

Section 1

The basics of Visual Basic programming

The fastest way to learn Visual Basic programming is to do it, and that's the approach that the chapters in this section take. So in chapter 1, you'll learn how to develop a substantial Visual Basic application. When you're done, you'll know how to build a user interface and how to add the code to it that makes it work the way you want it to.

Then, chapter 2 presents the coding essentials that you'll use in the Visual Basic applications that you develop. Chapter 3 shows you how to develop a user interface that uses more forms and controls. And chapter 4 shows you how to use the Visual Basic features that help you test and debug an application. By the time you finish these chapters, you'll have developed an application that consists of three forms in a Multiple Document Interface. You'll have the essential skills that you need for working with every application you develop. And you'll have a clear view of what Visual Basic programming is and what you have to do to become proficient at it.

To make sure that you're successful as you develop, test, and debug the applications in each of these chapters, exercise sets are interspersed throughout the text. These exercises guide you through the development and enhancement of the applications; they encourage you to try new techniques; and they are an essential part of the learning process.

Mike Murach & Associates



2560 West Shaw Lane, Suite 101
Fresno, CA 93711-2765
(559) 440-9071 • (800) 221-5528

murachbooks@murach.com • www.murach.com

Copyright © 1999 Mike Murach & Associates. All rights reserved.

1

Introduction to Visual Basic programming

Although most programming books start with an introductory chapter or two, the quickest and best way to *learn* Visual Basic programming is to *do* Visual Basic programming. That's why this chapter shows you how to develop a substantial Visual Basic application without any preface or preamble. When you turn the page, you'll start learning how to build the user interface for the application. And before long, you'll start to experience the excitement of Visual Basic programming.

To make sure that you're able to successfully develop the application that's illustrated in the text, three exercise sets within this chapter guide you through its development. If you have extensive programming experience, you may want to develop the application without the guidance of the exercise sets. But otherwise, doing the exercises will help you learn faster and more thoroughly.

How to build a user interface	4
An introduction to forms and controls	4
How to start a new project	6
How to work with the IDE	8
How to add controls to a form	10
How to set properties	12
Common properties for forms and controls	14
The property settings for the Calculate Investment form	16
How to save a project and its forms	18
How to test a form	18
Exercise set 1-1: Build the user interface	20
How to write the code for an interface	22
How an application responds to events	22
How to code an event procedure	24
How to code a general procedure	26
How to refer to properties and methods	28
How to use functions	30
How to get help information	32
How to test and debug an application	34
Exercise set 1-2: Write the code	36
An enhanced version of the Calculate Investment form ..	38
How the enhanced interface works	38
The property settings for the new controls	38
The code for the enhanced application	40
How to print the code for an application	42
How to create an EXE file for an application	42
Exercise set 1-3: Enhance the application	44
Perspective	46

How to build a user interface

When you develop a Visual Basic application, you start by designing a user interface. To do that, you add controls like text boxes and command buttons to a form. Then, you set the properties for the form and its controls. After you get that interface working right, you can write the Visual Basic code that makes the interface do what the user wants.

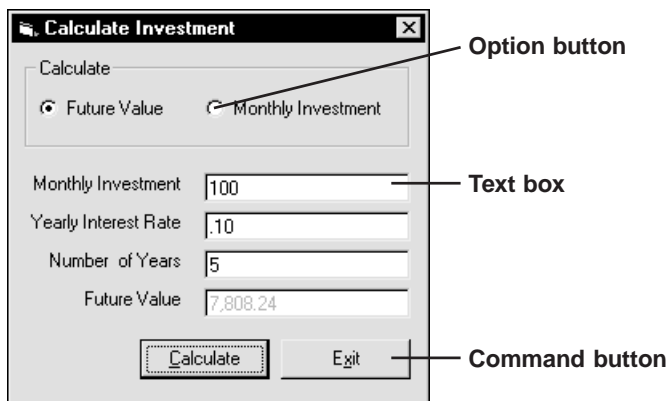
An introduction to forms and controls

Figure 1-1 shows a simple user interface for a Visual Basic application that does investment calculations. This is the interface for the application that you'll develop in this chapter. It consists of a single *form*. As you can see, this form is similar to a window in any other Windows applications with a Close button in its upper right corner. In a more complicated application, the user interface may require two or more forms as you will see in chapter 3.

To complete the user interface, *controls* are added to each form. These controls let the user interact with the application. When you use the application in this figure, for example, you can enter values into the *text boxes* and you can execute commands by clicking on the *command buttons*. Other common controls are *option buttons*, *labels*, *check boxes*, *combo boxes*, and *list boxes*.

Forms and controls are two types of *objects* that you'll work with as you develop Visual Basic applications, and each of these objects has its own *properties*, *methods*, and *events*. In this chapter, you'll learn how to use some of these properties, methods, and events as you develop the application for the user interface shown in this figure.

A typical Visual Basic form



Concepts

- *Forms* provide the basis of the user interface for a Visual Basic application.
- *Controls* are added to a form so it provides the data and actions that are required. Some of the most common controls are *command buttons*, *text boxes*, *labels*, *option buttons*, *check boxes*, *list boxes*, and *combo boxes*, but many others are also available.
- Forms and controls are two types of *objects* that you work with in Visual Basic.
- An object has *properties*, *methods*, and *events*. You can set the properties of an object to control how the object looks and acts. You can invoke the methods of an object to perform actions on the object. And you can write code that responds to the events that happen to an object.

How to start a new project

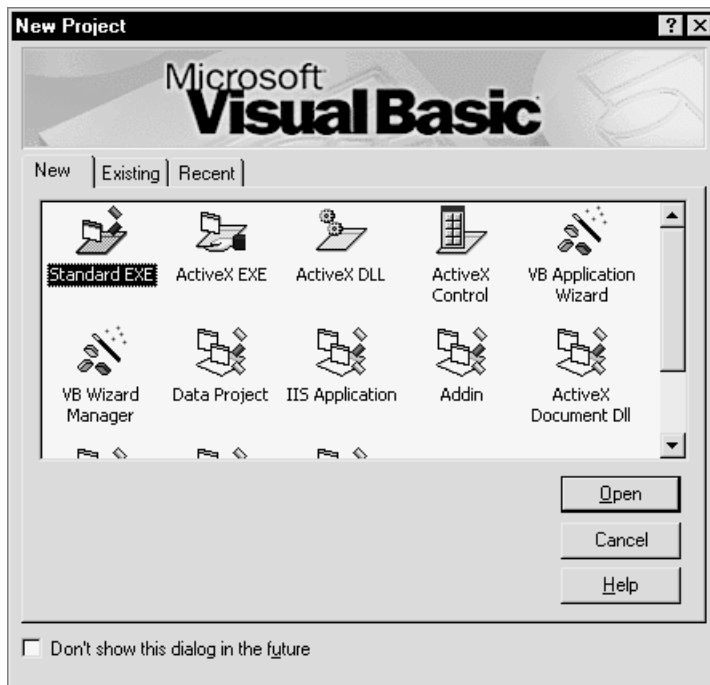
In Visual Basic, the applications that you develop are called *projects*. To start a new project, you use the New Project dialog box shown in figure 1-2. This is the dialog box that's displayed when you start Visual Basic, but a similar dialog box is displayed when you choose the New command in the File menu.

For most projects, you'll use the Standard EXE option to start a project that can be used to create an *executable file*, or *EXE*, that can be run from any machine that uses Windows 95, Windows 98, or Windows NT. When you select this option, Visual Basic starts a new project and displays a new form for it as shown in the next figure.

If you want to open an existing project right after you start Visual Basic, you can use the Existing and Recent tabs of the New Project dialog box. If, for example, you click on the Recent tab, you'll see a list of the projects that you worked on most recently. Then, you can double-click on the project you want to open. If the project you want isn't available on the Recent tab, you can click on the Existing tab to see a dialog box that's similar to the Open dialog box used in most Windows programs.

After you start Visual Basic and work on one project, you can start another new project by using the New command in the File menu. In this case, only the New tab is shown in the dialog box. Similarly, you can open an old project by using the Open command in the File menu, which shows only the Existing and Recent tabs in the Open dialog box.

The dialog box that's displayed when you start Visual Basic



Operation

- Visual Basic lets you create several different types of projects. Most of the time, though, you choose the Standard EXE option to create a standard project.
- When you start Visual Basic, the New Project dialog box lets you open a new project, an existing project, or one that you worked on recently.
- After you've worked on one project, you can use the New command in the File menu to start another project or the Open command to open an existing project.

How to work with the IDE

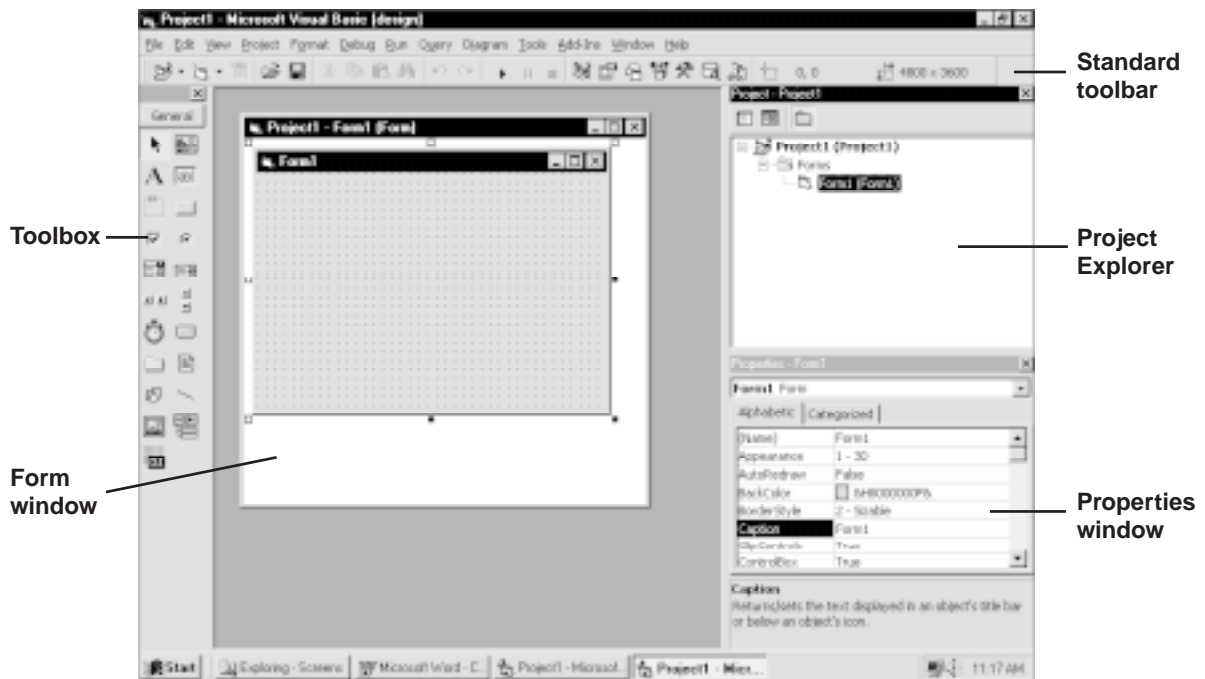
Figure 1-3 presents the Visual Basic application window. Since you use this window to perform all phases of application development, it is known as the *Integrated Development Environment*, or *IDE*. Like other Windows programs, Visual Basic provides menus and toolbars that you can use to perform its various operations. You can also right-click at appropriate places to get shortcut menus that provide context-sensitive commands.

To develop a form in Visual Basic, you work in the *Form window*. When you start a new project, a single blank form is displayed in this window. Then, you can use the *Toolbox* and the *Properties window* to develop this form. And if you add more forms to the project, you can use the *Project Explorer* to manage the forms.

By default, the Toolbox, Properties window, and Project Explorer are docked on the sides of the application window. However, they can also float in the middle of the application window. If you spend a little time experimenting with the skills described in this figure, you shouldn't have any trouble displaying, hiding, positioning, and resizing any of the windows in the IDE.

If you're familiar with other Windows applications, you should recognize some of the buttons in the Standard toolbar like the Open, Save, Cut, Copy, and Paste buttons. These buttons work much as they do in any other Windows application. If you place the mouse pointer over one of the toolbar buttons, you should see a ToolTip that gives its function. That will help you find the buttons for displaying the toolbar, Properties window, or Project Explorer.

The IDE for a new project



How to work with the Form window

- To select the form, click on it. Then, to change the size of the form, drag one of the handles that appear around it.
- To maximize or minimize the Form window, click on the Maximize and Minimize buttons.

How to work with the Toolbox, Project Explorer, and Properties windows

- To display one of these windows, you can click on the appropriate toolbar button or select the appropriate command from the View menu.
- To hide one of these windows, click on the Close button in the upper right corner of the window.
- To dock a floating window, drag it by the title bar to the edge of the window. To undock a docked window, drag it by the title bar away from the edge of the window. You can also dock or undock a window by double-clicking on the window's title bar.
- To size a window, position the mouse over the edge of the window until it becomes a double-arrow and then drag the edge of the window.

How to use shortcut menus

- When you right-click on a window or an object in a window, a context-sensitive shortcut menu is displayed. Then, you can select the command you want from that menu.

Figure 1-3 How to work with the IDE

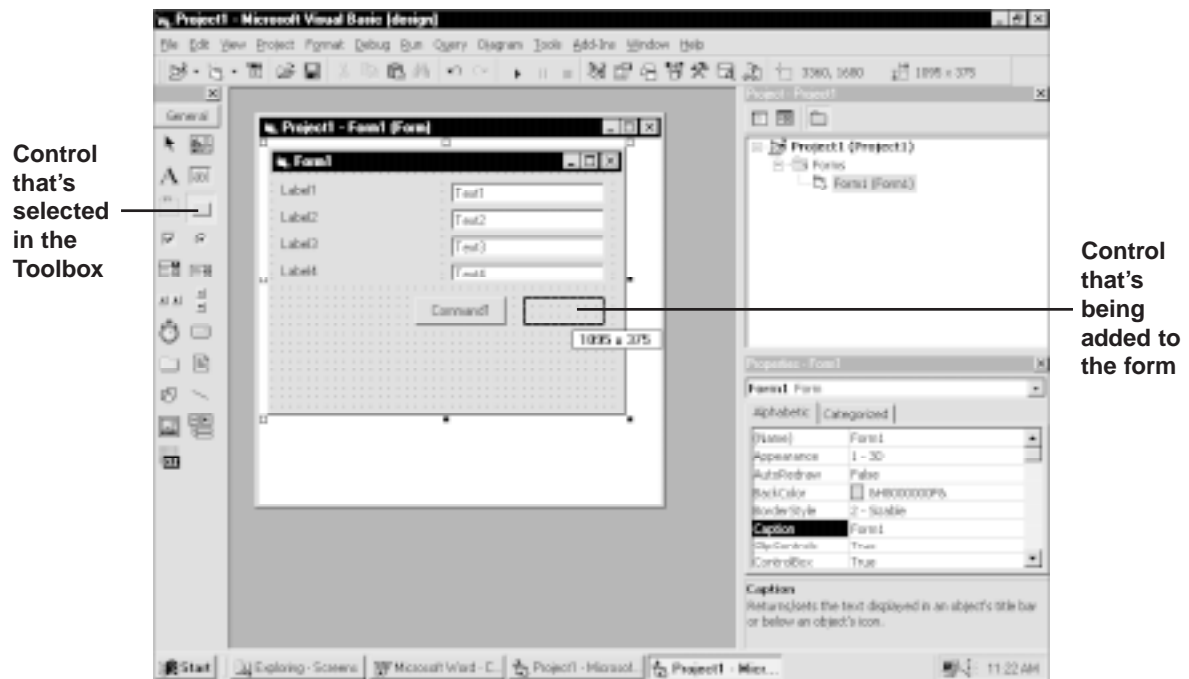
How to add controls to a form

Figure 1-4 shows how you can use the Toolbox to add controls to a form. As you can see, one way to do that is to click on the control in the Toolbox, then click and drag in the form to place and size the control. In this figure, for example, a command button is being added to the form. You can also double-click on a control in the Toolbox to add a control in the center of the form.

After you add controls to a form, you often need to size and position them. If you experiment with the skills described in this figure, you shouldn't have any trouble working with one control at a time. Working with more than one control at a time, however, can take some practice.

Let's say, for example, that you have four text box controls on your form and you want to make them all the same size with the same alignment. First, you need to select all four text box controls. To do that, you can hold down the Shift key and click on the four controls. In this case, the last control you select will be highlighted by handles that look different than the others. Then, you can use the commands in the Format menu to give the other three text boxes the same height, width, and alignment as the highlighted control.

A form after some controls have been added to it



Operation

- To add a control to a form, click on the control in the Toolbox. Then, click in the form where you want to place the control and drag the pointer on the form to size the control. (You don't just drag the control from the Toolbox to the form.)
- You can also add a control by double-clicking on it, which places the control in the middle of the form. Then, you can move and size it.
- To select a control, click on it. To move a control, drag it. To size a selected control, drag one of its handles.
- To select more than one control, hold down the Shift key and click on the controls you want to select. You can also select more than one control by clicking on a blank spot on the form and then dragging around the controls you want to select. Notice that one of the selected controls is highlighted by handles that look different from the others.
- To align or size a group of selected controls relative to the highlighted control, use the Align or Size command in the Format menu. You can also drag one of the selected controls to move all of the selected controls to a new location.

Figure 1-4 How to add controls to a form

How to set properties

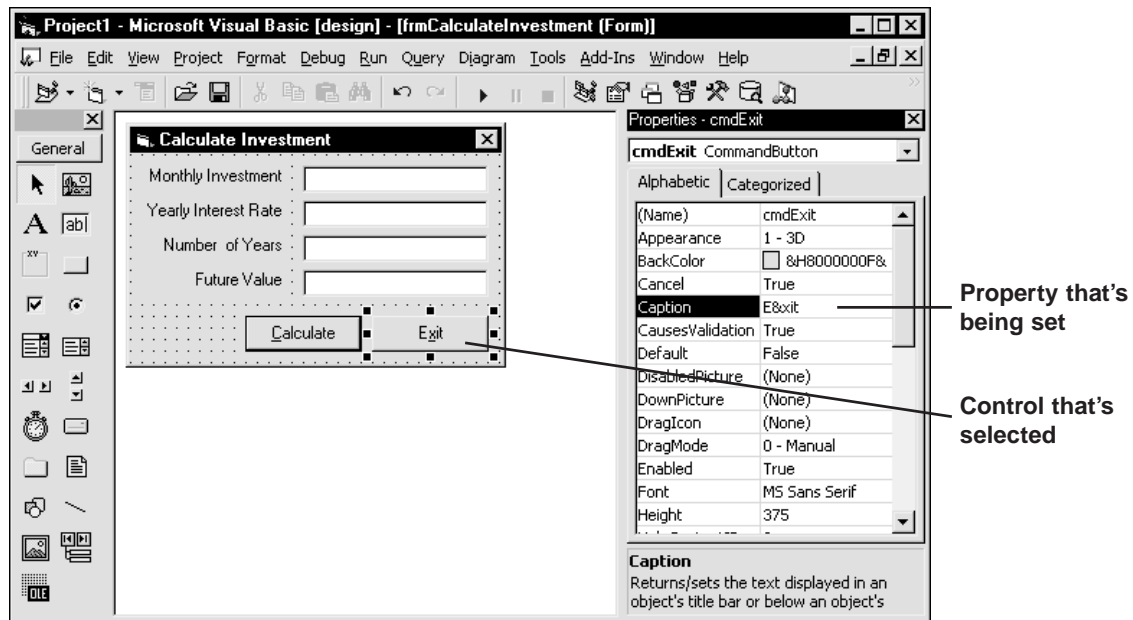
As you design a form, you set some of the *properties* for the form and its controls. That way, the form and its controls will look and work the way you want them to when the form is initially displayed.

To work with the properties of a form or control, you work in the Properties window as shown in figure 1-5. To display the properties for a specific form or control, you click on that object to select it. Then, when you click on a property in the Properties window, a brief description of that property is given at the bottom of the window. To change that property, you change the entry to the right of the property name.

To display properties alphabetically or by category, you can click on the appropriate tab at the top of the Properties window. At first, you may want to use the Categorized tab because it gives you an idea of what the different properties do. Once you get the hang of working with properties, though, you'll probably find the properties you're looking for faster by using the Alphabetic tab.

As you work with properties, you'll find that most properties are set correctly by default. In addition, many properties such as Height and Width are set interactively as you size and position the form and its controls in the Form window. As a result, you usually only need to change a few properties for each object. In particular, you usually need to change the Name and Caption properties that are described in the next figure.

A form after the properties have been set



Operation

- In the Properties window, the properties for the selected object are displayed. To display the properties for another object, click on the object in the Form window. You can also select an object from the drop-down list at the top of the Properties window.
- To change a property, enter a value into its text box or select a value from its drop-down list if it has one. If a button with an ellipsis (...) appears at the right side of a property's text box, you can click on it to get help setting the property.
- When you click on a property in the Properties window, a brief explanation of the property appears at the bottom of the window. For more information, press F1 to display the help information for the property.

Figure 1-5 How to set properties

Common properties for forms and controls

Figure 1-6 shows 17 common properties for forms and controls. The first three properties apply to forms and controls, the next six properties apply only to forms, and the last eight properties apply to some common controls. Note, however, that some of these control properties only apply to certain types of controls. That's because different types of controls have different properties.

When you set the Name property for an object, you can use lowercase letters at the start of the object name to identify the object type. For example, the *cmd* in *cmdExit* means that the object is a command button. Although you're not required to use prefixes like this when you name objects, the prefixes make your code easier to read and understand.

When you set the Caption property for some objects, such as command buttons, you can use an ampersand (&) to create shortcut keys. If, for example, you enter *E&xit* for the Caption property of a command button, *E^xit* will appear on the command button. Later on, when the application is in use, the user will be able to activate that command button by pressing Alt+x.

As you work with other properties, you'll find that you can often set them by selecting a value from a drop-down list. For example, you can select a True or False value for the Visible property of an object. Occasionally, though, you have to enter a number or text value for a property like 1 or 2 for the TabIndex property.

General properties

Property	Example	Description
Name	cmdExit	Sets the name that you use to refer to the object when you write the code for the application. You can use a three-letter prefix at the beginning of the name to identify the type of object.
Caption	E&xit	Sets the text that identifies a form or control. For some controls, you can type an ampersand (&) to underline the next letter. Then, the user can press Alt plus the underlined letter to activate the control.
Visible	True	Determines whether an object is displayed or hidden.

Form properties

Property	Example	Description
BorderStyle	2 – Sizable	Sets the border style for the form. This affects both the appearance and function of the form.
ControlBox	True	Determines whether a control box will be displayed in the upper left corner of the form.
MaxButton	True	Determines whether a Maximize button will be displayed on the form.
MinButton	True	Determines whether a Minimize button will be displayed on the form.
ShowInTaskbar	True	Determines whether a button representing the form will be shown in the Windows taskbar when the form is run.
StartPosition	2 – CenterScreen	Determines how the form will be positioned on the screen when it is run.

Control properties

Property	Example	Description
Alignment	0 – Left Justify	Sets the justification for the text within some controls such as labels and text boxes.
Cancel	True	Determines whether the control will be activated when the user presses the Esc key.
Default	True	Determines whether the control will be activated when the user presses the Enter key.
Enabled	True	Determines whether the control will be enabled or disabled.
Locked	False	Determines whether the data in some types of controls such as text boxes and combo boxes can be edited.
TabIndex	0	Sets the order in which the controls receive the focus when the user presses the Tab key. Enter 0 for the first control, 1 for the second control, and so on.
TabStop	True	Determines whether the control will accept the focus when the user presses the Tab key to move from one control to another.
Text		Sets the text that's contained in some types of controls such as text boxes and combo boxes.

Figure 1-6 Common properties for forms and controls

The property settings for the Calculate Investment form

Figure 1-7 shows a form after some controls have been added to it and the properties for the form and the controls have been set. As you can see, you only need to change four properties for the form and two or three properties for the controls to get the interface to look the way you want it to. In addition, four properties (Name, Caption, Alignment, and Text) account for all but a few of the settings.

Since the form is designed so it's the right size for the controls that it contains, you can set the `BorderStyle` property to `Fixed Single`. Then, the user won't be able to change the size of the form by dragging the edge of the form. In addition, setting the `BorderStyle` property to `Fixed Single` automatically sets the `MaxButton` and `MinButton` properties for the form to `False` so the form won't contain `Maximize` or `Minimize` buttons. As you get used to working with controls, you'll find that it's common for a change in one property setting to affect other related property settings.

The only property you may need to set that isn't in this summary is the `TabIndex` property. This property is used to determine which control will receive the focus next when the user presses the `Tab` key. By default, the tab order is set to the order in which the controls were added to the form. To change this order, you can enter 0 for the `TabIndex` property of the first control that you want the user to tab to, 1 for the second control, and so on. As you make these entries, Visual Basic automatically adjusts the remaining numbers.

Please note that the `Default` property for the `cmdCalculate` button should be set to `True`. That means that this button will be activated when the user presses the `Enter` key on the keyboard. In this case, Visual Basic places a dark outline around this button to show that it is the default. Similarly, the `Cancel` property for the `cmdExit` button should be set to `True`. That means that this button will be activated when the user presses the `Esc` key, just as though the user clicked on this button.

The form for the Calculate Investment application

The property settings for the form

Default name	Property	Setting
Form1	Name	frmCalculateInvestment
	BorderStyle	1 - Fixed Single
	Caption	Calculate Investment
	StartPosition	2 - CenterScreen

The property settings for the controls

Default name	Property	Setting
Label1	Caption	Monthly Investment
	Alignment	1 - Right Justify
Label2	Caption	Yearly Interest Rate
	Alignment	1 - Right Justify
Label3	Caption	Number of Years
	Alignment	1 - Right Justify
Label4	Caption	Future Value
	Alignment	1 - Right Justify
Text1	Name	txtMonthlyInvestment
	Text	(empty)
Text2	Name	txtInterestRate
	Text	(empty)
Text3	Name	txtYears
	Text	(empty)
Text4	Name	txtFutureValue
	Text	(empty)
	Enabled	False
Command1	Name	cmdCalculate
	Caption	&Calculate
	Default	True
Command2	Name	cmdExit
	Caption	E&xit
	Cancel	True

Figure 1-7 The property settings for the Calculate Investment form

How to save a project and its forms

When you use Visual Basic, you work with a project file and one or more form files. To save changes to your project file and all form files, you can click on the Save Project button in the toolbar. The first time you click on this button, Visual Basic prompts you with dialog boxes like the ones shown in figure 1-8. The first dialog box is used to save the form file while the second dialog box is used to save the project file. When you complete these dialog boxes, Visual Basic automatically adds the *frm* extension for form files and the *vbp* extension for project files.

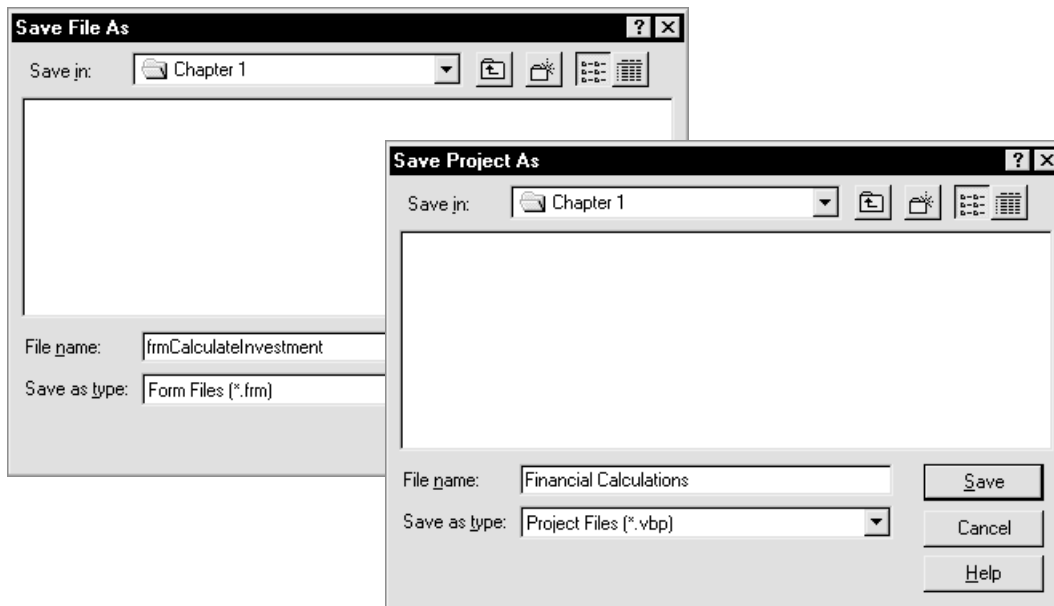
Most of the time, you'll want to save all the files for a project in their own folder. In this figure, for example, you can see that both the project file and the form file are being saved in a folder called Chapter 1. That makes it easier for you to manage the files for a project.

If you've entered a Name property for the form, Visual Basic displays that name in the Save As dialog box for the form. Since that's usually the name you want, it makes sense to set the Name property before you save the form. In this figure, for example, the form file is being saved as `frmCalculateInvestment`. On the other hand, the project file is being saved as `Financial Calculations` because the project will eventually contain another form that performs a different type of financial calculation. After you've saved a project and its forms, you can see the names you've used in the Project Explorer window.

How to test a project

When your project contains only one form, you can test that form by clicking on the Start button in the toolbar or by pressing F5 as summarized in figure 1-8. Then, the form will run in its own window. As you would expect, you can stop this form from running by clicking on the Close button in the upper right corner. You can also stop it by clicking on the End button in the Standard toolbar of Visual Basic.

The dialog boxes for saving the form and project files



How to save a project and its forms

- To save a project and its forms for the first time, click on the Save Project button in the toolbar. Visual Basic then asks you to provide a name for each form as well as a name for the entire project.
- Once saved, you can click on the Save Project button to save the updated versions of the forms and project at any time. Or, to save the changes to the current form only, you can press Ctrl+S.
- If Visual SourceSafe has been installed on your PC, a dialog box may ask whether you want the project added to SourceSafe the first time you save a project. If so, click on the No button since SourceSafe is designed for version control when two or more people are working on the same project.

How to test a project

- To start a project, click on the Start button in the toolbar or press F5. When the project consists of only one form, this starts that form.
- When a form is running, you can stop it by clicking on the form's Close button or by switching to Visual Basic and clicking on the End button in the toolbar.

Figure 1-8 How to save and test a project

Exercise set 1-1: Build the user interface

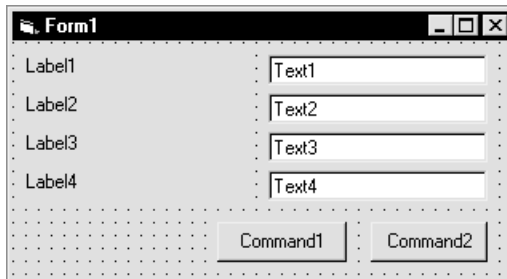
This exercise set guides you through the process of creating a user interface. As you work, you'll see how easy this is when you use Visual Basic. Then, in exercise set 1-2, you'll add the code that makes the interface work the way you want it to. And in exercise set 1-3, you'll enhance both the form and the code to make it work even better.

Start a new project

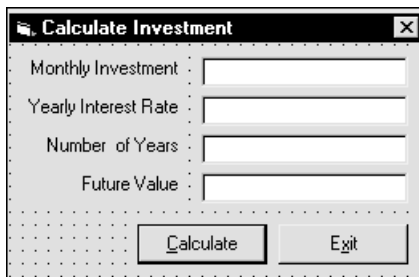
1. Start Visual Basic and begin a Standard EXE project.
2. If necessary, use the toolbar buttons to open the Toolbox, Project Explorer, and Properties window and click on the Close button of the Form Layout window. Then, arrange the IDE as shown in figure 1-3 and experiment with the techniques presented in that figure.

Add controls to the new form

3. Use the techniques in figure 1-4 to add controls to the form with approximately the same sizes and locations as this:

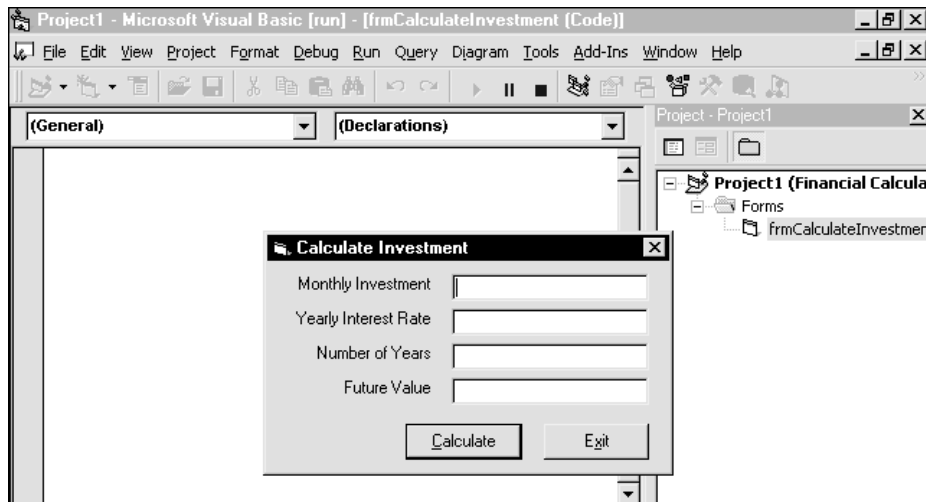


4. Use the mouse to size and position the top label control as shown above. Next, select all of the label controls by holding down the Shift key as you click on them. Then, use the commands in the Format menu to size and align them the same as the top label control. Repeat this process with the text boxes and command buttons.
5. Use the Properties window to set the properties for the form and its controls as shown in figure 1-7. Also, check the TabIndex properties to make sure the ones for the four text boxes and two command buttons are numbered from 0 through 5. When you're done, your form should look something like this:



Save and test the form

- Click on the Save Project button in the Standard toolbar to save the project and its forms in their own folder. In the first dialog box, click on the Create New Folder button and create a new folder for your project named Chapter 1 Financial. Then, save the form file as frmCalculateInvestment and the project file as Financial Calculations in this folder. If a dialog box asks whether you want to add this project to SourceSafe, click on the No button. When you're done, note that the project and form names have been added to the Project Explorer.
- Click on the Start button in the toolbar or press F5 to run the form. When you do, the Calculate Investment form will run like this:



- Experiment with the Calculate Investment form to see what it can do. When you enter text into the text boxes, notice how the text is aligned and formatted. When you press the Tab key, notice how the focus moves from one control to the other (but if the focus stops on the Future Value text box, you haven't set its Enabled property to False). When you click on the command button, notice how it indents and then pops back out just like any other Windows command button. Notice that the Calculate button has a dark outline around it to show that it is the default button (if it doesn't, you haven't set its Default property to True). As you can see, you've already accomplished a lot without writing a single line of code.
- If you notice that some of the properties are set incorrectly, click on the Close button in the upper right corner of the form to close the form. Then, make the necessary changes, and run the form again. This time, click on the End button in Visual Basic's Standard toolbar to close the window for the form. When you're satisfied that the form is working right, save all the changes you made by clicking on the Save Project button in the toolbar.

Exit from Visual Basic

- Exit Visual Basic by clicking on the Close button in the upper right corner of the Visual Basic window.

How to write the code for an interface

After you build a user interface that consists of forms and controls, you write the code that makes this interface work the way you want it to. This is the programming part of application development. The language that you use for this programming is called Visual Basic (VB). A variation of this language called Visual Basic for Applications (VBA) can be used with other Windows applications like Access and Excel.

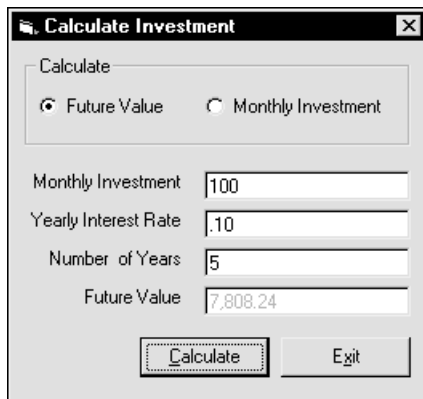
How an application responds to events

Visual Basic applications are *event-driven*. That means that they work by responding to the events that occur on objects. To respond to an event, you write code in the form of an *event procedure* as illustrated in figure 1-9. This figure shows the event procedure that's executed when the user clicks on the Exit command button on the Calculate Investment form. Between the Sub and End Sub statements that start and end the procedure, the Unload statement unloads the form.

The most common event for most controls is the Click event. This event occurs when the user clicks on an object with the mouse. In addition, three other common events are described in this figure. Note, however, that most objects have many more events that the application can respond to. For example, events occur when the user positions the mouse over an object or when the user presses or releases a key.

Note also that not all events are started by user actions. For instance, the GotFocus and LostFocus events can occur when the user moves the focus to or from a control, but they can also occur when the Visual Basic code moves the focus to or from a control. Similarly, the Unload and Load events occur when the Unload and Load statements are executed, not by a user action.

Event: The user clicks on the Exit button.



Response: The procedure for the Click event of the Exit button is executed.

```
Private Sub cmdExit_Click()
    Unload frmCalculateInvestment
End Sub
```

Common control events

Event	Occurs when ...
Click	...the user clicks on the control.
DbClick	...the user double-clicks on the control.
GotFocus	...the focus is moved to the control.
LostFocus	...the focus is moved from the control.

Common form events

Event	Occurs when...
Load	...the form is loaded but before it is displayed.
Unload	...the form is unloaded.

Concepts

- Windows programs work by responding to *events* that occur on objects. To tell your project how to respond to an event, you code an *event procedure*.
- An event can be an action initiated by the user like the Click event, or it can be an action initiated by program code like the Load or Unload event.

How to code an event procedure

To write code, you work in the *Code window* shown in figure 1-10. In this window, you can code *event procedures* for any object and event in an application. As you'll soon see, you also use this window to code other types of procedures.

From the Form window, you can start an event procedure by double-clicking on an object. Then, Visual Basic opens the Code window and generates the Sub and End Sub statements for the default event of the object. In this figure, for example, you can see the statements that are generated when you double-click on the Calculate command button. Then, you can write the code for the procedure between these statements.

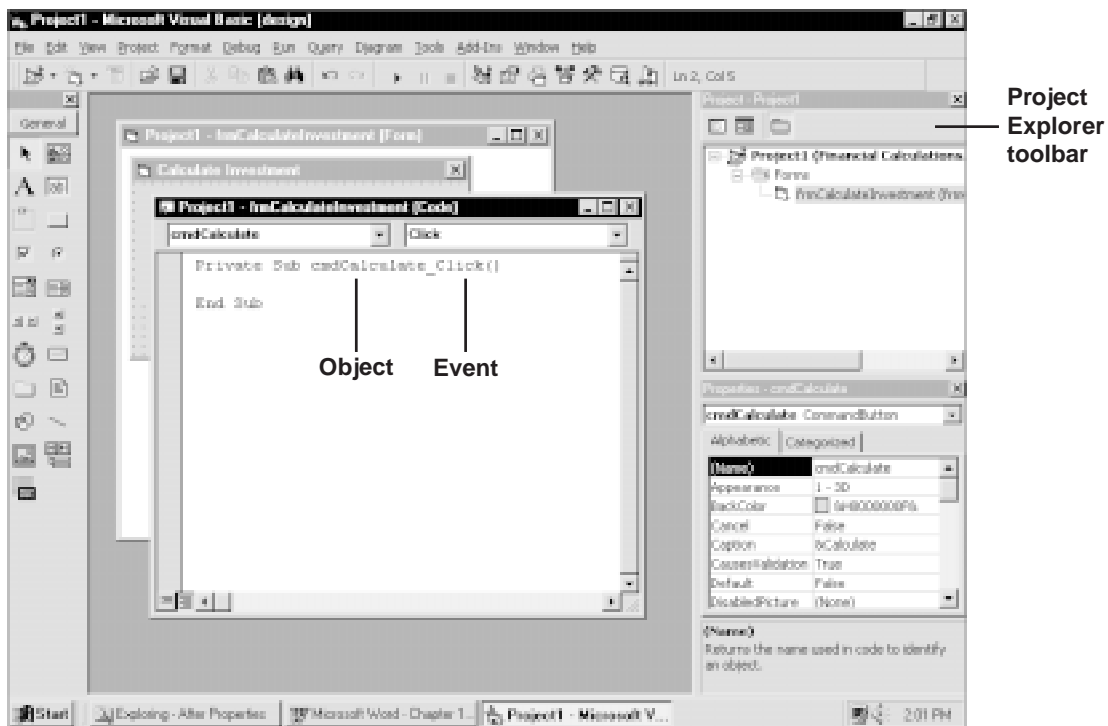
Once you're in the Code window, you can start other event procedures by selecting the object and event from the drop-down lists at the top of the window. When you choose an object, you'll see that Visual Basic selects the default event for that object and adds a Sub and End Sub statement for that event procedure to the Code window. If that's not the event you want, you can choose a different event from the drop-down list so the correct Sub and End Sub statements are added to the Code window. Then, you can delete the Sub and End Sub statements that you don't want.

When you write code that's associated with a form, the code is stored in a *form module*. In this figure, for example, the title bar of the Code window shows that the procedure belongs to the form module for the Calculate Investment form. In the next chapter, you'll learn that you can also write procedures that are stored in *standard modules*. These procedures can then be used by the procedures in other modules.

The keyword *Private* in the Sub statement in this figure means that this procedure is a *private procedure* so it can't be called from other modules in the application (although this application doesn't have any). In the next chapter, though, you'll learn how to code *public procedures* that can be called from more than one module.

The third word in the Sub statement in this figure consists of an object name, an underscore, and an event name: `cmdCalculate_Click`. This is the name of the event procedure and that's how the names of event procedures are formed. The parentheses that follow this name indicate that no *arguments* can be passed to the procedure, which is the way most event procedures are coded.

The Code window with the start of an event procedure in it



Operation

- You use the Code window whenever you write a procedure for an application.
- To display the Code window and start an event procedure for the default event, double-click on the form or control that you want to write the procedure for. Or, to start an event procedure from the Code window, choose the object and event from the drop-down lists at the top of the window. In either case, Visual Basic generates the Sub and End Sub statements for the procedure so you can write the code between those statements.
- You can also code a procedure by typing the Sub statement directly into the Code window. When you complete that statement, Visual Basic automatically drops down two lines and adds the End Sub statement.
- The name of an event procedure consists of the object name, an underscore, and the event name. For example, the procedure named cmdCalculate_Click is for the Click event of the cmdCalculate button.

Three ways to switch between the Form and Code windows

- Click on the View Object or View Code buttons in the Project Explorer toolbar.
- Use the Windows menu to switch between open windows.
- Press Ctrl+F6 or Ctrl+Shift+F6 to move to the next or previous window.

Figure 1-10 How to code an event procedure

How to code a general procedure

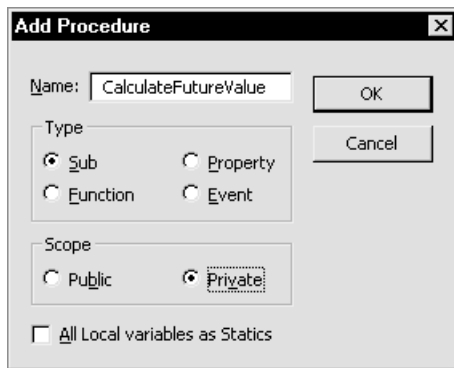
In contrast to an event procedure, a *general procedure* doesn't respond to an event. Instead, it is *called* by an event procedure or another general procedure. In figure 1-11, for example, the event procedure for the Click event of the Calculate button consists of a single statement that calls the general procedure named CalculateFutureValue.

You normally code a general procedure when its code needs to be used by more than one procedure. That way, you don't have to repeat the code in each of the procedures that require it. You'll see this illustrated later on in this chapter.

One way to start the coding of a general procedure is to use the Add Procedure command in the Tools menu to display the dialog box in this figure. Here, you can type the name of the general procedure and select the Sub type and the Private scope. Then, when you click on the OK button, the Sub and End Sub statements are generated in the Code window.

Another way to start a general procedure is to type the Sub statement directly into the Code window. When you complete that statement, Visual Basic automatically adds the End Sub statement so you can enter the procedure code between these statements.

The Add Procedure dialog box for a private Sub procedure



An event procedure that calls a general procedure

```
Private Sub cmdCalculate_Click()
    CalculateFutureValue _____ Procedure name
End Sub
```

A general procedure that's called by the event procedure above

```
Private Sub CalculateFutureValue() _____ Procedure name
    If Val(txtMonthlyInvestment) > 0 _
        And Val(txtInterestRate) > 0 _
        And Val(txtYears) > 0 Then
        txtFutureValue = _
            FV(txtInterestRate / 12, _
                txtYears * 12, txtMonthlyInvestment, 0, 1) * -1
    Else
        MsgBox "Invalid data. Please check all entries."
    End If
End Sub
```

Operation

- To code a general procedure, you can select the Add Procedure command from the Tools menu to display the Add Procedure dialog box. When you complete this dialog box, Visual Basic generates the Sub and End Sub statements so you can write the code for the procedure between those statements.
- You can also code a general procedure by typing it directly into the Code window.
- The settings in the Add Procedure dialog box above will start a *private Sub procedure*. In the next chapter, you'll learn more about that type of procedure as well as *public* and *Function procedures*.

How to refer to properties and methods

As you enter the code for a procedure, you often need to refer to the properties and methods of objects. To do that, you type the name of the object, a period (also known as a *dot operator*, or *dot*), and the name of the property or method.

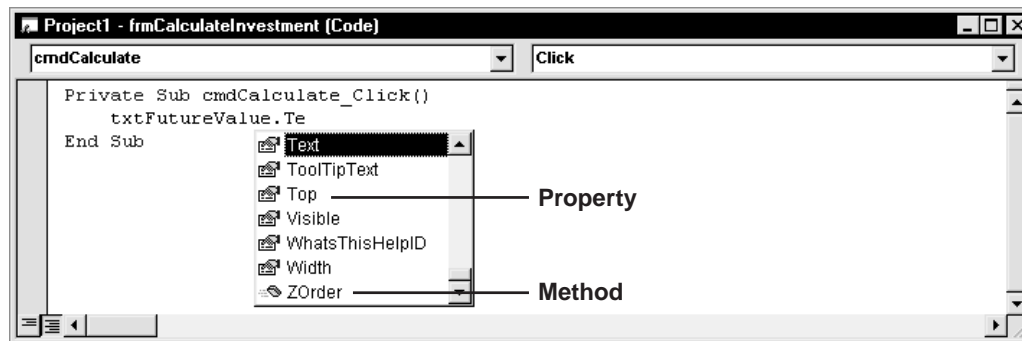
To make this easier for you, Visual Basic provides the Auto List Members feature shown in figure 1-12. After you type an object name and a dot, this feature displays a list of the properties and methods that are available for that object. Then, you can highlight the entry you want by typing the first few letters of its name, and you can complete the entry by pressing the Tab key.

To give you some idea of how properties and methods are used in code, this figure shows two examples of each. In the first example for properties, code is used to set an object's property to False. This code changes the initial value for that property that was set in the Properties window. In the second example, the value in the Text property of a text box object is tested to see whether it's greater than zero. In this case, the property name is omitted so the default property is used, which happens to be the Text property.

In the first example for methods, code is used to set the focus on a text box object. This in turn starts the SetFocus event, which can be used to start an event procedure. In the second example, code is used to hide a form. This is something you may want to do when the application consists of more than one form.

In practice, the property name is often omitted when the programmer wants to refer to the default property of an object. Although this shortens the code, it can also make it more difficult to understand because you have to remember what the default property is. For common objects, like text boxes, labels, and option buttons, though, you should remember what the default properties are so this isn't a problem.

A completion list that displays the properties and methods of an object



Statements that refer to properties

`cmdCalculate.Enabled = False`

Assigns the value “False” to the Enabled property of the command button named cmdCalculate so the button is disabled.

`If txtYears > 0 Then`

Tests the Text property of a text box named txtYears to see if it’s greater than zero. Because no property is specified, the default property (Text) is assumed.

Statements that refer to methods

`txtMonthlyInvestment.SetFocus`

Uses the SetFocus method to move the focus to the text box named txtMonthlyInvestment.

`frmCalculateInvestment.Hide`

Uses the Hide method to hide the form named frmCalculateInvestment.

The default properties of some common objects

Object type	Default property
Text box	Text
Label	Caption
Option button	Value

Operation

- To insert the correct property or method into your code, type the first few letters of the property or method and press the Tab key when the correct one is highlighted.
- Visual Basic will only display a completion list like the one shown above when the Auto List Members feature is on. To turn this feature on, select the Options command from the Tools menu, click on the Editor tab of the Options dialog box, and check the Auto List Members check box.

Figure 1-12 How to refer to properties and methods

How to use functions

To make it easy for you to do a variety of calculations and other common operations, Visual Basic provides many built-in *functions*. The FV function, for example, calculates the future value of a periodic investment. As a result, it can be used to do the calculation required by the Calculate Investment form.

When you use a function, you need to supply one or more *arguments* that provide the values that are required by the function. For the FV function, for example, you need to at least supply the interest rate, the number of periodic payments, and the amount of the periodic payments. In addition, this function provides for two optional arguments that affect the result that's returned to the program by the function.

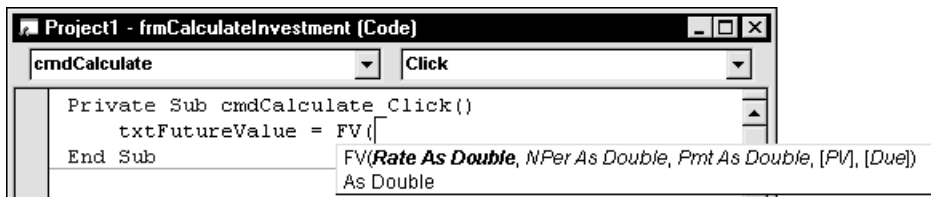
Here again, Visual Basic provides some help as you enter the arguments of a function as shown in figure 1-13. After you type the function name (in this case, FV) and an opening parenthesis, the Auto Quick Info feature gives you information about the function's arguments with optional arguments in brackets. Then, when you type the comma that's required after each argument, the next argument is boldfaced so you know where you are in the argument list. If you want to leave an optional argument at its default value, you just type a comma to move to the next argument in the list. When you complete the function by typing the closing parenthesis, Visual Basic may adjust some of the spacing before or after commas so you don't have to worry about that.

As you enter the arguments for a function like the FV function, you need to make sure that they are based on the same time period. If, for example, the periodic investment amount is for each month, then the interest rates and number of periods have to represent monthly values. To compensate for the way these values are entered into the controls of the form, you can code arithmetic expressions as shown in this figure. Here, since the investment amounts are monthly, the yearly interest rate is divided by 12 (`txtInterestRate / 12`) and the number of years is multiplied by 12 (`txtYears * 12`).

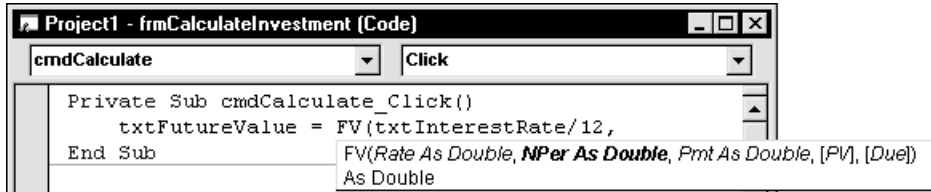
For some functions, the information supplied by the Auto Quick Info feature is all you need for entering the arguments. For other functions, though, you'll need to use the Help feature to get more information about what the arguments are. For the FV function, for example, you'll probably need to refer to the help information before you'll understand its fourth and fifth arguments. You'll also need to refer to the help information to figure out why it returns a negative value, which forces you to multiply it by -1 in order to get a positive result.

In the complete statement shown in this figure, the completed function is shaded. Here, you can see that the Text properties (the defaults) of three text boxes are supplied as the first three arguments, and the last two arguments are 0 and 1. The value returned by this function is then multiplied by -1, and this result is placed in the Text property of the Future Value text box.

The syntax for a function that accepts five arguments



The same function after the first argument has been entered



The complete statement

```
txtFutureValue = FV(txtInterestRate / 12, txtYears * 12, _
    txtMonthlyInvestment, 0, 1) * -1
```

Description

- In general, a *function* returns a value to a procedure based on the *arguments* that are sent to it. In the example above, the FV function returns its value to the Text property of the object named txtFutureValue.

Operation

- After you type the name of the function and a space or an opening parenthesis, the syntax for the function is displayed as shown above. Optional arguments in this syntax summary are enclosed in brackets [].
- As you enter the arguments for a function, Visual Basic boldfaces the current argument. To move to the next argument, type a comma.
- To get help information about the arguments for a function, press the F1 key while you're entering it.
- An argument can be an arithmetic expression as illustrated by the first argument in the example above: txtInterestRate / 12. This means that the argument will be the value in the text box named txtInterestRate divided by 12.
- Information about arguments will only be displayed when the Auto Quick Info feature is on. To turn this feature on, select the Options command from the Tools menu, click on the Editor tab, and check the Auto Quick Info box.

How to get help information

Visual Basic 6 does not use the standard Help feature that's used with other Windows programs. Instead, it uses a Help feature that's used with the Microsoft Developer Network, or MSDN. As you can see in figure 1-14, this Help feature uses the MSDN browser to navigate through the help information.

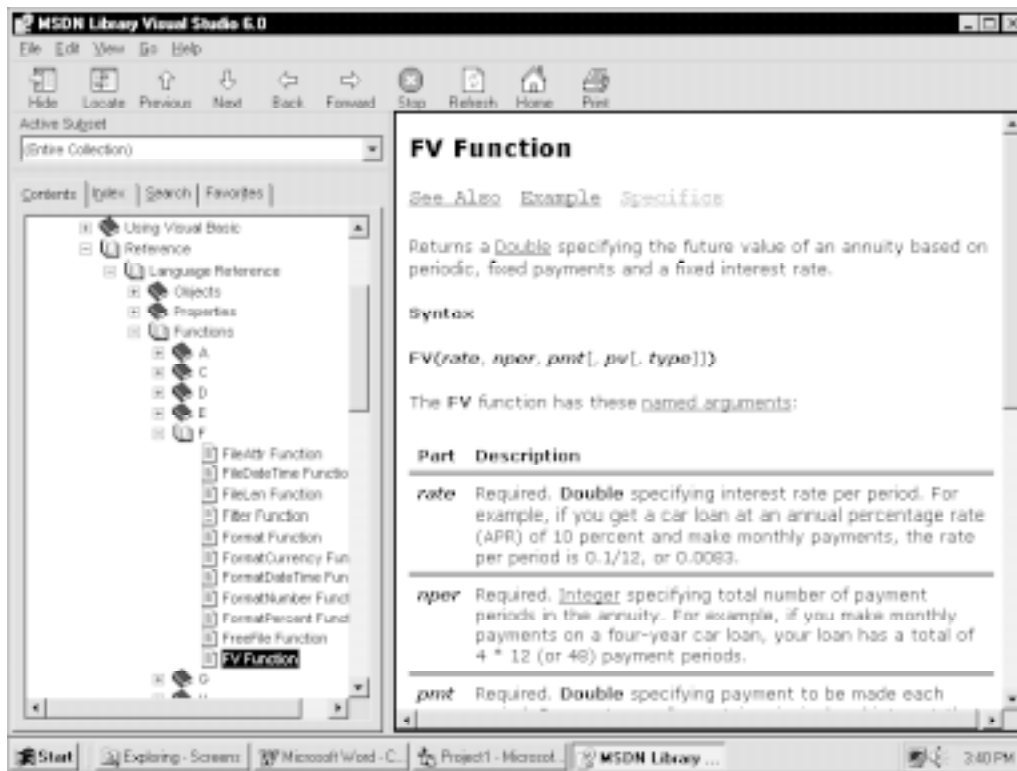
When you're working in the Code window or the Form window, the quickest way to get help information is to press F1 while the insertion point is in a word or an object is selected. This starts the MSDN browser. If the Help feature recognizes that word or object, this also displays its help information in the right pane of the browser. In this figure, for example, you can see the help information for the FV function.

Another way to access the MSDN browser is to select the Contents, Index, or Search command from the Help menu. You can also access this browser by pressing F1 when the insertion point isn't in a word that the Help feature recognizes. In either case, you can use the Contents, Index, or Search tab on the left pane to display the information you need in the right pane.

If you've used the Help features for other Windows programs, of course, you shouldn't have any trouble using this one, although it's so cumbersome that you may not like it. Otherwise, a few minutes of experimentation should get you comfortable with the use of this feature.

If you have a modem on your PC, you can use the MSDN browser to get help information from Microsoft's web site. One problem with this, though, is that once you install this browser your PC may try to connect to the Internet whenever you start your PC. This is a known problem that will be eventually be corrected, but it is nonetheless annoying.

The MSDN browser with the help information for the FV function



Description

- To display context-sensitive help information, position the insertion point in a keyword in the Code window or select an object in the Form window. Then, press F1. If the Help feature recognizes the keyword or object, the MSDN browser shown above will display the context-sensitive information.
- A second way to get help information is to select the Contents, Index, or Search commands from the Help menu. Then, you can use the Contents, Index, and Search tabs on the left pane of the MSDN browser to display help information in the right pane.
- Once you've used the MSDN browser to display a help topic, you can use the buttons in its toolbar to navigate through help topics and to print help topics. You can also click on underlined words to display more help information.
- If you have an Internet connection, you can use the MSDN browser to get information from Microsoft's web site.
- Depending on how you installed Visual Basic, the Help feature may require that you place one of the Visual Basic CD ROMs in your drive.

Figure 1-14 How to get help information

How to test and debug an application

When you develop a Visual Basic application, it's a good practice to test and debug the code each time you add a new procedure or a few related procedures. Figure 1-15 shows how. Then, when you're sure that those procedures work correctly, you code a few more procedures.

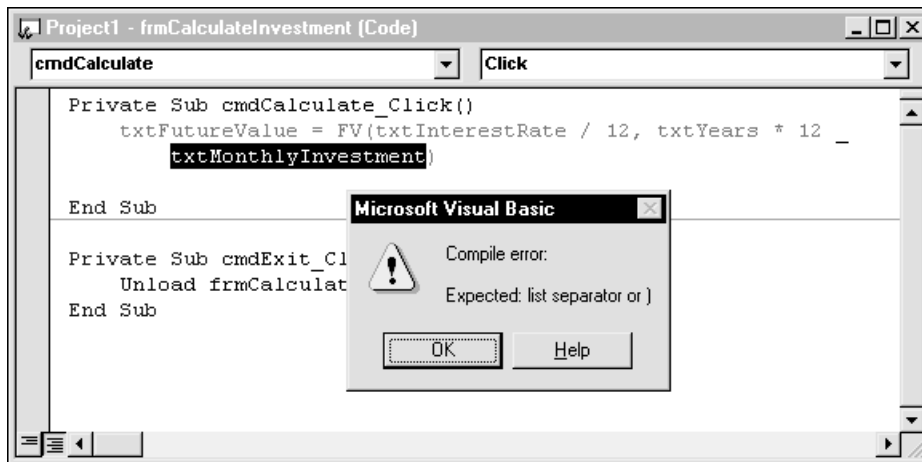
As you enter the code for a procedure and before you test it, Visual Basic checks the syntax for each statement and displays a dialog box when it detects an error. This type of error is illustrated by the first dialog box in this figure, and it is called a *compile error*. You correct this type of error as you enter the code. Can you spot the cause of the compile error in this example? (There's no comma between the second and third arguments.)

When you're ready to test the new procedures that you've added to the application, you can run the program by pressing the F5 key or clicking on the Start button in the Standard toolbar. Visual Basic then compiles and tests the application one statement at a time. If it detects an error when it compiles a statement, a dialog box for a compile-time error is displayed. If it detects an error when it runs the compiled statement, a dialog box for a *run-time error* is displayed. This type of error is illustrated by the second dialog box in this figure.

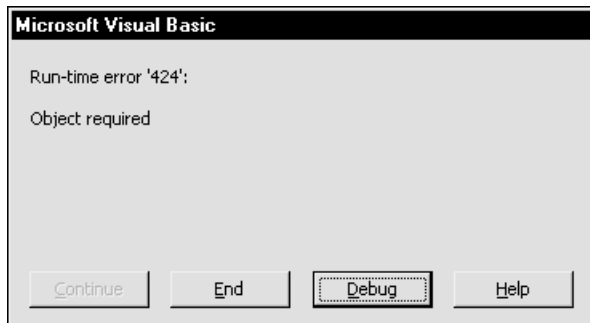
When you click on the Debug button in the dialog box for some types of run-time errors, Visual Basic switches you to the Code window and highlights the statement that couldn't be executed. This helps you figure out what the problem is so you can correct it.

All too frequently, though, the message in the dialog box is little or no help; the help information that you can get by clicking on the Help button doesn't help either; and the statement that's highlighted isn't really the statement that needs to be corrected. Beyond that, many programming errors are *logical errors* that don't show up as run-time errors. As a result, you have to find and correct those errors when you realize that the program isn't working the way it's supposed to. Fortunately, Visual Basic provides a variety of other features for debugging an application, and you can learn how to use them in chapter 4.

A dialog box for a compile-time error



A dialog box for a run-time error



Description

- When a *compile error* is detected, a dialog box like the first one above is displayed. Then, you can click on the OK button to return to the Code window and fix the error.
- When a *run-time error* is detected, a dialog box like the second one above is displayed. This means that the statement compiled okay, but an error occurred when the statement was executed. Then, if you click on the Debug button, you are switched to the statement that caused the error. Or, if you click on the End button, the project will be stopped and you will be returned to the IDE.

Exercise set 1-2: Write the code

In the last exercise set, you created the user interface for the Calculate Investment form. In this one, you will write the code for that form.

Open the project

1. Start Visual Basic and open the Financial Calculations project by double-clicking on the project name in the Recent tab of the New Project dialog box.
2. If necessary, use the Project Explorer to display the Calculate Investment form in the Form window. To do that, double-click on the Forms folder to open it, then double-click on the form name.

Code and test the event procedures

3. Double-click on the Exit command button to enter the Code window. This generates the Sub and End Sub statements for the Click event of this object. Then, you can enter the Unload statement between those statements so the entire procedure looks like this:

```
Private Sub cmdExit_Click()
    Unload frmCalculateInvestment
End Sub
```

4. From the Code window, pull down the Object list and select the Calculate command button (cmdCalculate). This generates the Sub and End Sub statements for the Click event of this object. Then, you can enter the FV function between those statements so the entire procedure looks like this:

```
Private Sub cmdCalculate_Click()
    txtFutureValue = FV(txtInterestRate / 12, txtYears * 12, _
        txtMonthlyInvestment) * -1
End Sub
```

Note here that the underscore character is used to continue the argument list from one line to the next. To get a compile error, though, omit or delete the underscore so you can see the type of message that you get. Note also that this function is multiplied by -1 so it will return a positive value.

5. Press F5 to test the application. Next, to see if the calculation works, enter numeric values in the text boxes and click on the Calculate button or press the Enter key to activate it. Assuming that the calculation does work, click on the Exit button or press the Escape key to end the program. If either of these procedures doesn't work right, of course, you need to find the bugs, correct them, and test the application again.
6. Press F5 to test the application again. This time, enter xx in the Years box or leave one of the fields blank. Then, click on the Calculate button, which will cause a run-time error. This shows that the code for this application needs to be enhanced so it checks for invalid data and advises the user accordingly. You'll learn how to do that when you return to the text. For now, though, click on the End button in Visual Basic's Standard toolbar to end the program.

Use a general procedure for the calculation

7. Select the Add Procedure command from the Tools menu. Then, enter CalculateFutureValue in the Name box, click on the Private button, and click on the OK button. This generates the Sub and End Sub statements for a general procedure.

Next, highlight the two lines of code in the cmdCalculate_Click procedure; cut them to the clipboard by using Ctrl+X or the Cut button in the toolbar; move the cursor to the blank line in the CalculateFutureValue procedure, and paste the statements into that procedure by using Ctrl+V or the Paste button in the toolbar. At this point, the entire procedure should look like this:

```
Private Sub CalculateFutureValue()
    txtFutureValue = FV(txtInterestRate / 12, txtYears * 12, _
        txtMonthlyInvestment) * -1
End Sub
```

8. Enter one statement in the procedure for the Click event of the Calculate button so it looks like this:

```
Private Sub cmdCalculate_Click()
    CalculateFutureValue
End Sub
```

9. Press F5 to test the program. If you've done everything right, the form should work the way it did before...as long as you enter valid numeric data into its text boxes. If it doesn't, you need to debug the code and test it again.

Get help information and add arguments to the FV function

10. When you return to the Code window, move the insertion point into the function name FV and press F1. This should open a Help window that tells you more about that function. Study this information to see if you can figure out what the fourth and fifth arguments do (that's not easy, is it?). With the Help window still open, modify the FV arguments so the fifth argument is 1, which means that the investment amount is deposited at the start, not the end, of each month. At this point, the function should look like this (and it will return slightly different results when executed):

```
txtFutureValue = FV(txtInterestRate / 12, txtYears * 12, _
    txtMonthlyInvestment, , 1) * -1
```

(Here, the fourth argument is omitted, which is indicated by the two commas in succession, so it's left at its default value.)

11. Return to the Help window and experiment with the Contents and Index tabs to get other information that may interest you. If you want to print a topic, click on the Print button in the toolbar. Then, close the Help window.

Save the project and exit from Visual Basic

12. Click on the Save button to save the project and its form. Then, click on the Close button in the upper right corner to exit from Visual Basic.

An enhanced version of the Calculate Investment form

In the remaining pages of this chapter, you'll learn how the Calculate Investment form can be enhanced so it does more and works better. First, you'll see how you can add three controls to it so it does two calculations instead of one. Second, you'll learn how to add code to it that checks to make sure that the users enter valid data and warns them when they don't. Last, you'll learn a quick way to create an executable file that can be distributed to the users of the application.

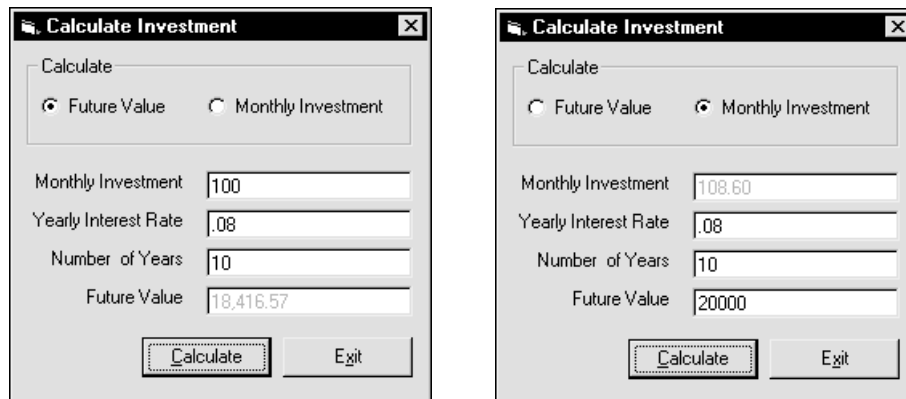
How the enhanced interface works

Figure 1-16 shows the enhanced user interface for this application. When the user clicks on the Future Value option button, the form calculates the future value of an investment based on the entries in the first three text boxes. When the user clicks on the Monthly Investment option button, the form calculates the monthly investment that is needed to attain a specific future value based on the entries in the last three text boxes.

The property settings for the new controls

Figure 1-16 also shows the property settings for the three controls that have been added to the form. Here, the Value property for the first option is set to True so it will be selected when the application starts. In addition, since the TabIndex properties for the text boxes and command buttons use the first six index values (0 through 5), the option buttons should use the next two values (6 and 7).

Two versions of the enhanced Calculate Investment form



The property settings for the new controls

Default name	Property	Setting
Frame1	Caption	Calculate
Option1	Name	optFutureValue
	Caption	Future Value
	TabIndex	6
	Value	True
Option2	Name	optMonthlyInvestment
	Caption	Monthly Investment
	TabIndex	7

Operation

- When you click on one option button within a frame, it is turned on and the others are turned off. To make this work correctly, though, you have to add the frame to the form before you put the option buttons within the frame. If option buttons aren't placed within a frame, two or more can be on at the same time.
- For this application, if you click on the Future Value option button, the Future Value text box should be disabled and the form should be used to calculate the future value. If you click on the Monthly Investment option button, the Monthly Investment text box should be disabled and the form should calculate the monthly investment amount. This means that you have to change the enabled properties of the appropriate text boxes whenever the user clicks on an option button.

Figure 1-16 The form and property settings for the enhanced interface

The code for the enhanced application

Figure 1-17 presents the code for all of the procedures in the enhanced application except the procedure for the Click event of the Exit button, which doesn't need to be changed. If you study this code, you'll see that it includes event procedures for the Click events of the two option buttons and the Calculate command button. It uses three general procedures that are called by the event procedures. And it uses the SetFocus method of two different text boxes to move the focus to them. This is the first use of *methods* that you've seen in this book, but by no means the last.

If you have much programming experience, you should be able to understand this code without much trouble. If you don't, the explanations that follow should help. To some extent, though, you won't understand the code until you read the next chapter, which presents the Visual Basic coding essentials in more detail.

The first procedure is executed when the user clicks on the Future Value option button. This procedure disables the Future Value text box, enables the Monthly Investment text box, and calls a general procedure that clears both of these text boxes. Then, it uses the SetFocus method to move the focus to the Monthly Investment text box.

The second procedure is executed when the user clicks on the Monthly Investment option button. This procedure contains four statements that are similar to the four statements used by the first event procedure.

The third procedure is the general procedure that's called by the first two procedures. This procedure clears the Monthly Investment text box and the Future Value text box by moving an empty text string into them. This type of text string is represented by two quotation marks in succession.

The fourth procedure is executed when the user clicks on the Calculate command button. It uses an If statement to call one general procedure if the FutureValue option button is selected and another general procedure if the other option button is selected.

The fifth procedure is a general procedure that performs the future value calculation. It uses an If statement to check that valid numeric values have been entered into the three enabled text boxes. To do that, it uses three Val functions, which extract the numeric values from these controls (if there are any). Then, if all three values are greater than zero, this procedure uses the FV function to perform the calculation and assign the resulting value to the Future Value text box. But if the values aren't valid, this procedure performs a MsgBox function that displays a dialog box with a message that warns the user about the invalid data and lets the user continue.

To display the result of the FV function with only two decimal places, the fifth procedure has the FV function within a FormatNumber function. In other words, one function is nested within another function. In this case, the outer set of parentheses contains the arguments for the FormatNumber function, while the inner set contains the arguments for the FV function.

The code for the enhanced application

```
Private Sub optFutureValue_Click()
    txtFutureValue.Enabled = False
    txtMonthlyInvestment.Enabled = True
    ClearTextBoxes
    txtMonthlyInvestment.SetFocus
End Sub

Private Sub optMonthlyInvestment_Click()
    txtFutureValue.Enabled = True
    txtMonthlyInvestment.Enabled = False
    ClearTextBoxes
    txtInterestRate.SetFocus
End Sub

Private Sub ClearTextBoxes()
    txtMonthlyInvestment = ""
    txtFutureValue = ""
End Sub

Private Sub cmdCalculate_Click()
    If optFutureValue = True Then
        CalculateFutureValue
    Else
        CalculateMonthlyInvestment
    End If
End Sub

Private Sub CalculateFutureValue()
    If Val(txtMonthlyInvestment) > 0 _
        And Val(txtInterestRate) > 0 _
        And Val(txtYears) > 0 Then
        txtFutureValue = _
            FormatNumber(FV(txtInterestRate / 12, _
                txtYears * 12, txtMonthlyInvestment, 0, 1) * -1)
    Else
        MsgBox "Invalid data. Please check all entries."
    End If
End Sub

Private Sub CalculateMonthlyInvestment()
    If Val(txtFutureValue) > 0 _
        And Val(txtInterestRate) > 0 _
        And Val(txtYears) > 0 Then
        txtMonthlyInvestment = _
            FormatNumber(Pmt(txtInterestRate / 12, _
                txtYears * 12, 0, txtFutureValue, 1) * -1)
    Else
        MsgBox "Invalid data. Please check all entries."
    End If
End Sub
```

Figure 1-17 The code for the enhanced application

Like the fifth procedure, the sixth procedure is a general procedure that performs a calculation after it checks for valid entries. It uses the `Pmt` function within the `FormatNumber` function and assigns the result to the `Monthly Investment` text box.

How to print the code for an application

If you want to print the code for an application, you can use the `Print` command in the `File` menu. The dialog box for this command lets you print the code for the module that you're currently working on or for the entire project. After you print the code, you can lay the pages side by side so you can review the procedures without jumping from one to another in the `Code` window. This often comes in handy when you're debugging an application.

How to create an EXE file for an application

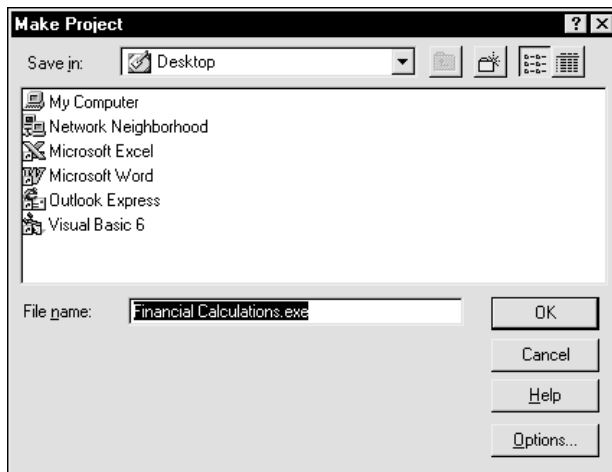
Once the application works the way you want it to, you can create an *executable* file, or *EXE*, for the application. Then, you can run the application from any computer that's running under `Windows 95`, `Windows 98`, or `Windows NT`.

To create an executable file, you select the `Make projectname.exe` command from the `File` menu, which displays the dialog box shown in figure 1-18. Then, you choose the folder that you want to save the file in and click on the `OK` button. If you choose the `Desktop` folder, this puts an icon on your desktop that can be used to start the application.

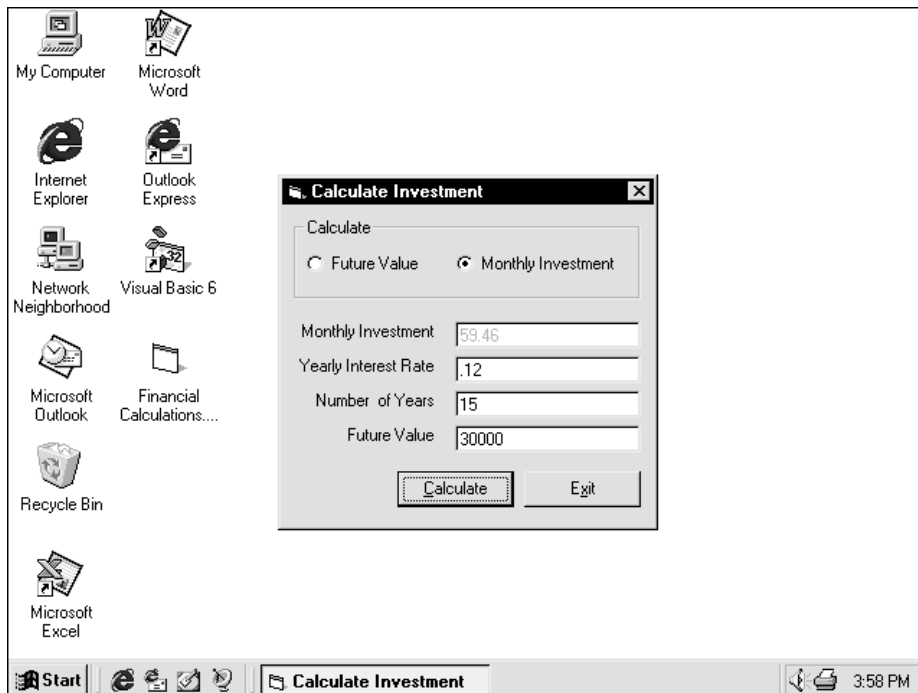
When you start the application, it runs independently, just like any other `Windows` program. In this figure, for example, you can see the application running on the desktop. But if you check the taskbar, you can see that `Visual Basic` isn't running.

For simple applications, that's all you need to do to create an executable file that can be distributed to users. For more complex applications, though, you need to create a setup program that properly installs the application and its components on each user's system. You can learn how to do that in chapter 15.

The dialog box for the Make command in the File menu



The application as it runs on a desktop



Note

- You can save an EXE file in any folder that you choose. When you save an EXE file in the Desktop folder, it is also displayed as an icon on the desktop as shown above. To delete both the icon and the EXE file, right-click on the icon and select the Delete command.

Exercise set 1-3: Enhance the application

In this exercise set, you'll enhance the application that you created in the last two exercise sets.

Open the project

1. Start Visual Basic and open the Financial Calculations project you created in the previous exercise sets.

Enhance the interface

2. Display the Calculate Investment form in the Form window, select the form, and resize it so it is taller. Then, press Ctrl+A to select all of the controls on the form and drag the controls toward the bottom of the form so you create some space at the top of the form.
3. Add the frame control at the top of the form as shown in figure 1-16. Then, add the two option buttons within this frame. Notice that each option button includes room for a caption.
4. Set the properties for the controls as summarized in figure 1-16. If necessary, adjust the sizing and positioning of the controls. When you're done, the form should look like the one in this figure.
5. Press F5 to test the form. Click on each option button to make sure that one is turned off when the other is turned on. If they don't work right, you probably need to delete the frame and option buttons from the form and add them again, this time making sure to add the frame first and the option buttons within the frame. When you're done testing the interface, click on the Exit button to stop the application.

Enhance and test the code

6. Double click on the Future Value option button to enter the Code window and generate the Sub and End Sub statements for its Click event. Enter the code for the Future Value option button as shown in figure 1-17.
7. Select the Monthly Investment option button from the Object list in the Code window. This should generate the Sub and End Sub statements for the Click event for that option button. Next, use copy-and-paste techniques to copy the statements from the event procedure for the Future Value option button to the new procedure. Then, modify this code so it looks like the code in figure 1-17.
8. Enter the ClearTextBoxes procedure directly into the Code window by typing:

```
Private Sub ClearTextBoxes
```

When you press the Enter key, Visual Basic will add the parentheses after the Sub statement and add the End Sub statement. Then, you can code the rest of this procedure as shown in figure 1-17.

9. Select the CalculateFutureValue procedure from the Procedure list in the Code window to move to that procedure. Then, modify this procedure so it looks like the one in figure 1-17 and be sure to add the FormatNumber function around the FV function. If you get compile errors as you enter the code, be sure that the parentheses and underscores are coded correctly.
10. Press F5 to test the application. At this point, the future value calculation should work and the result should be displayed with only two decimal places. In addition, a dialog box should be displayed when you enter invalid data in one of the text boxes. When you click on the Monthly Investment button, though, the calculation shouldn't work, although the appropriate text boxes should be enabled, disabled, and cleared. If anything isn't working right, of course, you need to exit from the form, fix the bugs, and test again.
11. Code the CalculateMonthlyInvestment procedure as shown in figure 1-17. After you enter the Sub and End Sub statements, you can copy the CalculateFutureValue procedure and paste it between the Sub and End Sub statements. Then, you can modify the code as necessary.
12. Modify the event procedure for the Click event of the Calculate command button so it uses an If statement as shown in figure 1-17.
13. Test the application again. This time everything should work. If you encounter any errors, of course, you must fix them and test again. When you've got the application working right, use the Print command in the File menu to print the code. Then, note that the procedures aren't in any logical sequence.

Make and run an executable file

14. Now that the application works the way you want it to, you can create an executable file for it. To do that, select the Make Financial Calculations.exe command from the File menu. In the resulting dialog box, keep clicking on the Up One Level button until you're at the Desktop folder. Then, click on the OK button to save the executable file on your desktop.
15. Click on the Save Project button in the toolbar to save the changes to the project and the form. Then, exit from Visual Basic and minimize all of the other application windows. When you're done, you should see an icon for the executable file on the desktop.
16. Double-click on the Financial Calculations icon to run the project. Use the application and you'll see that it works just the way it did when Visual Basic was running. Now, though, the executable application can be run from any PC that uses Windows 95, 98, or NT.
17. If you want to delete this application from your desktop, you can right-click on its icon and select Delete from the resulting shortcut menu. This also deletes the EXE file from the Desktop folder.

Congratulations! You've come a long way in just 45 pages.

Perspective

If you've developed the application that's presented in this chapter, you've learned a lot. You know how to get around the IDE. You know how to build an interface that consists of forms and controls. You know how to write the code that makes the interface work the way you want it to. You know what objects, properties, methods, and events are, and you've used them in your code. You've got a pretty good idea of how to develop, test, and debug an application. Above all, you've developed a working application that is by no means trivial to prove that you've learned something.

On the other hand, you've also got a lot to learn. So in chapter 2, you'll learn more about the Visual Basic language itself. That will give you more confidence when you're writing code. In chapter 3, you'll learn how to use several more controls and two or more forms in a single application. That will help you design more elaborate user interfaces. And in chapter 4, you'll learn the testing and debugging skills that you need as your applications become more complicated.

Terms you should know

form	Project Explorer
control	event-driven
text box	Code window
command button	event procedure
option button	form module
object	private procedure
property	general procedure
method	calling a procedure
event	dot operator
project	dot
Integrated Development Environment (IDE)	function
Form window	argument
Toolbox	compile error
Properties window	run-time error
	logical error
	executable file (EXE)