Chapter 6

Formatting and Arrays

1. Formatting
2. The Car Loan Problem
3. Modules and Code Reusability
4. The Scope of Variables
5. The Amortization Table Problem
6. Indefinite Loops
7. The Line Draw Program

GOALS

After working through this chapter, you will be able to:

- Use the **Format$** string function to format numerical values, dates, and times.
- Understand the concept of reusability.
- Understand and use the concept of the scope of variables. Both the visibility and the lifetime of variables are discussed.
- Use a scrollbar as input to a program. The value of the scrollbar is used to provide a subscript for an array.
- Use an array to store information.
- Use the **Do-While** loop construct in applications.
- Use two drawing methods: Circle and Line.
- Use three mouse events: MouseDown, MouseUp, and MouseMove.
- Use arrays to store information in a binary file.

OVERVIEW

Formatting output and using arrays to store lists of values are the two major focus areas of this chapter. The Car Loan project, the Amortization Table project, and the Line Draw project all emphasize the user interface. Two of the three make extensive use of arrays. The third uses graphic methods, mouse events, arrays, and files to create a simple drawing program. You will use the techniques you learn in this chapter repeatedly.

Concepts of code reusability and the scope of variables are very important both in your further study of computer science and in the practical application of programming techniques in the workplace. Reusability implies far more than just saving time and labor by reusing portions of code; it is a programming philosophy that drives the industry.

By understanding the scope of variables used in programs, you will be able to build robust programs—in other words, programs that won't fail when used.

Formatting

This section covers how to format numbers and strings. Formatting is the process of making a number or string appear in a particular way on a form or on a printed page. For example, you could choose to format the number 331455055 in either of these two different ways:

- $3,314,550.55, as currency
- 331-45-5055, as a social security number

The underlying value—the number itself—remains the same, but it is presented differently in these two examples.

Up until this point, you have printed values as either whole numbers or decimals. You are now ready to print currency with dollar signs and two decimal places, percentages with percentage signs, and times and dates with appropriate format.

Why format? First, formatting strings or numbers in different ways changes their meaning. As you can see in the example above, formatting 331455055 as currency is an entirely different way to use the number than formatting it as a social security number. Second, formatting your output makes it easier to read and understand. As soon as users see a dollar sign, for example, they know they are looking at an amount of money.

The Format Function

The **Format$** function in Visual Basic receives values from a procedure. The procedure gets back from the function a formatted string ready for display. This versatile function can format many different

kinds of numbers, including integers, decimals, percentages, currency, strings, dates, and times.

You have used the **Format$** function before. Remember the following line of code?

```
Equation = "y = " & Format$(Slope, "Fixed") & " x + " & Format$(YInt, "Fixed
```

The first usage of the **Format$** function in the line above is passed two values: a numerical value (*Slope*) and a string literal ("Fixed"). The function formats the value of the variable *Slope*. The string constant is an instruction to the **Format$** function describing the way the value should be displayed.

According to this instruction, the value of *Slope* should be displayed with at least one digit to the left of the decimal point and two digits to the right of the decimal point. If the value is 1.234332, for example, the value returned by the **Format$** function is "1.23". If the value sent to the function is 0.0123, the value returned by the function would be "0.01".

You can set up the **Format$** function in two ways to prepare a number for display:

- Using a predefined numerical format, such as "Fixed".
- Using a string that you provide. This string specifies, digit by digit, the way the value should look when displayed.

This section explores these two options.

Using Predefined Numerical Formats

This table lists the predefined formats for numbers. To use one of the predefined formats with the **Format$** function, you supply the function with the name of the predefined format as a string. We did this in the example above when we used the Fixed predefined format.

Format Name	*Effect*
General Number	Displays number with no particular modifications
Currency	Commas to show thousands, two digits to the right of the decimal point, negative numbers in parentheses
Fixed	One digit to the left and two digits to the right of the decimal point
Standard	Commas to show thousands, two digits to the right of the decimal point
Percent	Value multiplied by 100, percent sign appended to the right, two digits to the right of the decimal point

Scientific	Shows value as a decimal number between 0 and 10 times the appropriate power of 10
Yes/No	Displays *No* if the value is 0, *Yes* if the value is not 0
True/False	Displays *False* if the value is 0, *True* if the value is not 0
On/Off	Displays *Off* if the value is 0, *On* if the value is not 0

To use the predefined formats in the table, just include them, as shown, between double quotes and send them as parameters along with the values to be formatted to the **Format$** function:

```
txtPrice.Text = Format$( Price, "Currency")
txtMass.Text = Format$( Amount, "Scientific")
txtRate.Text = Format$( IntRate, "Percent")
```

In the first sample, the value of *Price* is formatted as currency. Commas are added to indicate thousands and two digits are included to the right of the decimal point. If the value of *Price* is 13456.9011, the formatted value returned would be "13,456.90".

Similarly, if *Amount* is 0.0000012992, the Scientific format converts the value to the string "1.30E-06". This format shows the number rounded to two decimal places to the right of the decimal point.

In the third example, if the value of *IntRate* is 0.0678, the **Format$** function shown converts the value to the string "6.78%". Once again, this predefined format shows two decimal places to the right of the decimal point.

Experimenting with the Numerical Formats

The instructions that follow take you step-by-step through a program that demonstrates the different built-in formatting strings used with the **Format$** function. To experiment with the formats:

1. Start Visual Basic. If Visual Basic is already running, select New Project from the File menu.
2. Double-click on the body of the form to open the Code window.
3. In the Form_Load procedure, enter the following code:

```
Dim Number As Long
Number = 345811
```

continued

```
Debug.Print Format$(Number, "General Number")
Debug.Print Format$(Number, "Currency")
Debug.Print Format$(Number, "Fixed")
Debug.Print Format$(Number, "Standard")
Debug.Print Format$(Number, "Percent")
Debug.Print Format$(Number, "Scientific")
Debug.Print Format$(Number, "Yes/No")
Debug.Print Format$(Number, "True/False")
Debug.Print Format$(Number, "On/Off")
```

Debug.Print prints results directly in the Debug window.

4 Run the program by clicking on the Single-Step button in the toolbar, the button with the single shoe print. Arrange the Code window and the Debug window so you can see both at once. As you single-step through the program, the next line of code to be executed will be highlighted. The results of the previous line will be visible in the Debug window.

If you instead run the program by pressing F5 or clicking on the Run button in the toolbar, the output appears in the Debug window almost instantaneously. It's hard to tell which line of code produced which line of output.

5 Stop the program by clicking on the Stop button in the toolbar.

Creating Formats Manually

As convenient as the predefined formats are, you may not find one that fits your needs in a specific program. For instance, you may want to include more than two decimal places in a percentage. You can do that by creating a custom format for the percentage. Visual Basic uses symbols to represent the ways digits and other symbols appear when formatted and displayed. Here's an example of a custom format:

```
txtAmount.Text = Format$(Amount, "##0.####%")
```

Each number sign, "#", represents a digit in the final string. With this format, 0.00012933 would be displayed as 0.0129%. The two number signs at the beginning of the format string are unused. They are needed only when the final value has two or three digits to the left of the decimal point.

The "0" immediately to the left of the decimal point also represents a digit. If the value being converted is greater than 1, this 0 is replaced with a digit. If the value is less than 1, Visual Basic places a 0 in the final string. Thus, the "0" is different from the "#", because a digit always appears in this position, to the left of the decimal place.

The number signs to the right of the decimal point indicate how many digits should be displayed on that side. In this case, there are four places to the right of the decimal point. Therefore, the converted value is rounded to the fourth decimal place.

Not only is the percentage sign appended to the final string, the value is also automatically multiplied by 100. This converts 0.05 to 5%, or 1.02 to 102%.

These **Format$** functions will be useful in the next program in the chapter: the Car Loan program.

Using Predefined Time and Date Formats

The table in this section lists the predefined formats for times and dates. The Effect column shows the result of executing:

```
Debug.Print Format$(Now,"xxx")
```

where "xxx" is the Format Name shown in the table. **Now** is the built-in function that reports the system time and date.

Format Name	*Effect*
General Date	5/19/95 11:15:30 AM
Short Date	5/19/95
Medium Date	19-May-95
Long Date	Friday, May 19, 1995
Short Time	11:15
Medium Time	11:15AM
Long Time	11:15:30 AM

NOTE:
For more about the Format$ function, search for Format in the Visual Basic Help system.

Localizing Format Strings

Microsoft products are sold throughout the world. Many countries use different formats for dates and numbers. Windows is customized to reflect national differences. **Format$** is sensitive to these differences. Strings returned by the **Format$** function match the preferences established when Windows is first installed. You can change these preferences by using the Control Panel. If you do so, you will see the strings returned by the **Format$** function reflect those changes.

Adding the Date to the Digital Clock

Here's a chance for you to go back to an old program and improve it. This is exactly the type of process professional programmers go through all the time. To experiment with the date and time formats discussed

above, reopen the Digital Clock program from Chapter 3. In the Timer event handler, you used the following line to copy the system time to **lblTime**:

```
lblTime = Time
```

The system function **Now** returns not only the system time, but the system date. This value can be substituted in the line above with a **Format$** function:

```
lblTime = Format$(Now, "xxxx")
```

Replace "xxxx" with each of the time and date format strings listed above and note the result of each. Before you leave the project, add a second label to display the date in long form. Resave the project and form file.

QUESTIONS AND ACTIVITIES

1. Write a program similar to the one used to test the numerical formats for the date and time formats. Step through the program using the Single-Step button in the tool bar.
2. Write a statement that calls the **Sqr(n)** function. (Calling a function means using it in an expression.)
3. What value is returned when this function is called?

   ```
   Format$(2342.1,"Currency")
   ```

4. What value is returned when this function is called?

   ```
   Format$(.23421,"Percent")
   ```

5. Write the statements using the concatenation operator, the ampersand, and the **Format$** function to build the following strings. Use appropriate variable names and the built-in format patterns (such as "Currency" or "Scientific").
   a) "x = 1.456E-03"
   b) "Weekly pay is $145.45"
   c) "Length is 345.22 cm."
   d) "The discount is 4.55%"
6. Given that $x = 234.5567$ and $y = 0.07886$, write the strings that result from the following statements:
   a) Format$(x, "Currency") & " per month"
   b) Format$(x, "Fixed") & " pounds per inch"
   c) "Mark it down " & Format$(y, "Percent")
   d) "It weighs " & Format$(y, "Scientific") & " grams"

7. Build a manual format string for the **Format$** function that returns "00.234" from the value 0.23411.

The Car Loan Program

Buying a car often requires borrowing money. That means that you will be paying for the car in monthly installments rather than all at once. Can you afford the car you want? What terms should you ask for on the loan? You can use a Visual Basic program to answer these questions rather than making the calculations yourself.

This program uses formatted output to show calculated information about getting a car loan. The user enters the loan amount, the length of the loan in years, and the interest rate of the loan. The program calculates and displays the monthly payment, the total interest paid on the loan, and the total amount paid.

You will learn how to use frames on forms to group objects.

Understanding Interest

If you did not have to pay interest, you could determine your monthly payment on a car easily. You would simply take the loan amount and divide by the number of payments.

Banks and other corporations, however, charge interest on loaned money. This interest is charged monthly and added to the loan amount, or principal. As a result, you have to add the interest to the principal each month before dividing by the number of payments.

This program uses a formula that combines these activities into a single expression:

- Calculating monthly interest
- Adding that interest to the loan amount
- Reducing the loan amount by the amount of the monthly payment
- Repeating the process for each monthly payment

The formula for calculating the monthly payment for original loan amount *B* is:

```
p = B*i/(1-(1+i)^(-n))
```

where *p* is the monthly loan payment, *B* is the original loan amount, *i* is the monthly (not the yearly) interest rate, and *n* is the total number of payments.

Consider the following example. The original loan amount (*B*) is 10000, the yearly interest rate (*rate*) is 0.05, and the time period for the loan is three years. These calculations are necessary:

- Monthly rate ($i = rate/12 = 0.05/12 = 0.004167$)
- Number of payments ($n = years * 12 = 3*12 = 36$)
  Using these values, the calculation for the monthly payment becomes:

```
p = $10,000*0.004167 / (1-(1+0.004167)^(-36))
p = $299.71
total payback amount: 299.71 x 36 = 10789.56
total interest: 10,789.56 - 10,000 = 789.56
```

Starting Out

To calculate a car loan, you need the user to enter three values:

- Loan amount
- Yearly interest rate
- Years covered by the loan

As you have often done in other projects, you will use textboxes for user input. You will also try something new: grouping these three textboxes together. To group the textboxes, you use a frame. Frames are captioned boxes containing controls. For a control to be inside a frame, it must be drawn in the frame. You draw the frame first, then place the controls inside it. As always, you will use labels to caption the textboxes.

A frame is a container control. When you move the frame by clicking on it and dragging, the controls that are inside it move with the frame. If you delete the frame, you also delete all the controls that it contains.

In previous projects, you have left input textboxes empty. In this project, though, you will provide initial values in these textboxes. Loading the boxes with sample data accomplishes two goals:

- The user can calculate a sample payment schedule without doing anything other than clicking on a button.
- You provide the user with a guide to the correct format for the values entered. For example, the program will return an error if the user enters commas in the loan amount. Visual Basic cannot convert a string with commas into a value.

Someone trying to decide about a car loan is interested in three pieces of information:

- Expected monthly payment
- Total interest they will pay
- Total amount they will pay over the period of the loan

You use three labels to display this information, as well as another three labels as captions. You will also group these objects in a frame, which you will name "Calculated Values".

To run the program, you need three command buttons (though you could also use a menu bar). These buttons let the user start the calculations, clear the textboxes for new values, and exit the program. The finished form is shown in Figure 6-1.

NOTE:

The program makes some intermediate calculations that are not displayed. You do not have to display all values the program produces.

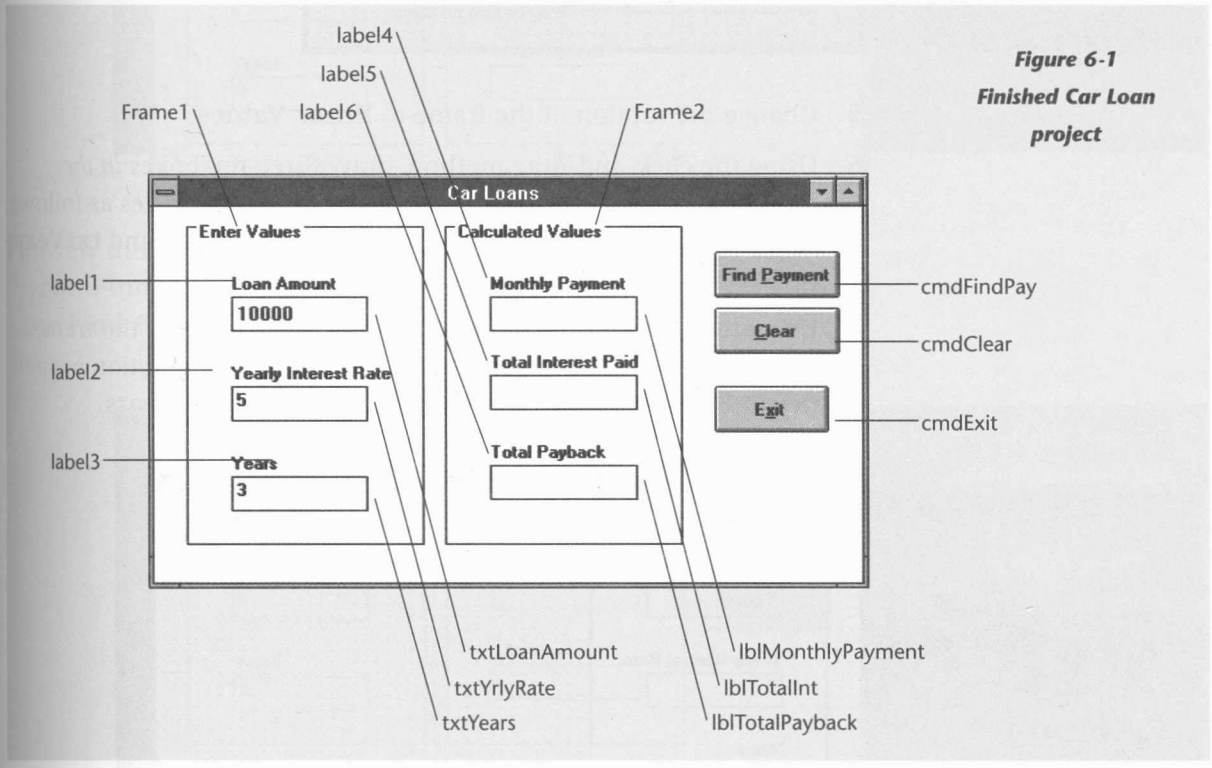


Figure 6-1
Finished Car Loan project

Setting Up the Form

Follow these steps to set up the Car Loan form:

1. Start Visual Basic. If Visual Basic is already running, select New Project from the File menu.
2. Change the caption of the form to **Car Loans**.
3. Click on the Frame icon in the Toolbox.

4 Draw the frame on the form by clicking where you want the left-hand corner of the frame. Drag to the position of the right-hand corner (see Figure 6-2).

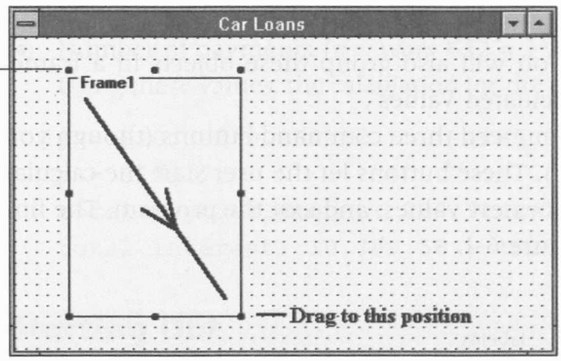


Figure 6-2
Placing a frame on the form by clicking and dragging

5 Change the caption of the frame to **Enter Values**.

6 Using the click-and-drag method, draw three textboxes *in the frame*. In the Properties window, change their properties as follows:
- ◎ Name property: **txtLoanAmount**, **txtYrlyRate**, and **txtYears**
- ◎ Text property: **10000**, **5**, and **3**

7 Using the click-and-drag method, draw three labels in the frame. Arrange the labels over the textboxes. Change the Caption properties to **Loan Amount**, **Yearly Interest Rate**, and **Years**.

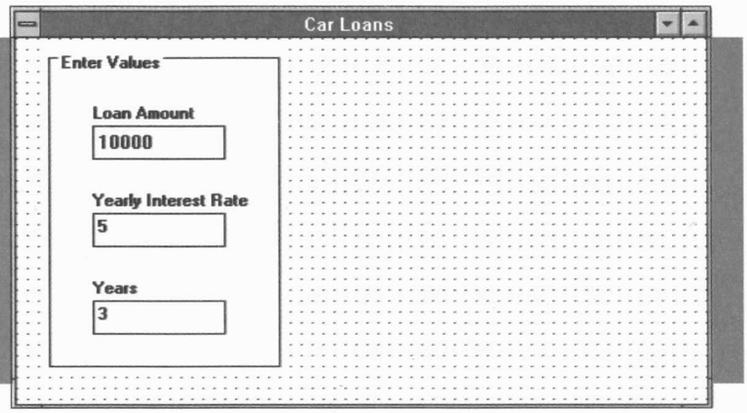


Figure 6-3
Placing labels and textboxes in the frame

8 Draw a second frame for the calculated values (monthly payment, total interest, and total payback amount). Change the Caption property of the frame to **Calculated Values**.

9 Draw three labels in the frame. Name them **lblMonthlyPayment**,

lblTotalInt, and **lblTotalPayback**. Change their BorderStyle to Single Fixed. Delete their captions.

10 Draw three labels in the frame with the captions **Monthly Payment**, **Total Interest Paid**, and **Total Payback**. Place these over the three labels created in step 9.

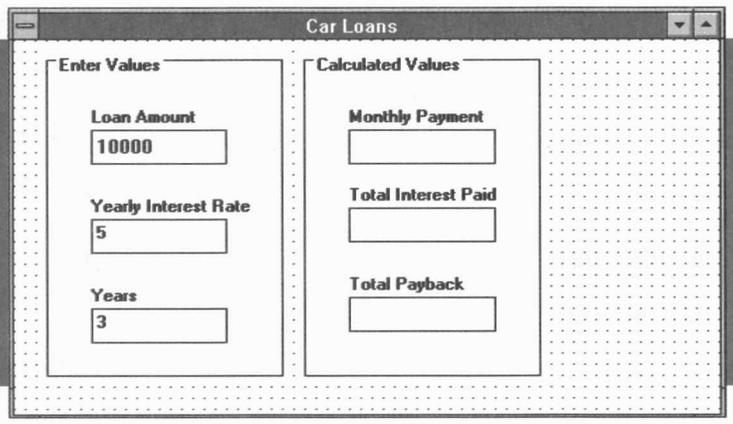


Figure 6-4
Adding more labels in a second frame

11 Draw three command buttons. Change their names to **cmdFindPay**, **cmdClear**, and **cmdExit**. Change their captions to **Find &Payment**, **&Clear**, and **E&xit**.

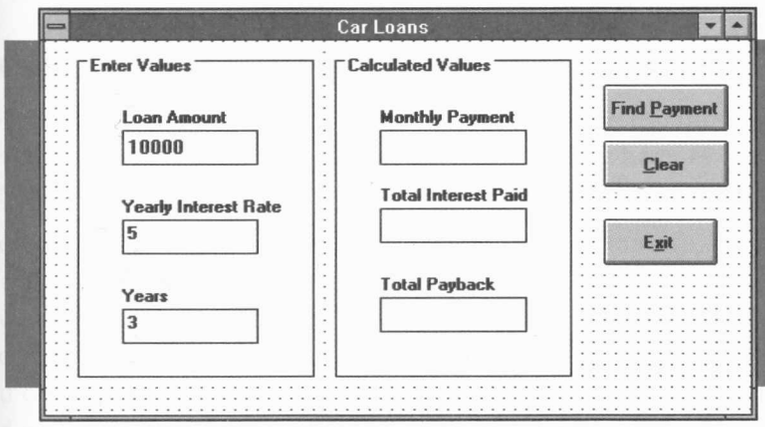


Figure 6-5
Adding command buttons to the form

Coding the Command Buttons

The work of this program is performed by the code attached to the Find Payment button. To write this code, you need to understand a few details about interest. Because interest is compounded (calculated) on a monthly basis, the interest rate used in the program is one-twelfth of the yearly interest rate. In addition, the user is expected to enter the yearly interest rate as a percentage—8.5 instead of 0.085, for example.

As a result, you need to have the program divide the interest rate by 100 to convert it to a decimal.

You want the Clear button to work in a slightly different way in this program. What if the user wants to experiment with changing just the yearly interest rate, leaving the other two values the same? The code you will write for this button does not clear the textboxes. Instead, the user can use the Delete key to change a value or two there. The Clear button clears only the labels that display the calculated values.

Follow these steps to write the code for the buttons:

1 Double-click on the cmdFindPay button to open the Code window. In the cmdFindPay_Click() procedure, enter the following code:

```
'—Variable declarations
Dim LoanAmount As Currency, MonthlyPayment As Currency
Dim TotalInt As Currency, TotalPayback As Currency
Dim YrlyRate As Single, MonthlyRate As Single
Dim Years As Integer, Payments As Integer
'—Reading values from the form
LoanAmount = Val(txtLoanAmount)
YrlyRate = Val(txtYrlyRate)
Years = Val(txtYears)
'—Intermediate calculations
MonthlyRate = YrlyRate / 1200
Payments = Years * 12
'—Monthly Payment
MonthlyPayment = LoanAmount * MonthlyRate / (1 - (1 + MonthlyRate) ^ (-Payments))
'—Total Payback
TotalPayback = MonthlyPayment * Payments
'—Total Interest Paid
TotalInt = TotalPayback - LoanAmount
'—Display results
lblMonthlyPayment = Format$(MonthlyPayment, "Currency")
lblTotalInt = Format$(TotalInt, "Currency")
lblTotalPayback = Format$(TotalPayback, "Currency")
```

2 In the cmdClear_Click() procedure, enter the following code.

```
'—Clearing the Display labels
lblMonthlyPayment = ""
lblTotalInt = ""
lblTotalPayback = ""
'—Set focus back to the Loan Amount
txtLoanAmount.SetFocus
```

3 In the cmdExit procedure, enter the instruction **End**.

Finishing Up

Now you are ready to test the program. To do so:

1 Save the project and form files.

2 Run the program and experiment with different values for the loan amount, yearly interest rate, and duration of the loan. Try some of the values in the table below. Note: some of the values below will cause the program to stop with a run-time error. How would you change the program to prevent the errors from occurring?

Loan Amount	*Yearly Interest Rate*	*Duration*
30000	8%	4
30000	9%	4
30000	8%	3
30000	8%	5
150000	8%	30
150000	8%	15
30000	-12%	5
30000	8%	-3
10	8%	5

QUESTIONS AND ACTIVITIES

1. When you are placing objects in a frame, why can't you simply double-click on the objects in the Toolbox? Why do you have to use the click-and-drag method?

2. Use the Car Loan program to find the approximate loan amounts that satisfy the given situations:
   a) You can afford $150 a month, the yearly interest rate is 12%, and you want a 3-year loan.
   b) You can afford $250 a month, the yearly interest rate is 9%, and you want a 4-year loan.
   c) You can afford $275 a month, the yearly interest rate is 15%, and you want a 5-year loan.

3. Set up the following default values for the Car Loan program:

```
txtLoanAmount = "15000"
txtYrlyRate = "8.5"
txtYears = "6"
```

   Run the program and observe the results.

4. Add the lines and controls to calculate and display the ratio of the total payback amount to the original loan amount.
5. Rewrite the Car Loan program to enter the yearly percentage as 0.07 instead of 7%.
6. An accelerator key is the underlined character in the caption of a command button. Pressing Alt + character executes the command. Change the accelerator key in the Find Payment button caption to the letter "F".
7. Rewrite the code attached to the Clear button to clear the input boxes as well as the display labels.
8. The formula used in the Car Loan project works just as well for home mortgages. Run the program and enter the following data:
   Loan amount: 180000
   Yearly interest rate: 7.5
   Years: 30
   Find and record the monthly payment, the total interest paid, and the total payback.

Section

Modules and Code Reusability

Every programmer wants to write more code in less time. "Maximum effect, with minimum effort" is an old saying most students would embrace. One of the keys to this kind of productivity is reusing code. Why rewrite code you have already written once?

You ran into this issue in the Who's in Sports? program in Chapter 5. There, you had to copy and paste sections of code twice to finish the subroutine for the Open and Display command. Each time you pasted the section of code, you had to change the names of variables, but you did not have to retype all the code.

In a sense, this is reusing code; but the approach is fairly weak and fragile. There are many opportunities for error. What if you forget to change all the variable names? This approach can also be tedious, depending on how many changes you have to make each time you paste the code.

Reusability is a far more powerful concept than cutting and pasting. The idea is to separate the code that does not change from the code that does. Then, you can reuse these unchanged "modules" of code again and again just by including these modules in your projects.

If you think about it, you'll see that the objects you place on forms are also an example of reusability. They are ready-made components that you can simply "plug in" to your forms in order to obtain predictable and well-tested behavior. Controls let you reuse someone else's work.

Besides saving time, there are other advantages to reusing program components. You can create more consistent forms and programs if you use the same elements as you need them. You can also save yourself debugging time. Once you know a section of code is debugged, for example, you will not need to debug it every time you use it.

Modules

The module is the basic unit of organization in Visual Basic projects. The Project window lists all the modules and controls in the currently open project. Modules come in two types: form modules and code modules. So far, you have only used form modules. A form module contains the form, the controls, the procedures attached to those controls or to the form, and variable declarations. A project can have more than one form module.

A code module contains procedures and declarations in a single file. The procedures and variables declared in the code module are recognized (callable) throughout the project. A code module is not associated with any form—as the name implies, it contains only code. Therefore, the code it contains cannot include event procedures for particular controls (every control that a project uses is placed on some form). However, event procedures can call code in code modules.

FORM MODULES

You have already used the form module. It contains the complete layout of each form, including textboxes, command buttons, labels, menus, and so forth. It also contains the procedures and variable declarations that have been attached to the various controls. The code written for the Click event of a command button, or the code written to shift the focus from one control to another, is included in the form module file.

A project can have more than one form module. While none of your projects has used more than one form, most real-life applications use several. Each one of these forms is saved in its own file. Each one can exist without the others. Each one contains its own layout—the form itself, its controls, their placement, and other initial values of their properties—together with all the code required to make the form functional. The form module completely specifies a form and its behavior.

You can include the same form in more than one project. Of course, it only makes sense to do so with forms that are very general in purpose. For

example, suppose you created a form that allowed the user to enter a person's name, address, and phone number, and that performed all necessary error checking on the entered values. You could then easily incorporate this form into other projects. This is a powerful form of code reusability.

There are two ways to add a new form to a project. One way is to select New Form from the File menu. When you do so, a new form opens on the desktop and its name is added to the Project window. Another way to add a form to a project is to click on the New Form button in the toolbar.

When you save your project for the first time, Visual Basic will prompt you to save each form of the project, including any you have added. To specifically save a new form, select its entry in the Project window and choose Save File from the File menu. Each form can be selected from the Project window and saved with a unique name.

CODE MODULES

The code module contains no form definitions at all. It contains procedures and declarations that solve particular problems. These modules are created and added to the list of modules in the Project window by clicking on the Code Module icon in the toolbar. When you do so, the Module window opens (see Figure 6-6):

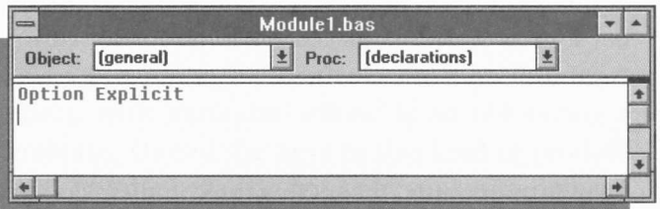


Figure 6-6
Module window

The icon appears as a constellation of forms, and indeed, the code module is used as a common resource for forms. It contains code shared by more than one form and it ensures that the variables declared in the code module are available for use by all other modules, including form modules.

Controls

As mentioned above, controls themselves provide reusability, because they allow you to reuse someone else's work. Visual Basic ships with many useful controls, and the Professional Edition contains still more. But a Visual Basic programmer's choices do not end there. The component-based model of programming that Visual Basic offers has proved so

popular that an entire market in Visual Basic controls has sprung up to meet the demand.

Many companies sell controls for use with Visual Basic, generally at attractive prices. Many of these third-party controls are general-purpose, such as full-featured spreadsheet controls that let you enter formulas in cells and that recalculate values automatically. Other controls perform very specific tasks such as voice recognition. The number and variety of available third-party controls is quite large. In fact, after a professional Visual Basic programmer has designed a program, and before writing a single line of code unnecessarily, he or she will routinely investigate whether any third-party controls can be used to create the program.

STANDARD CONTROLS

The Visual Basic toolbox comes with a number of useful tools. You've already used several in your programs: the textbox, the label, the command button. And although it is not properly called a "tool", the Menu Design window is a very powerful part of Visual Basic. In chapters to come, you'll use several more of the standard tools that come with all versions of Visual Basic.

CUSTOM CONTROLS

You can add tools to the Toolbox by placing their files in the System directory of Windows. Visual Basic then automatically adds the new controls to the Toolbox and adds the file containing the controls to the Project window. The Toolbox expands to accommodate the new controls. You can click or double-click to place one of these new controls, just as you do with the standard controls.

For Visual Basic version 3.0 and earlier, files that contain Visual Basic controls have the extension **.vbx**. Visual Basic 4.0 introduces a new, more powerful kind of control; the files containing these controls have an **.ocx** extension. Each such file can contain one or more controls.

QUESTIONS AND ACTIVITIES

1. What is "code reusability"?
2. What is listed in the Project window?
3. How is a code module added to a project? What is the code module used for?
4. Run Visual Basic. Use the buttons in the toolbar to add two forms and two code modules to the default project. Note the appearance

of the Project window. Select Save Project from the File menu. Respond to each prompt with No.

5. Run Visual Basic. Design and print a form to collect a name and address. The form should have labeled textboxes to enter the last name, the first name, the address, the city, the state, and the zip code. Select Save File As from the File menu to save just the form file (not the project).

4

Section

The Scope of Variables

To use code modules effectively, you need to explore the concept of variables in greater depth. So far, you have only used variables within event procedures in a form module. These variables applied only to the event procedures in which they were located. As you reuse code, though, you will want access to the same variables from different event procedures, and sometimes even from different modules. How widely you can use a variable depends on the variable's scope.

A variable's scope is determined by its:

- Visibility
- Lifetime

Visibility is a question of the number of procedures, forms, and modules that can use (or "see") the variable. Look at Figure 6-7. If you use the **Dim** keyword to declare a variable in the general declarations area of a module, it is visible to all procedures in that module. If you use the **Global** keyword rather than the **Dim** keyword to declare a variable in the general declarations section of a code module, then that variable will be visible to all modules in the project. If instead, you declare the variable *within* an event procedure, as you have so far, it is visible only to that event procedure.

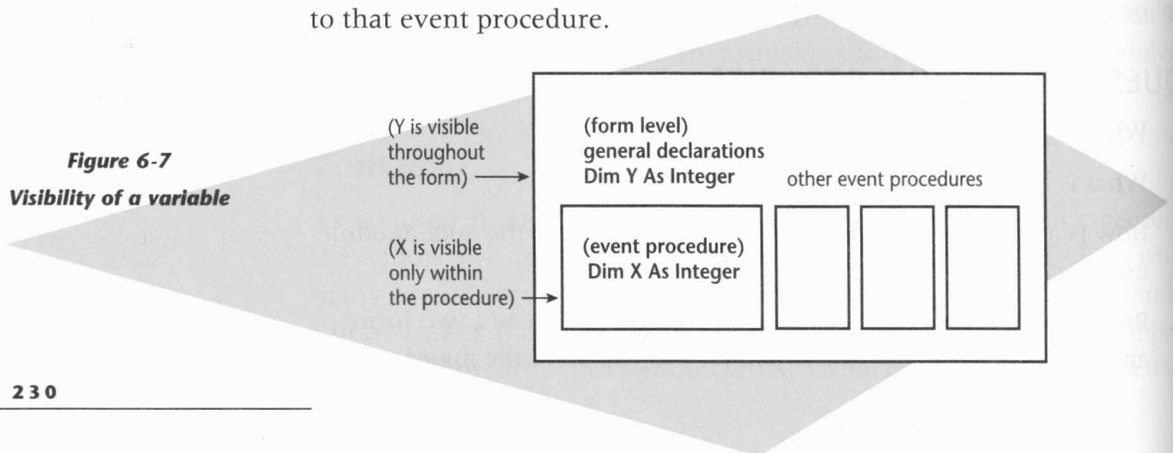


Figure 6-7
Visibility of a variable

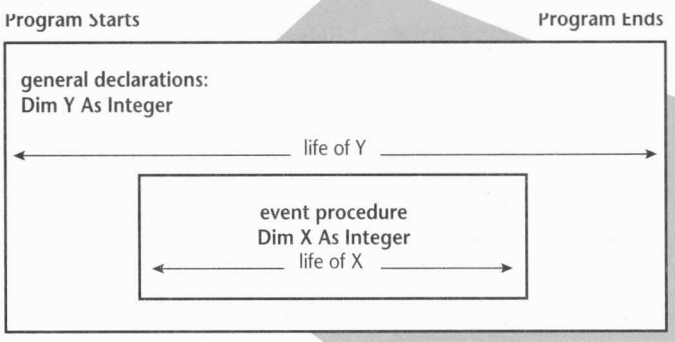


Figure 6-8
Lifetime of a variable

The lifetime of variables is determined by whether their values persist after the associated event ends. See Figure 6-8. In this example, *X* has a limited lifetime. *Y*, on the other hand, is declared in a code module and persists for the life of the program.

Any variable that you declare within a procedure using a typical **Dim** declaration will not be persistent. Sometimes, however, you do want to retain information across events. For example, suppose you wanted to count the number of times that the user clicked a particular button, in order to respond differently after the tenth click. You would need to retain a running count of the number of times the button's Click event occurred, incrementing it each time the procedure was entered.

No other procedure needs to see this persistent value holding the click count, so ideally the value should not be stored in a global variable. You want the variable to have local visibility, but you want its lifetime to be the lifetime of the program. Visual Basic lets you declare variables with this kind of scope. They are called static variables.

In terms of scope, then, there are three kinds of variables:

- Local
- Static
- Global

You define the scope of variables by where you place them and the declaration statement you use. All three of these kinds of variables are discussed in this section.

Using Local Variables

By placing variables inside event procedures, you define them as local variables. They can be seen only by the event procedure in which you have placed them. The lifetime of local variables is limited as well. When the procedure is entered, each local variable is assigned a portion of memory where the value of the variable will be stored. As soon as the program executes the **End Sub** statement, the variable "disappears"—the memory that it occupied is reclaimed for use by some other procedure's local variables. The value of the variable, which was stored in that portion of memory, is lost.

Nor can you access the old value when the program once again enters the *same* event procedure. The old value of the variable, estab-

lished during the previous time through the procedure's code, remains inaccessible. A new value is assigned.

Local variables do have one important advantage: they are safe. If you use local variables, you never have to worry if a change in the variable's value will mess up a calculation in some other procedure. The value of the variable is always completely contained within the procedure.

Using Static Variables

A static variable is a kind of local variable. The scope of a static variable is in between that of a local and a global variable. A static variable can only be seen by a single event or code procedure. However, when the procedure ends, the value of a static variable is not lost. Once a static variable obtains a value within an event procedure, it maintains that value when the event procedure is called again. That means that the lifetime of the variable extends beyond the end of the event procedure.

In Figure 6-9, *Y* is declared as a static variable in an event procedure and *X* is declared as a normal local variable in the same procedure. The value of *Y* persists even thought the particular event procedure may start and stop. The value of *X*, local to the event procedure, persists only while the procedure is running. Each time the procedure runs, a new value for *X* is created.

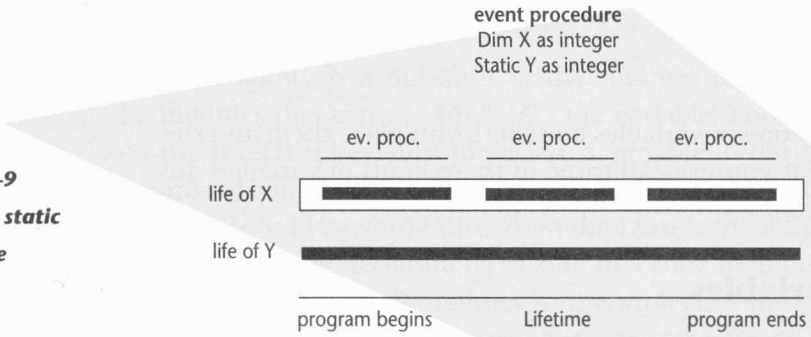


Figure 6-9
Lifetime of a static variable

How do you tell local from static variables? Both are declared in the same place, within an event procedure. However, you identify static variables by using the word **Static** in the place of the word **Dim**:

```
Static Count As Integer
```

Here's a program to illustrate the use of a static variable.

1. Start Visual Basic. If Visual Basic is running, select New Project from the File menu.
2. Change the caption of the form to **Static Variable**.

3 Place a command button on the form. Name the button **cmdCount**. Change the caption to **&Count**.

4 Place a label below the command button. Change the name to **lblHowMany**. Delete the caption.

5 Put a second label above the first, with the caption **How Many?** Change the FontSize to 9.75. See Figure 6-10.

6 Double-click on cmdCount to open the Code window. Insert this code into cmdCount_Click():

```
Static Count As Integer
Count = Count + 1
lblHowMany = Str$(Count)
```

7 Run the program. Click on the Count button several times and note the result.

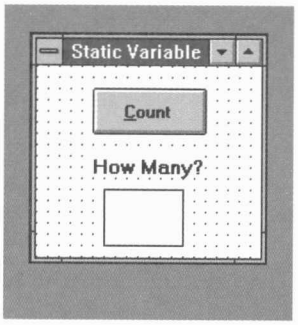


Figure 6-10
Static Variable form with command button and two labels

The first time the program executes the Click event procedure, *Count* is assigned the value 0. Before *Count* is displayed, its value is increased by 1. This value of *Count* is then displayed in the label, lblHowMany. See Figure 6-11.

Every time you click on the Count button, a new number appears in the label. Each new number is one higher than the old.

Now, make *Count* a local variable and watch the result:

1 Halt the program with the Stop button on the toolbar.

2 Save the project and form files.

3 Replace the **Static** keyword with **Dim**.

4 Run the program. Click on the Count button several times and note the result.

5 Halt the program with the Stop button on the toolbar.

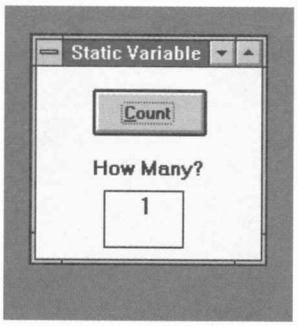


Figure 6-11
Displaying the value of the Count variable

Just like the **Static** statement, the **Dim** statement declares *Count* to be of Integer type. It is now no longer static, though, so its value will not persist. Each time you (or a user) clicks on the command button, the program creates a new instance of the variable *Count*. Each time, the variable is assigned the value of 0. Then, the code in the Click routine adds 1 and assigns the value to the textbox. Clicking on the command button does not change the value displayed in the label.

One approach that won't work is removing the **Static** or **Dim** statement altogether. *Count* is a property of the form that indicates the number of controls. Removing the declaration, either the **Static** or the **Dim**, lets the value of *Count* default to this other use.

Using Global Variables

To share variables among procedures, you must use global variables. There are two types of global variables. You can create global variables whose values are available to all procedures in:

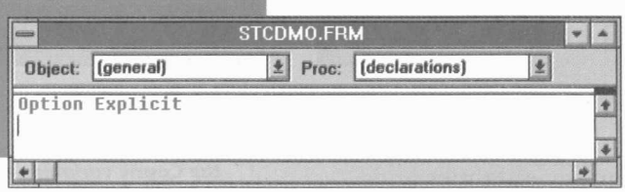
- A form module
- A code module

Any variable that is global within a form module is available to all procedures in the form. It is not, however, available to procedures in *other* forms in the project. A variable that is global to a code module can have one of two kinds of visibility. It can be available only to the procedures in that module, or it can be made available to all procedures in all forms of the project. (Variables that are visible to all procedures within a single module are sometimes said to have module scope.) You give a code module variable the second, wider kind of scope by using the **Global** keyword instead of the **Dim** keyword when you declare the variable in the general declarations section of the code module.

VISIBILITY IN A FORM MODULE

To make a variable's value accessible to all the procedures of a form module, you need to declare that variable within the general declarations area of the form. If you click on the Code window and scroll to the top of the Object list, you will see the right entry (general). Once you select that entry, Visual Basic displays (declarations) in the Proc drop-down list. See Figure 6-12.

Figure 6-12 Declaring a variable in the general declarations area of the form



Now that you have the general area open, you are ready to declare variables with global scope within the form module. You cannot define static or local variables here. A static variable declared here would simply be a global variable, so Visual Basic would not let you use the **Static** keyword in this section. Declaring a local variable in the general declarations section makes no sense: what would it be local to? In any case, there is no way to do so, because the syntax you use to declare a local variable within a procedure is the same as the syntax you use to declare a global variable in a form's general declarations section. You can experiment with the Static Variable demonstration program you just created.

Before defining the *Count* variable as global, you need to change its name. Count is a property of the form that indicates the number of controls on it. You cannot place a variable with this name in the general area of the Code window. This was not a problem so long as *Count* was only a local variable.

NOTE:

Notice that you do not use the word "global" when you declare a global variable in a form module. That word is reserved for declaring global variables with project scope in code modules.

1 Double-click on the Count button to open the Code window. In the cmdCount_Click() procedure, comment out both the statements that declare the *Count* variable.

2 Change the name of the variable from *Count* to *ClickCount*. The code now reads as shown in Figure 6-13.

3 Select *(general)* from the Object drop-down list. In this section, declare the variable *ClickCount* as an integer (see Figure 6-14).

4 Run the program. Click on the Count button several times and note the results.

5 Halt the program with the Stop button in the toolbar.

The variable *ClickCount* is declared in the general declarations section of the form module. Values assigned to the variable in one procedure are available to any procedure throughout the form module.

Figure 6-13
Code for cmdCount with the name of the Count variable changed

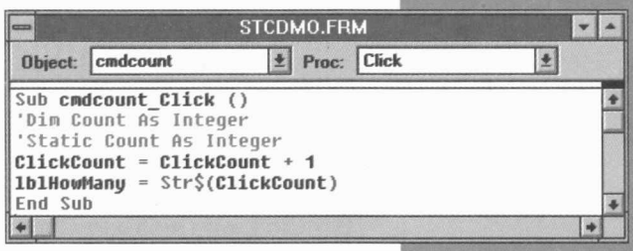


Figure 6-14
Declaring ClickCount as an integer

VISIBILITY IN A CODE MODULE

Any variables you declare in a code module are visible to all procedures in any form included in the project. So far, you have not used code modules in any of your Visual Basic programs. As an exercise, you'll create a code module for the Static Variable demonstration program.

To work with a code module:

1 Add a code module to the project. Either click on the New Module icon, or select New Module from the File menu. The Code window for the new module appears.
Note that the name of the new code module also appears in the Project window. See Figure 6-15.

2 In the code module, under **Option Explicit**, enter the statement:

```
Dim ClickCount As Integer
```

3 To return to the code for the form module, click on the form's entry in the Project window. Once the form's name is selected, click on the View Code button.

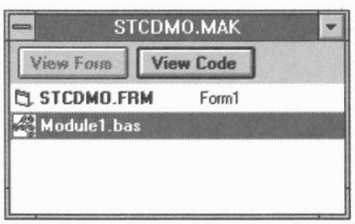


Figure 6-15
Code module in the Project window

4. Remove or comment out the **Dim ClickCount As Integer** statement in the general declarations section of the form.
5. Run the program.
   Visual Basic complains that *ClickCount* has not been declared. The **Dim** statement in the code module declares *ClickCount* within that module. To make *ClickCount* available anywhere in the project, change **Dim** to **Global**. You will do so in steps 6 through 9.
6. Halt the program by clicking on the Stop button on the toolbar.
7. Select the code module name in the Project window.
8. Click on the View Code button.
9. In the general declarations section, change **Dim** to **Global**. The line now reads **Global ClickCount As Integer**.
10. Run the program. Click on the Count button several times and note the results.
11. Halt the program.
12. Save the project file, the form file, and the code module file. Code module files are given the extension **.bas**.

QUESTIONS AND ACTIVITIES

1. What is the lifetime of a variable? What is the visibility of a variable?
2. Where are local variables declared?
3. If you change the value of a local variable within an event procedure, what changes occur outside that procedure?
4. If the **Option Explicit** statement is included in the general declarations area of a form, and a variable, declared locally in procedure A is used in procedure B, what happens?
5. What is a static variable? What is the difference between a static variable and a local variable?
6. Where do you put the declarations for a global variable? Where is the value of a global variable available?
7. There are really two kinds of global variables, ones declared with the **Global** keyword and those defined with regular **Dim** statements. What are the similarities and differences between the two? Where are each declared?

5 Section

The Amortization Table Program

In many ways, the Amortization Table program is exactly like the Car Loan problem earlier in this chapter. You start with the same problem: figuring out whether you can afford a loan at a given interest rate for a given number of years. You use the same approach for prompting users to provide total loan amount, yearly interest rate, and duration of the loan. That is, you place a frame, position three textboxes in the frame, and add labels.

Just like the other program, you use a second frame for the display of calculated values. And you use three command buttons: one for calculating, one for clearing, and one for exiting. When a user clicks on the Calculate button, the program displays the monthly payment in a label within the second frame.

The new functionality in this program lies in the amortization table in the lower third of the form (see Figure 6-16). To create this table, you will use an array for the first time. Arrays are powerful and fundamental data structures that will help you solve a wide variety of problems. Though this will be your first use of arrays, it certainly will not be your last.

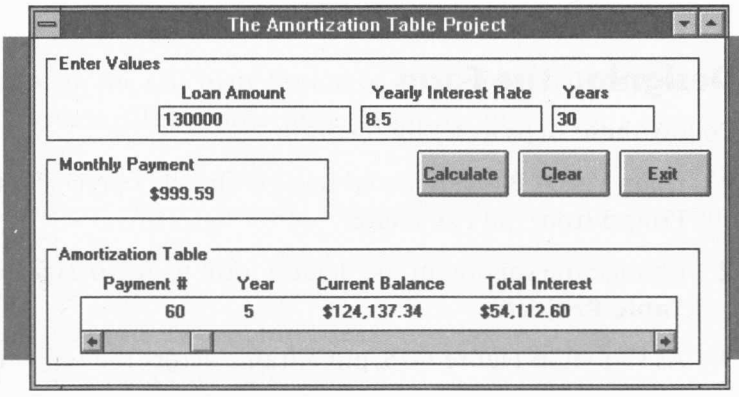


Figure 6-16 Amortization Table program

The program uses an array to store a table of values calculated as the program runs. For simplicity's sake, this array is displayed in a multiline textbox one line at a time. The user clicks on a horizontal scrollbar to determine which line of the table is displayed in the textbox.

The program is a collection of techniques that you have already learned. Frames group the values entered, textboxes collect values from the user, command buttons control the calculations, and a horizontal scrollbar controls the display of the amortization table. Using the array to store the results of the calculations needed to generate the amortization table and displaying the lines of the table in a label one at a time are new.

ARRAYS

The listboxes that you used in the Who's in Sports? program have many of the features of arrays. However, listboxes have additional features pertaining to their role as displayable interface elements. Unlike listboxes, arrays are not controls. You create them by declaring them in code. See Figure 6-17.

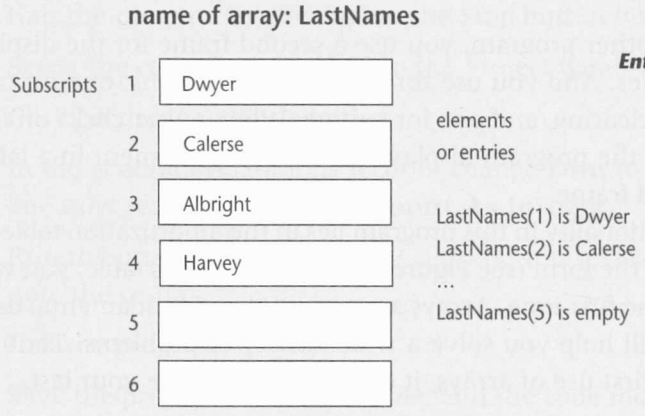


Figure 6-17
Entries in an array

ARRAYS

Arrays are lists of values. The members of the list are called the elements of the array. Arrays have names and subscripts. The name of the array is a variable name that refers to the entire list, not a single value. A subscript of the array is an integer value designating a particular value of the array.

Designing the Form

Follow these steps to set up the form.

1. Open Visual Basic. If Visual Basic is already open, select New Project from the File menu.
2. Change the caption of the default form to **Amortization Table Project**.
3. As shown in Figure 6-16, put a frame across the top of the form. Change the caption of the form to **Enter Values**.
4. Drag and drop three textboxes into the frame. Name the boxes **txtLoanAmount**, **txtYrlyRate**, and **txtYears**. Set the TabInde properties to 0, 1, and 2, respectively.
5. Drag and drop three labels into the frame. Position them above the textboxes and change the captions to **Loan Amount**, **Yearl Interest Rate**, and **Years**.
6. Draw a second frame. Change the caption to **Monthly Paymen**
7. Draw a label in the second frame. Delete the caption of the label Change the name to **lblMonthlyPayment**. Because there is no border, the label is invisible within the frame.

8 Draw a third frame across the bottom of the form. Change the caption to **Amortization Table**.

9 Draw a textbox inside the frame. Name the box **txtAmortTable**. Change the MultiLine property to True.

10 Draw a horizontal scrollbar inside the frame, below the textbox. Name the scrollbar **hsbPayment**. Moving the scroll box will change the payment displayed in the textbox.

11 Place three command buttons on the form. Name them **cmdCalculate**, **cmdClear**, and **cmdExit**. Change the captions to **&Calculate**, **C&lear**, and **E&xit**. Set the TabIndex to 3, 4, and 5, respectively.

Enter labels in the amortization table frame, above the textbox: **Payment #**, **Year**, **Current Balance**, and **Total Interest**.

Writing Code for the Command Buttons

This section of the program is much like the simple programs you wrote in Chapter 3: you handle input by reading values from textboxes on the form, calculate the monthly payment, and display the payment in a label on the form. Clicking on the Clear button prepares the form for new data.

1 Double-click on the Calculate button to open the Code window. In the cmdCalculate_Click() procedure, insert the following:

```
'-Variable declarations
Dim LoanAmount As Currency, MonthlyPayment As Currency
Dim YrlyRate As Single, MonthlyRate As Single
Dim Years As Integer, Payments As Integer
'-Reading values from the form
LoanAmount = Val(txtLoanAmount)
YrlyRate = Val(txtYrlyRate)
Years = Val(txtYears)
'-Intermediate calculations
MonthlyRate = YrlyRate / 1200
Payments = Years * 12
'-Monthly payment
MonthlyPayment = LoanAmount * MonthlyRate / (1 - (1 + MonthlyRate) ^ (-Pay-
ments))
'-Display results
lblMonthlyPayment = Format$(MonthlyPayment, "Currency")
```

2. Run the program. Enter data for the loan amount, the yearly interest rate, and the duration of the loan in years. Click on the Calculate button. The monthly payment appears in the frame, but the code is not yet complete.
3. Before finishing the code for the Calculate button, enter these lines for the Clear button.

```
'—Clearing the Display labels
lblMonthlyPayment = ""
'—Set focus back to the Loan Amount
txtLoanAmount.SetFocus
```

4. Enter **End** in the subroutine for the Exit button.
5. Save the project and form files.

Implementing the Table

So far, this program is almost identical to the Car Loan program. To create the amortization table, you need to add these additional steps:

1. Declare an array that is visible to the entire form.
2. Add code for the Calculate button that creates the lines of the table, stores the lines in the array, and sets up the horizontal scrollbar.
3. Add code to the horizontal scrollbar Change event to scroll through the lines of the amortization table.

These steps are covered one by one in this section.

DECLARING AN ARRAY

Arrays are declared with **Dim**, **Static**, and **Global** statements. You include the size of the array in parentheses following the name. For example:

```
Dim LastName(20) As String
```

This line declares an array of 20 strings with the name *LastName*. The number in parentheses the name shows that you have just declared an array.

LastName is an array of strings. You can also create an array of numeric values. As with variables, you declare the data type of the array after the word "As". For example:

```
Static PriceList(100) As Currency
```

The array in the amortization program (*AmortTable*) is an array of strings. Each string is made of values joined together. The Tab character,

embedded in each string, provides spacing for the display. The Tab character, ASCII code 9, moves the display to the next print zone. These print zones, spaced about 8 characters apart, allow you to arrange display items in columns.

The values joined together in these strings result from calculations performed during run-time. The program performs these calculations whenever a user clicks on the Calculate button. The values are displayed when the Change event of the horizontal scrollbar occurs. You declare the array of strings in the general declarations area of the form to make it visible to all event procedures throughout the form. The maximum size of the table is 360 lines.

To declare the array:

1. Open the project, if it is not already open. To open the Code window, double-click on the body of the form or click on the View Code button in the Project window.
2. In the general declarations area, enter this line:

```
Dim AmortTable(360) As String
```

ADDING CODE FOR THE CALCULATE BUTTON

You are now ready to add more code for the Calculate button. This is the code that sets up the array. It is different from the code in the Car Loan program.

To add the code:

1. With the Code window open, select cmdCalculate from the Object drop-down list.
2. Add this code to the end of the cmdCalculate_Click() procedure:

```
'-Additional local declarations
Dim PaymentNumber As Integer
Dim MonthlyInt As Currency
Dim TotalInt As Currency, CurrentAmt As Currency
Dim YearNumber As Integer, DispLine As String
Const TbCh = Chr$(9) 'the tab character
'-Initialize the TotalInterest and CurrentBalance
TotalInt = 0
CurrentAmt = LoanAmount
'-Set up loop
For PaymentNumber = 1 To Payments
   '-Make calculations
```

continued

```
    MonthlyInt = CurrentAmt * MonthlyRate
    TotalInt = TotalInt + MonthlyInt
    CurrentAmt = CurrentAmt + MonthlyInt - MonthlyPayment
    YearNumber = PaymentNumber \ 12
    '—Build display line
    DispLine = TbCh & Format$(PaymentNumber, "####")
    DispLine = DispLine & TbCh & Format$(YearNumber, "#0")
    DispLine = DispLine & TbCh & Format$(CurrentAmt, "Currency")
    DispLine = DispLine & TbCh & Format$(TotalInt, "Currency")
    '—Transfer display line to array element
    AmortTable(PaymentNumber) = DispLine
Next PaymentNumber    'end of loop
'—Set up scroll bar
hsbPayment.Min = 1
hsbPayment.Max = Payments
hsbPayment.LargeChange = 12  ' one year = 12 payments
hsbPayment.Value = 1
'—Put first line of table into the textbox
txtAmortTable = AmortTable(1)
```

Here is an explanation of these new lines of code. The first five lines of code declare variables. *PaymentNumber* is the number of the payment, *MonthlyInt* is the monthly interest amount, *TotalInt* is the total of interest payments, *CurrentAmt* is the current balance of the loan, *YearNumber* is the current year of the loan, *DispLine* is the string to build output, and *TbCh* is a constant representing the Tab character.

The following lines set total interest (*TotalInt*) to 0, then set current balance (*CurrentAmt*) to the value of LoanAmount. Next comes a **For-Next** loop. The instructions within the loop will be repeated for each payment, with the number of payments setting the upper limit of the loop. More variables are declared, then the code calculates the monthly interest amount for each payment, the running total of the interest payments, the new current balance of the loan for each payment, and the current year number.

The line **DispLine = TbCh & Format$(PaymentNumber, "####")** begins the process of joining the calculated values into a single line of display. This line starts the string with the Tab character and the payment number. The next line takes the string from the previous line and appends a Tab and the year number. Next, a Tab and the current balance are appended to the string, then a Tab and the total interest. At this point, the string is complete.

The line **AmortTable(PaymentNumber) = DispLine** copies

the line into the *AmortTable* array. The subscript of the array is provided by the payment number. Each line goes into a separate position in the array.

The loop ends, and several properties of the scrollbar are initialized. The last of these is the Value property of the scrollbar, which is initialized to 1. This corresponds with the first line of the amortization table. In the last line, the textbox is initialized with the first line of the table.

ADDING CODE TO THE CHANGE EVENT OF THE SCROLLBAR

Some of the last lines of code you inserted into the cmdCalculateClick() subroutine set up the scrollbar. Now, this final line, added to the Change event handler of the horizontal scrollbar, writes a new line of the table to the textbox when the value of the scrollbar changes. The position of the scrollbox is reflected in the value of the Value property of the scrollbar. This value picks the line of the amortization table to display.

To finish the code for the Change event:

1. With the Code window open, select hsbPayment from the Object drop-down list.
2. Enter this line of code in the hsbPayment_Change() event procedure:

```
txtAmortTable = AmortTable(hsbPayment.Value)
```

Finishing the Program

To check your work:

1. Run the program. Enter data for the loan amount, the yearly interest rate and the duration of the loan in years. Click on the Calculate button. The monthly payment appears in the frame. The textbox shows the first line of the amortization table.
2. Click on the right arrow box of the horizontal scroll bar. Click and hold on the right arrow box. Click in the channel to move the display one year at a time.
3. Stop the program.
4. Save the project and form files.

QUESTIONS AND ACTIVITIES

1. What other applications can you think of, involving tables of values, that could be handled in the same way as the amortization table?

2. In the code for cmdCalculate, change the scroll bar initialization code as follows:

```
hsbPayments.SmallChange = 2
hsbPayments.LargeChange = 24
```

   Run the program and observe the results.

3. Remove the declaration for *AmortTable* from the general declarations section of the form. Move the declaration to the Calculate button and run the program. What happens?
4. Add a code module to the project. Move the **Dim** statement for *AmortTable* to the general declarations section of the code module. Run the program. What happens?
5. Change the **Dim** statement in the code module to a global statement. Run the program. What happens?
6. How is a listbox like an array?
7. What is the subscript of an array used for?
8. Write the statement that assigns a string, *Msg*, to the ninth entry of the *AmortTable* array.
9. What symbol forces Visual Basic to perform an integer division?
10. What does the Tab character do in a string being prepared for display?
11. Write a statement that would assign the fourth entry of the *AmortTable* array to the textbox named txtDisplayLine.

6

Section

Indefinite Loops

The **For-Next** loop construction has served Basic well. For a long time, it was the only built-in looping statement for Basic programs. This loop, however, has a major flaw. To use it, the number of times the loop should be executed must be known before the loop actually begins.

During design time, you may set up the number of times through a **For-Next** loop. Or this number can be determined during run-time but *before* the loop begins. For example, in the Amortization Table program, the number of times the loop executed was determined by the number of payments to be made. You did not know this number when you wrote the program, because it depended on the values entered into the

textboxes during run-time. But the number was known before the loop began. The number of payments was the number of years of the loan times 12.

There are situations, however, that require loops, but the number of times the loop is executed is not known before the loop begins. This kind of loop is called an indefinite loop. All modern programming languages provide a way to write these indefinite loops. Visual Basic uses a **Do-Loop** statement.

This program introduces or reviews three topics:

- One version of the **Do-Loop** construct
- The **InputBox** function
- The **MsgBox** statement

The Do-Loop Construct

Look at the loop in Figure 6-18. The two lines following the **Do-While** line will execute so long as *Total* < 1000. While that statement is true, the loop continues. Some calculation or input in the body of the loop signals when this statement becomes false. For an indefinite loop to work, there has to be an action, in the body of the loop, that causes the condition being tested to become false.

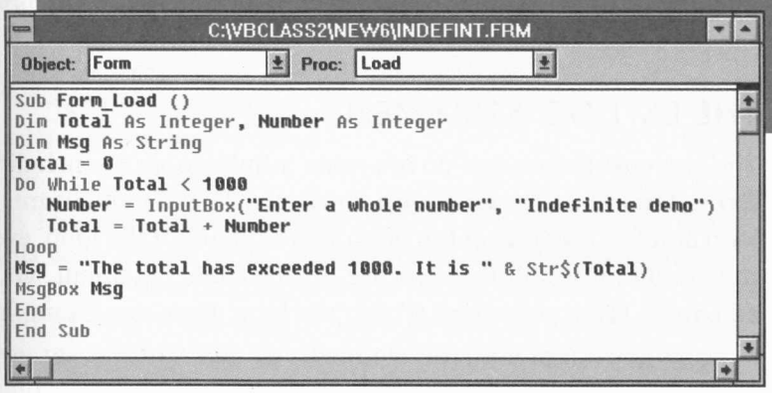


Figure 6-18
Coding an indefinite loop

As you can see from the figure, the syntax of a **Do-Loop** statement is:

Do While condition
.....Body of the loop (statements to execute as long as condition is true)
Loop

The danger with a **Do-Loop** is never getting out of it. If the condition being tested never becomes false, the loop never ends. For example:

NOTE:

If the condition tested is false, the statements of the loop are not executed even once.

```
Do While 2 < 4
   ...
Loop
```

The statements represented by the ellipsis (...) can never make 2 > 4. Therefore, the condition is always true. This "runaway loop" condition is known as an infinite loop.

The **Do-Loop** construct is actually more general than the **For-Next** loops that you have already used. The following example shows how to use a **Do-Loop** to achieve the same effect as a **For-Next** loop.

```
x = 1
Do While x <= 100
   ...
   x = x + 1
Loop
```

The loop above is a replacement for this **For-Next** statement:

```
For x = 1 To 100
   ...
Next x
```

In this example, the loop starts with $x = 1$ and the condition tests as true. While the loop is executing, the value of x is increased by 1 each time through the loop. Eventually, the value of x is no longer less than or equal to 100 and the condition tests false. When the condition is false, the loop ends.

THE EXIT DO STATEMENT

You can ensure that you do not write infinite loops by including an **Exit Do** statement in the **Do-Loop** construct. This statement interrupts the loop and lets the flow of the program jump out of the loop, even though the condition is still true. To use an **Exit Do** statement, you typically add an **If-Then** statement within the loop. Here's an example:

```
x = 1
Do While 2 < 4
   ...
   If x > 50 Then Exit Do
x = x + 1
Loop
```

The **If-Then** statement tests the condition **x > 50**. The value of x is incremented within the loop. Even though the condition of the **Do-Loop** is *always* true, when the value of x exceeds 50, the **Exit Do** state-

ment is executed. As a result, the next statement to be executed is the one following the **Loop** statement.

If you have nested **Do-Loop** statements, an **Exit Do** in the inner loop causes the program to jump out of the inner loop to the outer loop. If there are statements following the inner **Loop** statement, those statements are executed.

There are four other significant variations of the **Do-Loop** statement. Future programs use some of these forms. If you can't wait, check the Visual Basic Help system for descriptions of the other forms.

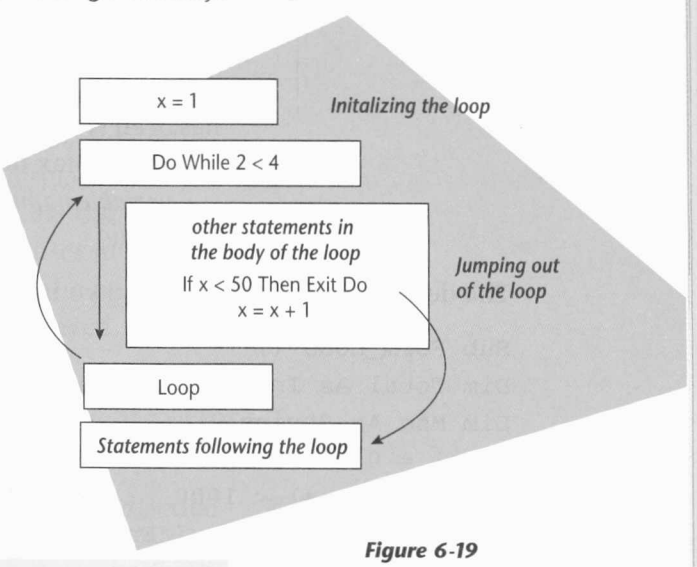


Figure 6-19
Jumping out of an indefinite loop

INDEFINITE LOOPS CONTROLLED BY INPUT

One of the common uses of an indefinite loop is to trigger the end of a series of input operations. When using **InputBox** to collect information from the user, you can designate a special input value with which the user can signal the end of the input process. Such a value, coming at the end of a list of values to signal the end of input, is called a sentinel value.

Here's a sample of code that implements this idea:

```
Msg = "Enter the age <0 to stop>"
Age = InputBox(Msg)
Do While Age<>0
   '—processes data...
   Age = InputBox(Msg)
Loop
```

This loop continues until the value of 0 is entered for the age. This code can be easily adapted for a variety of situations where keyboard data controls the flow of the program.

Sentinel values can also be used to indicate the end data that you read from a file. For example, in a file that contains a list of nonnegative numbers that you read one at a time, you could establish the convention that the value -1 indicates the end of a list.

The InputBox Function

In older versions of Basic, the **Input** statement was the main way to obtain keyboard input from the user. In Visual Basic, textboxes provide

the usual means for collecting typed information and displaying it after it has been typed. The **InputBox** function gives an alternative to textboxes.

The syntax of the **InputBox** function is:

variable name = **InputBox** (prompt string, window caption, default value)

The demonstration program shown in Figure 6-22 used an InputBox statement:

```
Sub Form_Load ()
Dim Total As Integer, Number As Integer
Dim Msg As String
Total = 0
Do While Total < 1000
   Number = InputBox("Enter a whole number", "Indefinite demo")
   Total = Total + Number
Loop
Msg = "The total has exceeded 1000. It is " & Str$(Total)
MsgBox Msg
End
End Sub
```

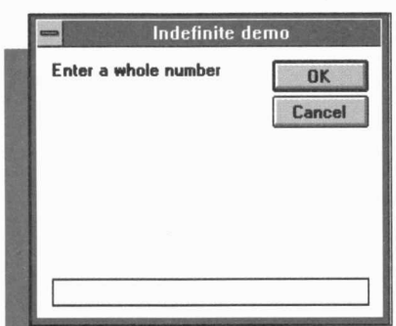


Figure 6-20
InputBox onscreen

The **InputBox** function creates a box near the middle of the screen. The caption of the box is given in the second string: "Indefinite demo". The first string is a user prompt: "Enter a whole number". See Figure 6-20 for the result of the **InputBox** statement above.

This example does not make use of the third possible parameter for the function (a default value). If you include a default value, then this value will automatically appear (see Figure 6-21). To accept this value, the user would press the Enter key.

You do not have to include a caption for the box. To omit the caption, leave the spot blank in the parameter list. The commas must appear unless both the caption and the default value are omitted. For example:

```
x = InputBox("Enter a number",,"345")
```

This statement creates the box, displays the prompting message "Enter a number", and provides a default value in the edit space. See Figure 6-21.

Figure 6-21
Inputbox without caption

The MsgBox Statement

The **MsgBox** statement provides a simple method for displaying information that you do not want the user to overlook. It displays a simple dialog box containing a message that you specify and an OK button. When your program executes a **MsgBox** statement, the user must click the OK button in order to perform any further actions in the program.

Look back at Figure 6-17. According to the code displayed there, a message box will be displayed when *Total* exceeds 1000:

```
Msg = "The total has exceeded 1000. It is " & Str$(Total)
MsgBox Msg
```

The string variable *Msg* is built from a text message and the actual total of the values. It is sent as an argument to the **MsgBox** statement. When this form of the statement is executed, a box appears near the center of the screen displaying the value of *Msg*. (see Figure 6-22). The execution of the program stops until the user clicks OK.

The **MsgBox** statement can take a number of different forms. For more information, take a look at the MsgBox Function, MsgBox Statement entry in the Visual Basic Help system.

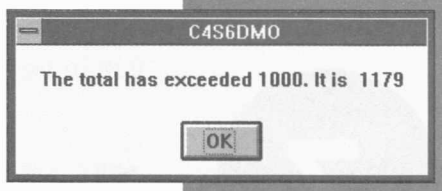


Figure 6-22
Message box on screen, awaiting user response

QUESTIONS AND ACTIVITIES

1. Write the **InputBox** statement that generates the window in Figure 6-22 and assigns the value entered to the variable *Score*.
2. Sketch the **InputBox** window that would result from this statement:

```
Age = InputBox("Enter the age",,"24")
```

3. Describe three reasons you might use a message box.
4. What happens if the condition tested in a **Do-While** loop never becomes false?
5. What's wrong with this section of code?

```
index = 0
Do While index <> 3.7
   ...
   index = index + 0.3
Loop
```

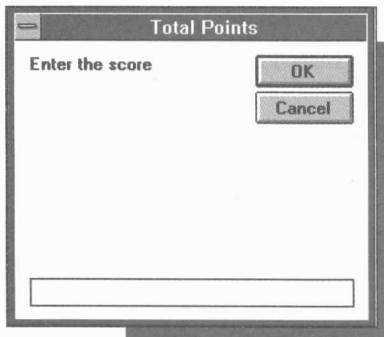


Figure 6-23
InputBox

6. Write **Do-While** loops to replace these **For-Next** loops:

   a)

```
For x = 1 To 25
   ...
Next x
```

   b)

```
For w = 0.5 To 10 Step 0.1
   ...
Next w
```

7. Where do you find the **Exit Do** statement, and what does it do?
8. What happens when an **Exit Do** statement is executed from the inner of two nested **Do-While** loops? Draw a diagram to explain your answer.
9. Compare the relative merits of using a textbox or **InputBox** to enter values into a program.

7

Section

The Line Drawing Program

This project uses graphic methods, mouse events, arrays, and binary file to build a simple drawing program. It uses graphics methods and mous events to build a line drawing program. The coordinates of the endpoint of the lines drawn are stored in arrays. When the drawing is complete the coordinates are stored in a file. Drawings can be loaded from files.

Starting Out

Before you can build a program that draws circles and lines on a form you need to explore two methods:

- **Circle** method
- **Line** method

The code for these methods can be placed in several places, includin in mouse events. Before you start the Line Drawing program, then, yo need to explore the effects of placing the code in different subroutines.

CIRCLE AND LINE METHODS

The **Circle** method is a built-in procedure used to draw circles on a object. The syntax is:

object.**Circle** (x,y),radius

The object, in this case, is the form, a Picture Box control, or the Printer object. The center of the circle is (*x,y*). *X* and *y* are screen coordinates. *Radius* is the radius of the circle. If you omit an object from the syntax, the program draws the circle on the current form.

The **Line** method is a built-in procedure you use to draw lines on an object.

object.**Line** (x,y)-(x',y')
object.**Line** -(x,y)

This method draws a line on an object. As is true of the **Circle** method, the object can be a form, a Picture Box control, or the Printer object. If you omit an object, the program draws the line on the current form. The first version of the **Line** method draws a line between (*x,y*) and (*x',y'*). The second version draws a line from the last point drawn to (*x,y*). The coordinates of this last point are saved in the CurrentX and CurrentY properties of the object. When you use the **Circle** method to draw a circle centered at a point (*x,y*), the last point drawn is set to that point (*x,y*).

MOUSE EVENTS

A MouseDown event occurs when a user presses a mouse button. A MouseUp event occurs when the user releases a mouse button. The parameters of the event procedure tell you which buttons are pressed or released, as well as the coordinates of the point where the press or release occurred. A MouseMove event occurs when the mouse's position changes. The parameters to the MouseMove event tell you the current coordinates of the mouse cursor.

These three mouse events each have four parameters. In this section, only the last two are important: X As Single, Y As Single. These provide the screen coordinates of the mouse pointer.

The mouse events let you program the mouse with finer control than the Click event gives you. Click events occur after a mouse button is both pressed and released. When the Click event occurs, you are not told the coordinates of the mouse cursor. In the program you are going to write, you will respond to mouse events by saving the mouse cursor coordinates when a mouse button is pressed, saving the coordinates when a button is released, and drawing a straight line between those two points. Clearly, the Click event does not provide enough information to accomplish this.

To experiment with the **Line** and **Circle** methods, follow these steps.

1. Open Visual Basic. If Visual Basic is already open, select New Project from the File menu.
2. Change the caption of the default form to **The Line and Circle Methods**.
3. Double-click on the form to open the Code window.
4. In the Proc drop-down menu, select the MouseDown event.
5. In the Form_MouseDown (...) event procedure, insert the following code. Here, *X* and *Y* are parameters of the MouseDown event giving the coordinates of the mouse cursor:

```
Const sglRadius = 50
Circle (X,Y), sglRadius
```

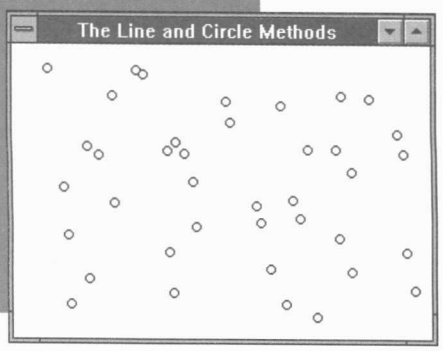


Figure 6-24
Drawing circles on the form

6. Run the program. Move and click the mouse on various spots on the form. Every time you press any mouse button, a small circle appears, centered at the point where you pressed the mouse button (see Figure 6-24). No action is taken in response to releasing mouse buttons or to mouse movement.
7. Stop the program.

Now, you will make the program more general by allowing the user to specify a different radius each time the program is run. The user can enter a new radius, or just accept the default value of 50:

1. Reopen the Code window.
2. Remove **Const sglRadius = 50** from the MouseDown event procedure.
3. Add this line to the MouseDown event:

```
Debug.Print X,Y
```

4. Add this line to the general declarations section of the form:

```
Dim gsglRadius As Single
```

5. Add this code to the Form_Load procedure (select Form from the Object drop-down list and Load from the Proc drop-down list):

```
gsglRadius = Val(InputBox("Enter a value for the radius",,"50"))
```

6 Change the name of the radius variable in the Form_MouseDown event from *sglRadius* to *gsglRadius*, so that the statement that draws the circle looks like this:

```
Circle (X,Y), gsglRadius
```

7 Run the program. An InputBox appears, prompting you to enter a radius. Enter a radius and click OK. Now all the circles that appear will be of that radius.

8 Click on the form in various places. Each time you press a mouse button, a circle appears with the radius you selected.

9 Click on the Debug window. The *X* and *Y* coordinates are displayed of the center of each circle you drew.

10 Stop the program, then run it again. In the InputBox, enter a different radius, and repeat the steps. When you are finished, stop the program.

Now, you will respond to the MouseUp event by using the **Line** method to draw a line between the points at which a mouse button was pressed and released. To do so:

1 Reopen the Code window.

2 Insert this line of code in the MouseUp event procedure:

```
Line -(X,Y)
```

3 Run the program. Enter a radius, as prompted.

4 Press a mouse button. Due to the instructions in the MouseDown event, the **Circle** method draws a circle on the form with the radius you entered, centered at the point where you pressed the button.

5 Now, press a mouse button *and drag* before you release the mouse button. The **Circle** method draws a circle due to the MouseDown event. When you release the mouse button, a MouseUp event occurs. As a result, the instructions in that event procedure are executed and a line appears.

6 Stop the program.

7 Save the project and form files.

If you press and then release a mouse button without moving the mouse, a line is still drawn. However, the beginning and endpoints of the line are identical, so the line is not visible.

Setting Up the Form

Setting up the appearance of this simple drawing program is easy. you can see in Figure 6-25, you need a menu bar with a single ent (File). This menu opens a drop-down list of four commands. There a no buttons or other objects on the form.

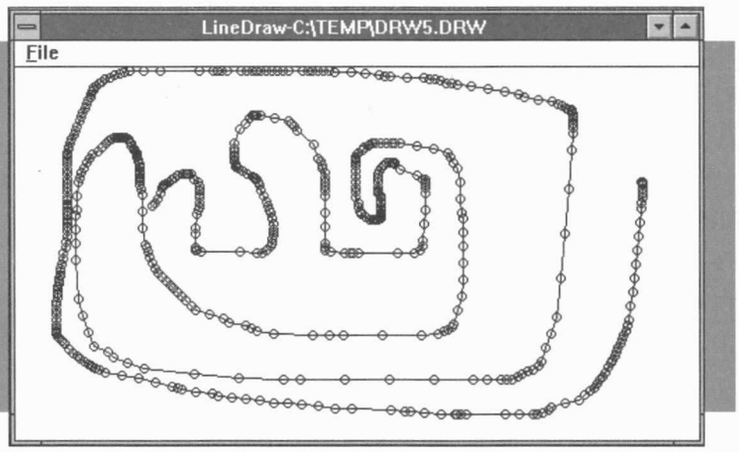


Figure 6-25
A simple drawing program

To set up the form:

1. Create a directory to store the Line Drawing project.
2. Start Visual Basic, or choose New Project from the File menu.
3. Change the caption of the form to **LineDraw**. Change the name o the form to **frmDrawLine**. Change the AutoRedraw property of the form to True. This redraws the contents of the form after a dialog box has appeared and vanished from the form, erasing part of its contents.
4. Create a menu for the form with the structure shown in Figure 6-26 (see Chapter 5, section 2 for a discussion of how to set up a menu):

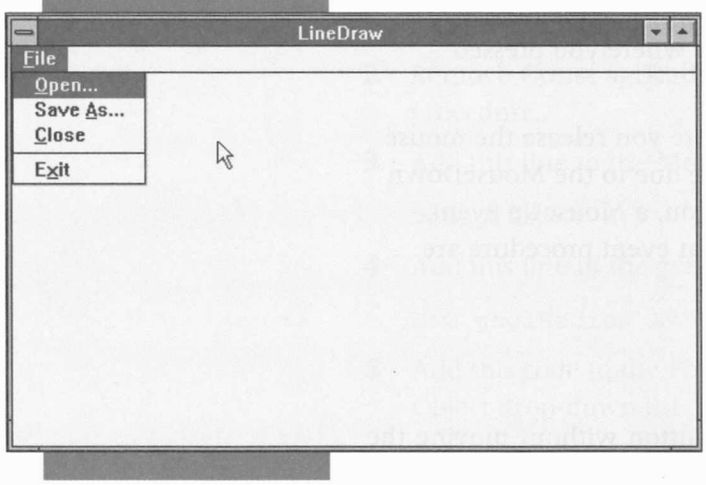


Figure 6-26
File menu

Entering Code: Stage 1

Now, add the following code and test the program:

1. In the general declarations section of the form, put the following statements:

```
Option Explicit
Dim gfDrawing As Integer
```

The *gfDrawing* variable is used as a flag. For more detail on flags, see the textbox.

2 Open the form's MouseDown event Code window and enter the code:

```
gfDrawing = True  '—used by MouseMove handler
Circle (X,Y),50
'—Set form's CurrentX, CurrentY properties
CurrentX = X
CurrentY = Y
```

The CurrentX and CurrentY properties provide coordinates used by the next drawing method. The **Line** method uses these coordinates as the first point of a line. True is a constant in Visual Basic equal to -1.

3 Run the program.

4 Press and release a mouse button on the form several times. The **Circle** method draws a circle every time a MouseDown event occurs. That circle has a radius of 50.

5 Stop the program.

6 Save the project and form files in the directory created from the project.

The changes you just made have produced a less complex program than you had before. You have laid the foundation for the next stage, in which you use the flag *gfDrawing* to determine whether or not to draw anything. In the next section, you will clear the *gfDrawing* flag in the MouseUp event. You will also draw in the MouseMove event, only if the flag is set.

Entering Code: Stage 2

This section adds the code that draws the images. You'll find the code surprisingly simple. The **Line** and **Circle** methods are used. Nothing at all is drawn if the flag, *gfDrawing*, is set to **False**.

1 Enter the following code in the MouseMove event procedure:

```
If gfDrawing Then
   Line -(X,Y)
   Circle (X,Y),50
End If
```

FLAGS

A flag is a variable used as a signal in a program. A flag may be True or False, on or off, or yes or no. Flags are set to show conditions. If the flag is set, the condition is True, on, or yes. Flags are used in If-Then statements. They are usually global variables (of Integer type), which are set and cleared in certain procedures to signal that some condition does or does not hold true. The flag is then tested in other procedures, and actions are taken depending on the state of the flag. A common example is the use of a flag to indicate whether a mouse button is down or not.

In this exercise, the variable's prefix, *gf*, indicates that the variable is visible to any procedure in the form. Visual Basic does not automatically make the variable global because of the "g" in the prefix. You made it global by placing it within the general declarations area of the form. When you use the variable in an event procedure, its prefix makes it easy to remember that you are using a global variable. The "f" portion of the prefix indicates that the variable contains a flag.

2 Enter the following code in the MouseUp event procedure:

```
gfDrawing = False
```

3 Enter the following command in the mnuExit_Click() event procedure:

```
End
```

4 Run the program. Press a button, drag, and release the butto Also try pressing and releasing buttons without dragging.

5 Stop the program, then save the project and form files. The **Circle** method draws a circle whenever either a MouseM or MouseDown event occurs. You don't, however, get a Mous Move event for every point that the mouse moves over. The fa you move the mouse, the more sparse the circles become.

Entering Code: Stage 3

Your next task is to deal with file input/output (i/o). Suppose, for ex ple, that the user of your program has created a drawing and now w to save it. How do you save a drawing?

Remember the list of *x* and *y* coordinates you saw displayed in Debug window? You can identify lines by the *x* and *y* coordinates their endpoints. You can identify circles by the *x* and *y* coordinates their centers. Therefore, you need to set up two arrays, one to recor coordinates and one to record *y* coordinates. Procedures can then sa the lists of coordinates to a file and read lists of coordinates from file After reading a list of coordinates from a file, the drawing that tho coordinates describe can be recreated.

To set up an array, you need to pick a maximum number of entrie In this program, you use two arrays, both of the same size. An adequat number for the maximum number of entries is 2000.

As you work through these steps, you will also experiment with cre- ating procedures that are not linked to particular events. One of these procedures (Sub DrawCircle) replaces calls to the **Circle** method in the MouseDown event. In addition to drawing circles, this procedure saves coordinates in the arrays.

To create the arrays:

1 Add the following lines to the general declarations section.

```
Const gnMaxPoints = 3000     ' gn = global number
'—gsng = global single
Dim gsngX(1 To gnMaxPoints) As Single
Dim gsngY(1 To gnMaxPoints) As Single
```

```
'-Actual number of points recorded
Dim gnNumPoints As Integer
'-gstr = global string
Const gstrTitle = "LineDraw"
```

2 Create a new procedure by entering the following lines in the general section:

```
Sub DrawCircle (X As Single, Y As Single)
'-Draw the circle
   Circle (X, Y), 50
'-Too many points?
   If gnNumPoints < gnMaxPoints Then
'-Add one more point
      gnNumPoints = gnNumPoints + 1
'-Save X and Y coordinates
      gsngX(gnNumPoints) = X
      gsngY(gnNumPoints) = Y
   End If
End Sub
```

3 In the MouseDown and the MouseMove procedures, replace

```
Circle (X,Y),50
```

with

```
DrawCircle X,Y
```

4 In the Form_Load procedure, enter the following line:

```
gnNumPoints = 0
```

5 Select mnuClose from the **Object** drop-down list in the Code window. In the mnuClose_Click procedure, enter the following code.

```
gnNumPoints = 0
frmDrawLine.Cls
frmDrawLine.Caption = gstrTitle
```

6 Select mnuSaveAs from the Object drop-down list. In the mnuSaveAs_Click procedure, enter the following code.

```
'-Collect filename from user
Dim strFn As String
strFn = UCase$(Trim$(InputBox("Filename", "OpenFile")))
'-Open binary file for output
Open strFn For Binary Access Write As #1
'-Save actual number of points
```

continued

PROCEDURES AND FUNCTIONS

Procedures and functions not linked to particular events can be included in a form module or a code module. To start a new procedure, you can select New Procedure from the View menu. Then, you choose Sub or Function and supply a name.

Another option is to open the general declarations section of a form's code. Under the declarations, enter the top line of the new procedure. As soon as you press Enter at the end of that line, Visual Basic creates the procedure, automatically adding the last line (End Sub). (To see any declarations in the general declarations area, you now have to select declarations from the Proc drop-down list.) Visual Basic will now list this procedure in the Proc list.

```
Put #1, , gnNumPoints
'—Local variable for loop
Dim i As Integer
'—Loop to actual number of points
For i = 1 To gnNumPoints
   '—Save coordinates
   Put #1, , gsngX(i)
   Put #1, , gsngY(i)
Next i
Close #1
'—Reset form caption
frmDrawLine.Caption = gstrTitle & "-" & strFn
```

7 The Open command collects the coordinates of the points and loads those coordinates in two arrays. Once the points are loaded, the DrawLines procedure draws the lines and circles. In the general section of the form, enter the following code.

```
Sub DrawLines ()
   Dim i As Integer
   CurrentX = gsngX(1)
   CurrentY = gsngY(1)
   '—Draw the first circle
   Circle (gsngX(1), gsngY(1)), 50
   '—Plot the rest of the lines and circles
   For i = 2 To gnNumPoints
      Line -(gsngX(i), gsngY(i))
      Circle (gsngX(i), gsngY(i)), 50
   Next i
End Sub
```

8 Select mnuOpen from the Object drop-down list. In the mnuOpen_Click procedure, enter the following code.

```
'—Collect filename from user
Dim strFn As String
strFn = UCase$(Trim$(InputBox("Filename", "OpenFile"
'—Open binary file for input
Open strFn For Binary Access Read As #1
'—Save actual number of points
Get #1, , gnNumPoints
'—Local variable for loop
Dim i As Integer
'—Loop to actual number of points
```

```
For i = 1 To gnNumPoints
'-Collect coordinates from file
   Get #1, , gsngX(i)
   Get #1, , gsngY(i)
Next i
Close #1
'-Reset form caption
frmDrawLine.Caption = gstrTitle & "-" & strFn
frmDrawLine.Cls
DrawLines
```

9 Run the program. Click and drag, then release.

10 Choose Save As from the File menu. Supply a complete pathname.

11 Choose Close from the File menu.

12 Choose Open from the File menu. Supply the same complete pathname.

The code for Open is almost identical to the code for Save As. Get is substituted for Put. At the end of the routine, the form is cleared and the circles and points are drawn with the DrawLines procedure.

Finishing the Program

Now, use the program:

1 Experiment with drawing, saving, opening, and closing.

2 Stop the program.

3 Save the project and form files.

4 Choose Make EXE File to create a stand-alone application.

QUESTIONS AND ACTIVITIES

1. Comment out the **Circle** statement from MouseDown, DrawCircle, and DrawLines. Run the program and observe the effect. Restore the **Circle** statements.

2. Change the declaration for *strFn* from local to global. Move the **Dim** statement to general declarations. Initialize *strFn* in the Form_Load procedure to "". Rewrite the **InputBox** functions to use *strFn* as a default value.

3. Although drawing starts and stops in response to mouse events, once they are saved and reloaded, the breaks between the connected lines are lost. Currently there is no record of where drawing

stops and starts. Modify the MouseUp event procedure to record coordinates (-1,-1) when the mouse button is released. Modify DrawLines to stop drawing when the (-1,-1) coordinates are encountered.

Summary

The **Format$** function converts values into strings according to certain fixed patterns or according to patterns you provide. The syntax is:

variable = **Format$**(*value, format string*)

Visual Basic shares code through

- Custom controls
- Code modules

To use a custom control, you must place the control's VBX file in the System folder of Windows. Visual Basic will add the filename to the project file list that is displayed in the Project window. You can place procedures you want available to more than one form in a code module.

The scope of a variable is determined by its visibility and lifetime. Visibility is a question of what procedures can use (or "see") a variable. The lifetime of a variable depends on whether the value of the variable is available apart from the limited lifetime of an event procedure. A variable declared and used in one event procedure is not available in another. Variables declared in the form module are available to all event procedures within the form. Variables declared with the **Global** statement in a code module are available to any form in the project and its event procedures.

A static variable, declared within an event procedure, isn't available to other procedures, but its value does persist beyond the end of the procedure in which it is declared. When the procedure is reentered, the static variable is there, with its old value, available for use.

The **Option Explicit** statement is entered in the general declarations section of either a form module or a code module. By using this statement, you require that each variable be declared before use.

You use arrays so that you can save a list of values. The list has a single name. Each element of the list has a unique whole number subscript. For instance, an array of names can be declared as follows:

```
Dim Names(100) As String
```

Another way to declare the same array:

```
Const MaxNames = 100
Dim gstrNames(1 To MaxNames) As String
```

Using 1 To MaxNames lets the programmer specify the starting subscript and use a constant as a maximum subscipt. Using a constant makes it easy to change the references to that value throughout the program with a single change.

The first characters of the name, *gstr*, shows the data type of the variable. The prefix *g* indicates the variable is global; the *str* shows that the variable contains strings.

The backslash (\) operator divides two numbers and gives the whole number result.

Scrollbars can be used to provide a subscript value to access the elements of an array.

The **Do-Loop** statement, one of a number of indefinite loop statements, allows a loop to execute an indefinite number of times. The number of times the loop is executed is controlled by factors within the loop. A sentinel-controlled loop continues until a certain value, called a sentinel, is entered or generated. The syntax of this statement is:

Do While *condition*
    statements to execute while condition is true
Loop

The **InputBox** function allows the entry of values without textboxes. The syntax is:

variable = **InputBox** *(Prompt, Caption, Default value)*

Prompt is a string containing the instruction to the user. *Caption* is the caption of the window. *Default value* is a string representing a default value for the variable. If a user presses Enter without changing the default value, this value is assigned to the variable.

Problems

1. The How-Long-in-School Problem

Write a program using textboxes for input, prompting the user to enter the number of hours a day spent in school and the number of daylight hours in a day. Calculate and display the percentage of daylight time spent in school. Use the **Format$** function to display the percentage.

2. The Sales Tax Problem

Write a program using InputBoxes to enter the amount of a purchase and the percentage sales tax charged. In Illinois the sales tax is 6.25%. You would enter that percentage as 6.25, omitting the percentage sign. Using the Currency and Percent format strings, display the original amount of the purchase (as currency), the sales tax percentage, the sales tax charged (as currency), and the total price including sales tax (as currency).

3. The Running Total Problem

Put a textbox on a form. This box is used to enter values. The values are added together. Declare a Single type variable, *RunningTotal*, in the general declarations section of the form. In the KeyPress event of the textbox, when KeyAscii is 13 (the user presses Enter), convert the string in the textbox to a number, and add it to *RunningTotal*. Place a single command button on the form. The button, when pressed, puts a label on the form containing the running total.

4. The Random Number Program

Rnd is a built-in function that returns a random single value between 0 and 1. Write a program with an indefinite loop to enter values generated by **Rnd** into a listbox on a form, until a value greater than 0.95 is generated.

5. The Baby-Sitting Problem

A season pass to a theme park is $65. Write a program using an InputBox to enter amounts of money earned baby-sitting. Keep a running total of the amounts entered, and count the number of jobs. When the total amount exceeds $65, display the amount saved and the number of jobs it took.

6. The Screen Coordinate Problem

Write a program that will display the values of *x* and *y*, the parameters sent to the MouseDown event, when the mouse is clicked on a form. Use two labels near the bottom of the form to display the values for *x* and *y*.

7. The Indefinite Average Problem

Write a program to enter a list of test scores with an InputBox, display them in a listbox, count them, keep a running total (as they are entered), and display the current average as each number is entered. Use these variables: *Total*, *Cnt*, *Average*, *Score*, and *Finish*. The first two are to keep your running total and count the number of entries. *Cnt* should be an Integer, but *Total* could be an Integer, a Single, or left as a Variant type. The *Average* and *Score* should be Single, and *Finish* should be an Integer, which is the type used to represent **True** and **False**. Use *Finish* as a condition to end the loop.

The structure of the loop should be:

```
Finish = False
   ...
Do While Not Finish
   ...
Loop
```

Finish should be set to true when a score of 0 is entered from the InputBox.

When the *Score* entered is not 0, it should be added to the listbox, counted, and added to the running total. The average should be calculated and displayed in a textbox.

8. The Goldbach Conjecture

Take any positive number. If it is even, divide it by 2. If it is odd, multiply by 3 and add 1. It is conjectured that for any positive number, the sequence of numbers generated by this rule ends in 1. For instance, if the number is 7, we have the following sequence:

7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

Write a program to use an InputBox to collect a number from the keyboard. Use a multiline textbox or a listbox to display the terms of the sequence. Because the length of the sequence is unknown, use an indefinite loop (a **Do-Loop**) to control the flow of the program.

9. The Shopping Problem

Write a program to help a shopper know when to stop. Prompt the user for an amount to spend. This is your base amount. Enter the amounts of the shopper's purchases and subtract from the base amount until the amount entered, if subtracted, would cause the base amount to be negative. Display the purchase amount that broke the bank and display the current base amount.

10. The Overloaded Circuit Problem

Most household electrical circuits are designed to handle about 15 amps of current. Write a program to allow the user to enter the amperage requirements of appliances, display the values in a listbox, maintain a running total, and display a warning message if the total exceeds 15 amps.

11. The Length of a Loan Problem

Write a program to enter a loan amount, a yearly interest rate, and a monthly payment. Use a **Do-Loop** to calculate the monthly interest, add that amount to the unpaid balance, and subtract the monthly payment until the unpaid balance is 0. The program should count the number of monthly payments required to pay off the loan and display that result. The monthly interest is the amount of the loan balance times the monthly interest rate. The monthly interest rate is the yearly interest rate divided by 12.