valid statement
frmMainRef.Show  ' replace frmMain with any valid form name
```

The first code line creates a reference to the form frmMain. It is needed in order to refer to existing forms. (The New keyword is explained more fully in Chapter 12.) The second code line is an instruction to hide the current form from view, while the third line displays the frmMain form. As a result, if the Main Menu form is named frmMain, the statement frmMainRef.Show displays the Main Menu form when the Return button is pressed. In general, each new form will have more than one button. However, one button should always return the user directly to the Main Menu.

5. Attach three similar lines of code to each Main Menu button used to display a new form. These lines of code take the form:

```
Dim frmDesiredRef As New frmDesired()
Me.Hide           ' this is a valid statement
frmDesired.Show  ' replace frmDesired with any valid form name
```

Initially, all Main Menu buttons will be used to display the same form added in Step 2. The reason for using the Hide method rather than the Close method is based on the assumption that we will be returning to each form many times, so we try to keep each form in memory to minimize loading times.

The information in Table 2-7 tells us that, operationally, our main menu will consist of five buttons, with each button's Click event used to display another form. Specifically, clicking on the btnInvRec button will activate an operational form for entering inventory receipts. Clicking the btnMailin button will activate an operational form for entering an order for the promotional 10-diskette pack. Clicking the btnWalkin button will activate an operational form for entering a walk-in order. Clicking the btnReport button will activate an operational form for producing reports, and clicking the btnExit button will terminate program execution. By using this information, and adding an information label to the Main Menu form, we construct Table 2-8 displaying the initial Main Menu properties table.

Table 2-7 The Completed Initial Rotech Systems TTL Table

Form: Main Menu Task	Trigger Object	Event
Enter Inventory Receipts	btnInvRec	Click
Enter & Process a Mail-in Response	btnMailins	Click
Enter & Process a Walk-in Request	btnWalkins	Click
Produce Reports	btnReport	Click
Exit the System	btnExit	Click

Table 2-8 The Rotech Systems Main Menu Properties Table

Object	Property	Setting
Form	Name	frmMain
	Text	Main Menu
Button	Name	btnMailins
	Text	&Mail Ins
	TabIndex	0
Button	Name	btnWalkins
	Text	&Walk Ins
	TabIndex	1
Button	Name	btnInvRec
	Text	&Inventory Receipts
	TabIndex	2
Button	Name	btnReports
	Text	&Reports
	TabIndex	3
Button	Name	btnExit
	Text	E&xit
	TabIndex	4
Label	Name	lblHeader
	Text	Rotech Systems Disk Promotion Program
	Font	MS Sans Serif, Bold, Size = 10

Figure 2-56 illustrates a form having the properties described by Table 2-8. Specifically, we have chosen to group the two buttons associated with order entry (Mail Ins and Walk Ins) in one column, align the two remaining buttons in a second column, and center the Exit button below and between the two columns. If there were an even number of buttons, we could have aligned them vertically into two columns, including the Exit button as the last button in the second column, and made all the buttons the same size. Figures 2-57 and 2-58 show two alternatives to Figure 2-56. In each case, the size and placement of each button is determined by the programmer, with the only overriding design consideration at this point being that the size of each button within a grouping should be the same and that the buttons should align in a visually pleasing manner. Although we will give a number of form design guidelines in Section 3.6, the basic rule is to produce a functionally useful form that is dignified and not ornate. The design and colors of your form should always be appropriate to the business whose application you are producing.

We have given the Mail Ins button the initial input focus because of the majority of times a user will be interacting with it and the fact that it deals with the system's most dynamic data.. Thus, by simply pressing the Enter key, a user will activate the most commonly used button in the menu. The tab sequence then ensures a smooth transition that moves the focus from the Mail Ins button down the first column to

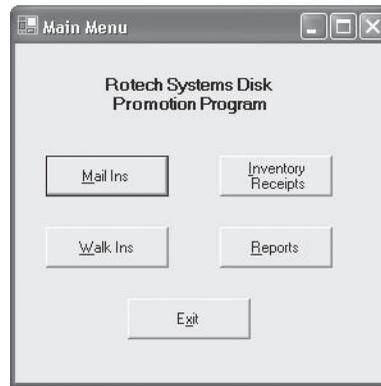


Figure 2-56 Rotech's Initial Main Menu Form

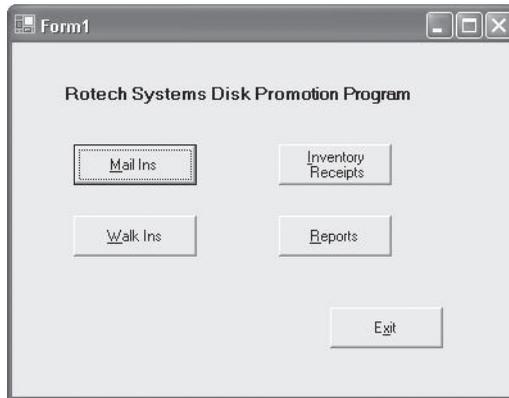


Figure 2-57 A Possible Main Menu Layout



Figure 2-58 Another Main Menu Layout

the top of the second column of buttons, where the *Inventory Receipts* button is located, and then down this column, over to the *Exit* button, and back to the top of the first column.

In reviewing Figure 2-56, note that the window furnishes information about the application and provides a current list of available choices shown as a sequence of buttons. We now need to add additional forms and provide code so that pressing any of the buttons, except the *Exit* button, hides the current Main Menu form and displays the new form appropriate to the selected menu choice. Pressing the *Exit* button performs the normal operation of ending program execution.

Adding a Second Form

At this stage in our application's development we have neither sufficient understanding of the system's requirements nor sufficient knowledge of Visual Basic to construct a

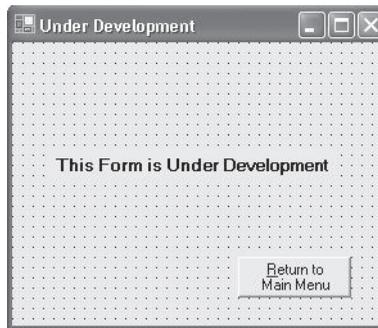


Figure 2-59 The Project's "Under Development" Form

meaningful form to carry out the tasks indicated by the Main Menu. We can, however, easily add a second form, shown in Figure 2-59, to indicate that the form is under development. Note that this second form contains a Label object and a Button object, the same types of objects included in our Main Menu form. The properties for this second form are listed in Table 2-9. Because this form performs the single task of returning to the Main Menu form, a TTL table listing this single task is not created.

To add this second form to our project, select the Add Windows Form item from the Project menu, as shown in Figure 2-60. This will open the Add New Item dialog box shown in Figure 2-61, from which you should select the Windows Form icon under the Templates category. This will add the second form to the Project Explorer Window, as shown in Figure 2-62. Once you have generated this new form, configure it with the objects and properties listed in Table 2-9.

The form you have just created is an example of a stub form. The functional use of a stub form is simply to see that an event is correctly activated. In our particular case, it will be used to test the operation of the Main Menu to ensure that it correctly displays a second form and provides a return back to the Main Menu from the newly developed

Table 2-9 The "Under Development" Form's Properties Table

Object	Property	Setting
Form	Name	frmStub
	Text	Under Development
Button	Name	btnReturn
	Text	&Return to Main Menu
Label	Name	lblReturn
	Text	This Form is Under Development
	Font	MS Sans Serif, Bold

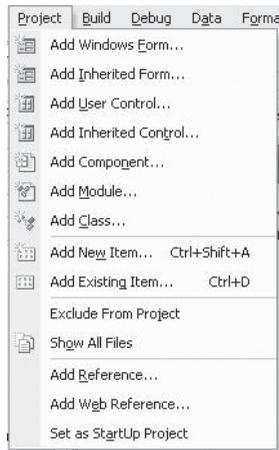


Figure 2–60 Adding a Second Form Using the Project Menu

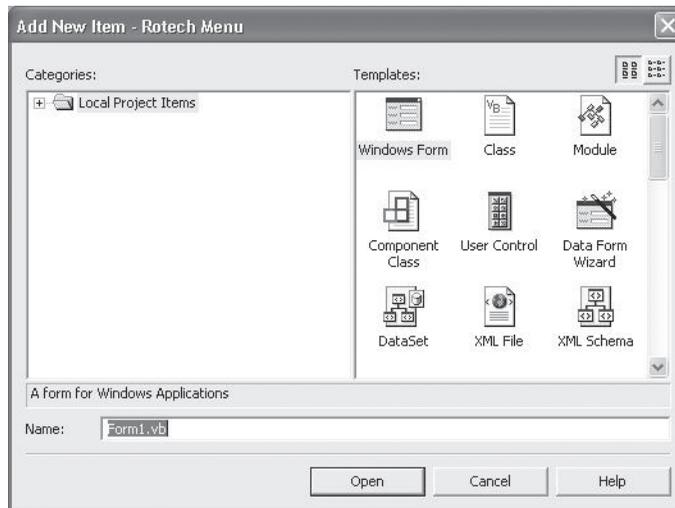


Figure 2–61 The Add New Item Dialog Box

form. To do this, we now have to add the code that correctly shows and hides forms from the user's view.

Displaying and Hiding Forms

The Visual Basic methods provided for displaying and hiding forms from a user's view are listed in Table 2–10. When a project is first executed, the default is to automatically load and display the project's opening form. Other forms must then be loaded into

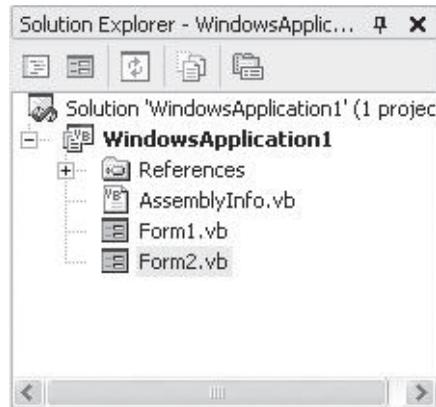


Figure 2–62 The Solution Explorer Window Showing Two Forms

Table 2–10 Form Display Statements and Methods

Instruction	Type	Syntax	Example	Description
Show	Method	Form-nameRef.Show()	frmStubRef.Show()	Displays a form. If the variables for the form are not already loaded into memory, they will be before the form is displayed.
Hide	Method	Me.Hide()	Me.Hide()	Hides a Form object, but does not unload its variables from memory.
Close	Method	Me.Close()	Me.Close()	Hides a Form object and unloads its variables from memory.

memory before they can be displayed, and forms that are no longer needed should be unloaded to conserve memory space. Note that in order to use the Show method, we need to create a reference to the forms that are being manipulated. In Table 2–10, we use Form-nameRef and frmStubRef to designate this reference. To hide or close a form, use Me to refer to the form that should be hidden.

Using the methods listed in Table 2–10, we easily can write event procedures to hide the Main Menu form and load and display the Under Development stub form when any Main Menu button, except the Exit button, is pressed. The required event procedure codes are:

Main Menu Event Procedures

```
Private Sub btnMailins_Click(ByVal sender As Object, ByVal e As _
    System.EventArgs) Handles btnMailins.Click
    Dim frmStubRef As New frmStub()
```

```

    Me.Hide()
    frmStubRef.Show()
End Sub

Private Sub btnWalkins_Click(ByVal sender As Object, ByVal e As _
    System.EventArgs) Handles btnWalkins.Click
    Dim frmStubRef As New frmStub()
    Me.Hide()
    frmStubRef.Show()
End Sub

Private Sub btnInvrec_Click(ByVal sender As Object, ByVal e As _
    System.EventArgs) Handles btnInvrec.Click
    Dim frmStubRef As New frmStub()
    Me.Hide()
    frmStubRef.Show()
End Sub

Private Sub btnReports_Click( ByVal sender As Object, ByVal e As _
    System.EventArgs) Handles btnReports.Click
    Dim frmStubRef As New frmStub()
    Me.Hide()
    frmStubRef.Show()
End Sub

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

```

Each of these **Click** event procedures causes the frmMain form to be hidden and the frmStub form to be displayed whenever any Main Menu button is pressed, except for the **Exit** button. The code for the stub form's return button's **Click** event is:

```

Private Sub btnReturn_Click( ByVal sender As Object, ByVal e As _
    System.EventArgs) Handles btnReturn.Click
    Dim frmMainRef As New frmMain()
    Me.Hide()
    frmMainRef.Show()
End Sub

```

The **Return** button simply hides the current stub form and displays the initial frmMain window. The advantage of using a stub form is that it permits us to run a complete application that does not yet meet all of its final requirements. As each successive form is developed, the display of the stub form can be replaced with a display of the desired form. This incremental, or stepwise, refinement of the program is an extremely

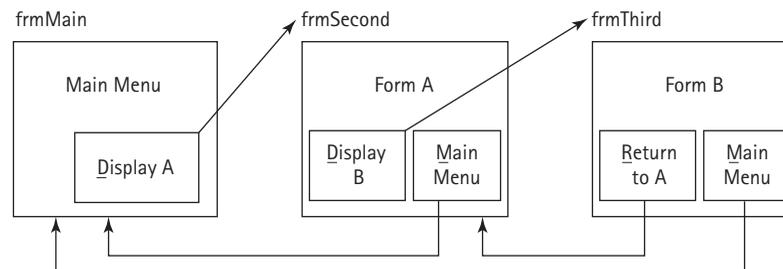
powerful development technique used by professional programmers. Another advantage to this rapid application prototyping technique is that, as new features are required, additional buttons can easily be added to the Main Menu form.

One modification that can be made to our Main Menu form is to have each button call a single, centrally placed, general-purpose procedure that would then make the choice of which form to hide and display based on the button pressed. We consider this approach in Chapter 7 after both selection statements and general-purpose procedures are presented. You will encounter both the current technique and the general-purpose procedure technique in practice, and which one you adopt for your programs is more a matter of style than substance.

Note: The Rotech Systems project, as it exists at the end of this section, can be found on the website <http://computerscience.jpup.com/bronsonvbnet> in the ROTECH2 folder as project rotech2. Note that rotech2 and all subsequent projects are also folders.

Exercises 2.6

1. Implement the two screens and the relationship between the two forms previously shown in Figure 2-54. When a user presses the single button on the form with the label Main Menu, the screen labeled Form A should appear. When the Return button on Form A is pressed, the form labeled Main Menu should appear.
2. a. Implement the three screens and the relationship shown in the accompanying figure. The Main Menu form should have a single button that displays Form A. Form A should have two buttons: one to return to the Main Menu and one to display Form B. Form B should also have two buttons: one to return control to Form A and one to return control directly to the Main Menu.



- b. Add a fourth form with the label Form C to the project created for Exercise 2a. This form is to be displayed using a button on Form B. In this case, how many buttons should Form B contain for controlling the display of forms? Note that when forms are chained together in the manner shown, there are at most three buttons: one button to “back up” one level and return to the immediately preceding screen, one button to return directly to the Main Menu, and a third button to call the next form in the chain. For multilevel forms, it is also convenient to provide a “hot key” (such as the F10 function key that can be used on all forms) to return directly to the Main Menu form. A second “hot key” (such as the F9 function key) always redisplay the preceding form. (How to create this type of “hot key” is presented in Section 6.7.)

3. Either create the menu system developed in this section or locate and load the project from the <http://computerscience.jpup.com/bronsonvbnet> website (project rotech2 in the rotech2 folder).
4. a. Add a new form, with the following properties, to the Rotech project:

Object	Property	Setting
Form	Name	frmWalkIn
	Text	Walk In Processing Form
Button	Name	btnReturn
	Text	&Return to Main Menu
Label	Name	lblReturn
	Text	This Form is Under Development
	Font	MS Sans Serif, Bold, Size = 10

This new form's button should return control to the Main Menu form when the button is clicked.
- b. Modify the Main Menu Walk In button's Click event so that it displays the form you created in Exercise 4a, rather than displaying the frmStub form.
5. (Case study)

For your selected case project (see project specifications at the end of Section 1.5), complete the project's initial TTL table and then create a working Main Menu that corresponds to the data in the completed table. Additionally, add an Under Development stub form to the project and verify that a user can successfully cycle between forms.

2.7 Knowing About: The Help Facility

No matter how experienced you become at using Visual Basic, there will be times when you'll need some help in either performing a particular task, looking up the exact syntax of a statement, or finding the parameters required by a built-in function. For these tasks you can use Visual Basic's Help Facility. Help includes the entire reference manual, programming examples, and the complete Microsoft Development Network (MSDN) Library.

To access the Help Facility, select either the Contents, Search, or Index options from the Help menu, as shown in Figure 2-63. When any of these options is selected, the screen shown in Figure 2-58 will be displayed.

Note that this main help window is divided into two panes. The left pane, which is referred to as the Navigation pane, contains the three tabs labeled Contents, Index, and Search. Each of these tabs provides a different way of accessing information from the help facility, as summarized in Table 2-11. The right section, which is the Documentation pane, displays all information retrieved from the Library. Note on the left side the words Filtered by. This should typically be set to `Visual Basic and Related` when you're developing a Visual Basic project.

As shown in Figure 2-64, the Contents tab, because it is on top of the other tabs, is the currently active tab. This happened because the Contents option was selected from

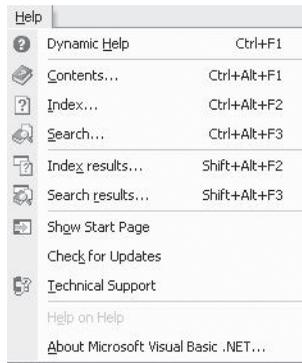


Figure 2–63 The Help Menu Options

Table 2–11 The Help Tabs

Tab	Description
Contents	Displays a Table of Contents for the documentation. This table can be expanded to display individual topic titles; double-clicking a title displays the associated documentation in the Documentation pane.
Index	Provides both a general index of topics and a text box for user entry of a specific topic. Entering a topic causes focus to shift to the closest matching topic within the general index. Double-clicking an index topic displays the associated documentation in the Documentation pane.
Search	Provides a means of entering a search word or phrase. All topics matching the entered word(s) are displayed in a List box. Clicking on a topic displays the corresponding documentation in the Documentation pane.

the **H**elp menu (see Figure 2–64). If either the **I**ndex or **S**earch options had been selected, the same main help window would appear, except that the respective **I**ndex or **S**earch tab would be active. However, no matter which tab is currently active you can switch from one tab to another by clicking on the desired tab.

The Contents Tab

The Contents tab provides a means of browsing through all of the available reference material and technical articles contained within Visual Studio .NET, and specifically within Visual Basic .NET. This tab provides a table of contents of all of the material and displays the topics using the standard Windows tree view. For example, if you expand the topic Visual Studio .NET shown in Figure 2–64 by clicking on the plus sign box [+], and then expand the Introducing Visual Studio .NET topic, you will see the tree shown in Figure 2–65. The information provided in the documentation pane was displayed by

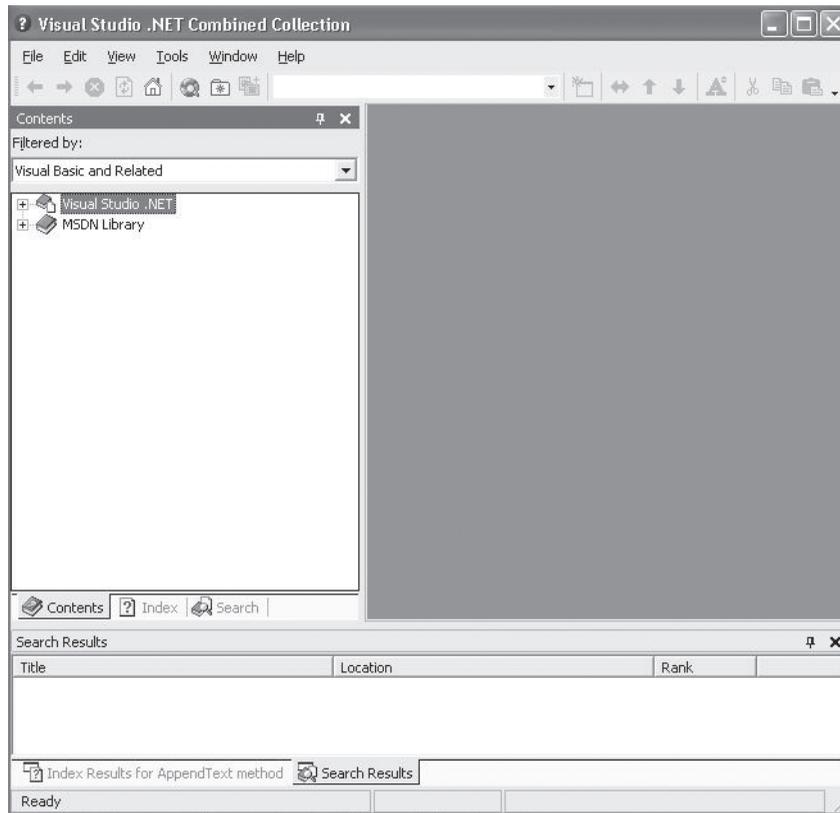


Figure 2-64 The Main Help Window

double-clicking on the page highlighted in the Navigation pane. A hard copy of the displayed page is obtained by either selecting the **P**rint option from the **F**ile menu or right-clicking in the Documentation pane and selecting the Print option.

A very useful feature of the documentation is the hyperlink embedded within the displayed documentation text. By positioning the mouse on underlined text and clicking, the referenced text will be displayed. This permits you to rapidly jump from topic to topic, all while staying within the Documentation pane.

The Index Tab

As shown in Figure 2-66, the Index tab is on top of the other tabs, which makes it the active tab. This tab operates much like an index in a book, with one major improvement: In a book, after looking up the desired topic, you must manually turn to the refer-

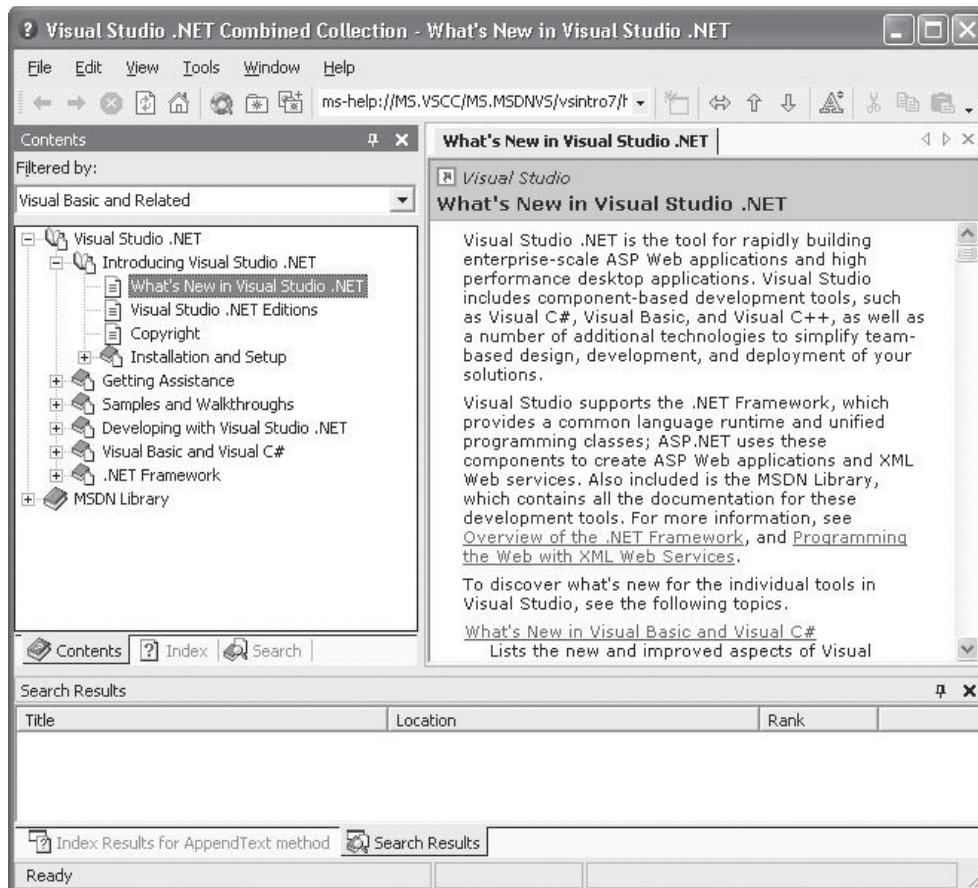


Figure 2-65 Using the Contents Tab

enced page or pages. In the help facility, this “look up and display” is automatic once you indicate the desired topic. Selection of the topic is accomplished by double-clicking on an item within the list of topics contained in the list box (this is the box located at the bottom of the tab). To locate a topic, you can either type the topic’s name within the tab’s **Keyw**ord text box, which causes the item to be highlighted in the List**Box**, or use the scroll bar at the right of the list box to manually move to the desired item.

For example, in Figure 2-67, the topic AppendText Method has been typed in the **Look For**: text box, and the list box entry for this topic has been selected. As each letter is typed in the text box, the selected entry in the list box changes to match the input letters as closely as possible. In this case, because there are multiple library entries for the highlighted topic, if you double-click the highlighted topic, the Multiple Topics dialog box shown in Figure 2-68 appears. By double-clicking the desired item directly in

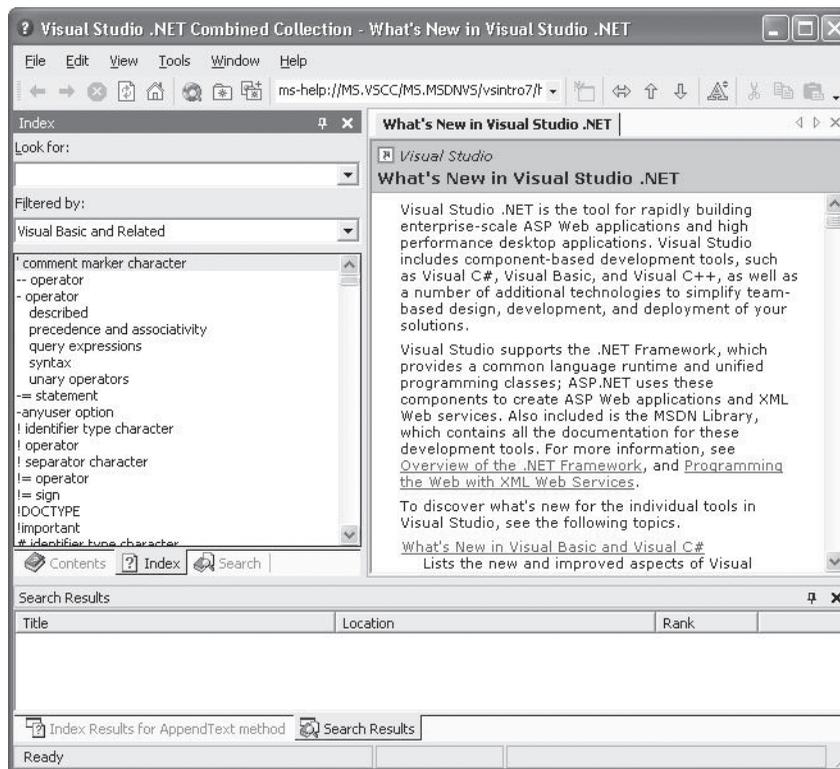


Figure 2-66 The Activated Index Tab

the list box, the documentation for the selected topic is displayed in the Documentation area. Figure 2-69 illustrates the documentation for the item highlighted in Figure 2-68.

A useful feature of the help facility is that once you have selected and displayed the desired topic, you can easily generate a hard copy of the information. This is accomplished by either selecting the Print item from the File menu button at the top of the MSDN Library window, or by using the context menu provided by clicking the right mouse button from within the displayed information.

The Search Tab

The Search tab, shown as the active tab in Figure 2-70, permits you to search for the words entered in the tab's text box in either the complete MSDN Library or sections of it. When creating a search phrase, you should enclose a phrase within double quotes

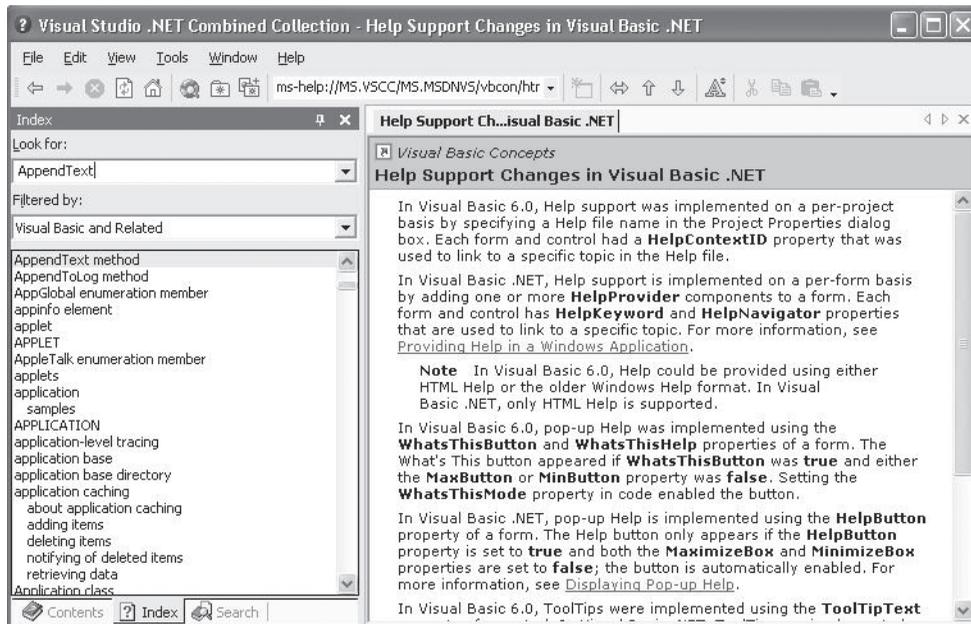


Figure 2-67 The Index Tab with a Typed Entry

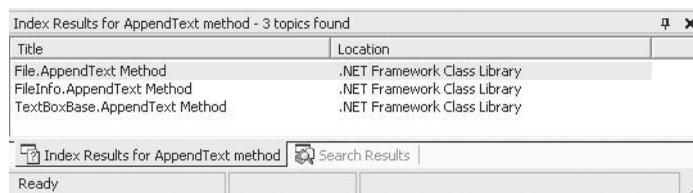


Figure 2-68 The Multiple Topics Dialog Box

and make use of Boolean operators to limit the search. For example, using the filter “Visual Basic and Related,” a list of two topics was generated for the phrase “sqrt method” when the List Topics button shown in Figure 2-70 was pressed. The double quotes direct the search to look for the words “sqrt method” together. If the double quotes are omitted, the search finds all occurrences of either the word “sqrt” or the word

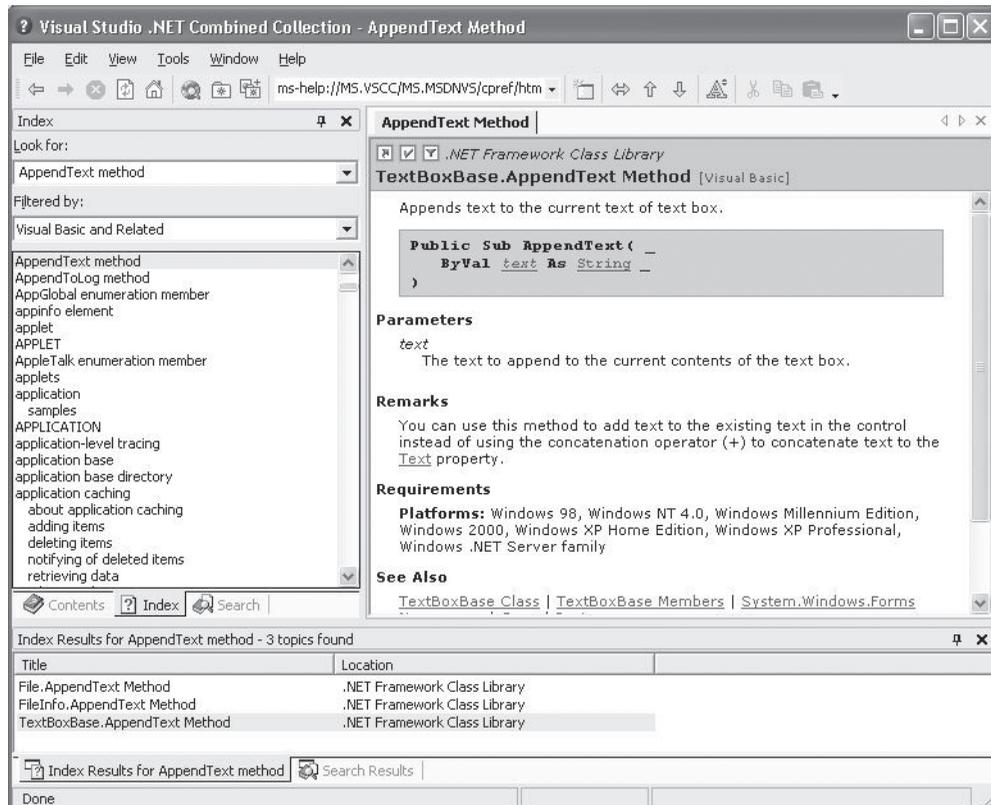


Figure 2-69 The Documentation Display

“method,” and returns ten matches. The documentation shown in Figure 2-70 was displayed by clicking the second topic listed in the list box. As an aid to reading the help text, it is sometimes useful to have the search words highlighted in the help text. This can be accomplished by checking the box “Highlight search hits (in topics)” located below the search button.

Dynamic Help

Dynamic Help is an excellent way to get information about the IDE and its features. The Dynamic Help window displays a list of help topics that changes as you perform operations. This window occupies the same location as the Properties window. If the Dynamic Help window is not open, click **Help** on the menu bar and then click **Dynamic Help**. When you click a word or component such as a form or control, links to relevant articles appear in the Dynamic Help window. This window also has a toolbar that provides access to the **Contents**, **Index**, and **Search** Help features. Figure 2-71 shows a Dynamic Help window.

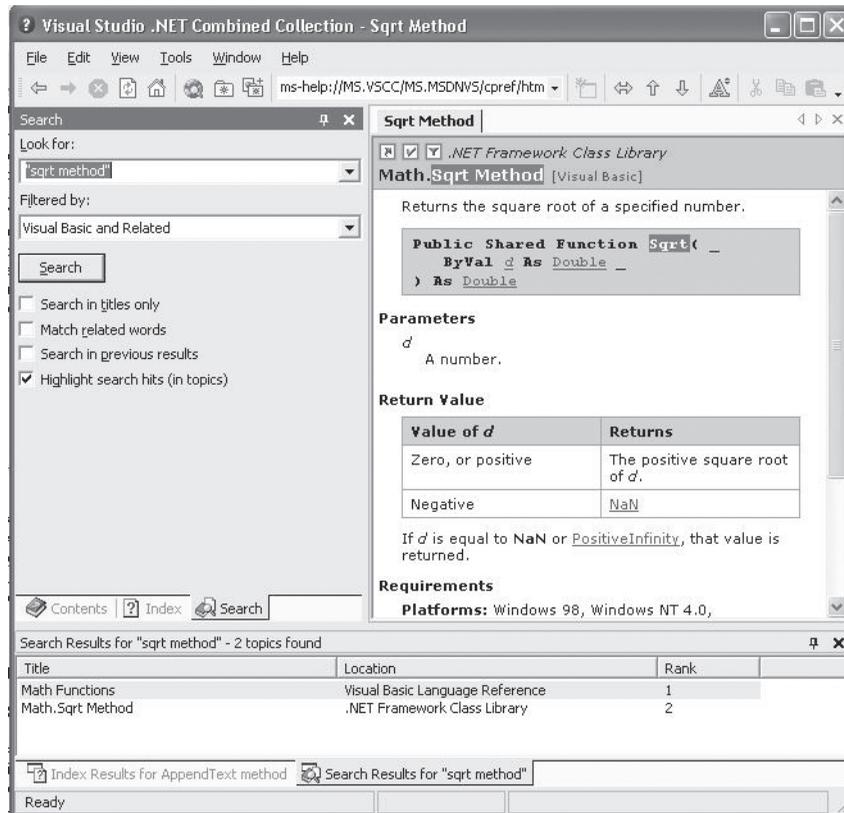


Figure 2-70 Using the Search Tab

Context-Sensitive Help

A quick way to view Help on any topic is to use context-sensitive Help. This is similar to dynamic help, except that it immediately displays a relevant article rather than presenting a list of articles. To use this facility, select an object such as a form or control and press F1. You can also highlight a word in a line of code and press F1 to get information on that topic.

2.8 Common Programming Errors and Problems

One of the most frustrating problems when learning Visual Basic is not being able to locate all of the elements needed to create an application. For example, you may not have the Form object, Toolbox, nor Properties window on the design screen. To bring up the form, either retrieve an existing project or select **N**ew Project from the File menu



Figure 2-71 The Dynamic Help Window

(hot key sequence Alt+F, then N). To open the Toolbox object or the Properties window, select the View option from the menu bar and then select the desired window.

A common error made by beginning programmers is forgetting to save a project at periodic intervals at design time. Although you can usually forego periodic saves, every experienced programmer knows the agony of losing work as a result of a variety of mistakes or an unexpected power outage. To avoid this, you should develop the habit of periodically saving your work.

2.9 Chapter Review

Key Terms

accelerator key
code module
Code window
button
comment
controls
design screen
design time

dialog box
empty string
event
executable statement
focus
form
Form module
graphical user interface

hot key	properties
identifier	run time
label	sizing handles
menu bar	string
methods	text box
MessageBox.Show	toolbar
nonexecutable statement	Toolbox

Summary

1. An object-oriented language permits the use and creation of new object types.
2. Visual Basic .NET is a true object-oriented language.
3. Event-based programs execute program code depending on what events occur, which depends on what the user does.
4. GUIs are graphical user interfaces that provide the user with objects that recognize events such as clicking a mouse.
5. An *application* is any program that can be run under a Windows Operating System.
6. The term *design time* refers to the period of time during which a Visual Basic application is being developed under control of Visual Basic.
7. The term *run time* refers to the period of time when an application is executing.
8. A Visual Basic program consists of a visual part and a language part. The visual part is provided by the objects used in the design of the graphical user interface, whereas the language part consists of procedural code.
9. The basic steps in developing a visual basic program are:
 - a. Creating the GUI.
 - b. Setting the properties of each object on the interface.
 - c. Writing procedural code.
10. A form is used during design time to create a graphical user interface for a Visual Basic application. At run time the form becomes a window.
11. The most commonly placed objects on a form are buttons, labels, and text boxes.
12. Each object placed on a form has a name property. Development teams may choose guidelines for naming objects but in this book, form names begin with the prefix frm, text boxes begin with txt, and buttons begin with btn. Names must be chosen according to the following rules:
 - a. The first character of the name must be a letter.
 - b. Only letters, digits, or underscores may follow the initial letter. Blank spaces, special characters, and punctuation marks are not allowed; use the underscore or capital letters to separate words in an identifier consisting of multiple words.
 - c. A name can be no longer than 1016 characters.
 - d. A name should not be a keyword.

Test Yourself–Short Answer

1. A Visual Basic programmer works with the application in two modes: run time and _____.



2. The letters GUI are an acronym for _____.
3. There are two basic parts to a Visual Basic application; they are: _____ and _____.
4. At run time, a form becomes a _____.
5. Using the right-click mouse button will produce a _____.
6. List the three design steps required in creating a Visual Basic application.
7. What is the difference between the Name property and the Text property?
8. Write a Visual Basic statement to clear the contents of a text box named txtText1.
9. Write a Visual Basic statement to clear a form.
10. Write a Visual Basic statement to place the words “Welcome to Visual Basic” in a text box named txtWelcome.

Programming Projects

Note: On all programming projects that you submit, include your name (or an identification code, if you have been assigned one), and the project number in the lower left-hand corner of the form.

1. a. Create the run time interface shown in Figure 2-72. The application should display the message “My first Visual Basic Program” in a text box when the first button is clicked, and the message “Isn’t this neat?!?” when the second button is clicked. Both messages should be in MS Sans Serif, 18 point, bold font.

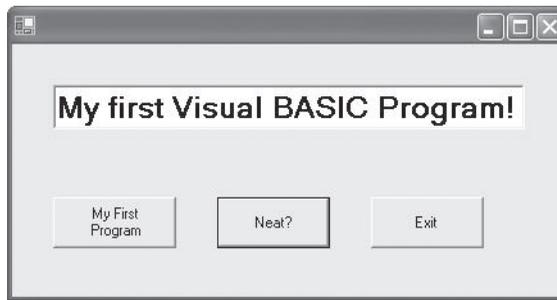


Figure 2-72

- b. (Extra Challenge) Make the label appear randomly on the form by using the intrinsic function RND, the form's Width and Height properties, and the label's Left property.
2. Create the form shown in Figure 2-73 that displays a label with the caption text Hello World. The form should have four buttons. One button should have the caption Invisible, and when this button is pressed the Hello World caption should become invisible, but should have no effect on any other label. A second button, with the caption New Message, should make the Hello World caption invisible and display the text This is a New Message. Both messages should be in MS Sans Serif, 14 point, bold font. The third button, with the caption

Reset, should make the `Hello World` caption reappear. Finally, an Exit button should terminate the program.



Figure 2-73



