Chapter 4

# Decision Making and Looping

## GOALS

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

- Use **If-Then** to ensure that a sequence of statements will be executed only when a given condition applies.
- Use **If-Then-Else** to select which of two sequences of statements will be executed, depending on whether a given condition is true or false.
- Write Basic expressions to form "Boolean conditions," which are expressions whose possible values are the constants True and False.
- Use each of the three forms of **If-Then** appropriately.
- Apply the triangle inequality in an application program.
- Explain events, parameters, and properties in more detail than previously.
- Use compound Boolean expressions, with And and Or.
- Use the **Mod** operator to test divisibility.
- Keep running totals and count sequences of statements.
- Apply concepts of divisibility and running totals to the Prime Number application.
- Apply the concept of running totals to join strings together.
- Build strings for display in multiline textboxes.

## OVERVIEW

*In this chapter, you will see how to use the three forms of the **If-Then** statement, how to test divisibility, how to keep a running total, and how to count the occurrences of a particular statement. You will also be introduced to the fundamental looping statement, the **For-Next** statement.*

*These constructs allow more complex "flow of control" within a program than the simple straight-line flow that you have used in programs so far.*

*Flow of control refers to the order in which statements are executed. In straight-line code, the flow of control is obvious, even trivial: the next statement to be executed is the statement immediately following the current one.*

*The* ***If-Then*** *statement makes it possible to execute other statements conditionally—for example, only when the value of a given variable exceeds 10. The* ***For-Next*** *statement causes the statements it encloses to be executed repeatedly a specified number of times. The computer, in other words, does more work in response to less code (and less typing).*


*Section*

## If-Then Statements

You use **If-Then** statements to make choices in a Visual Basic program. You make the same kind of choices every day. Should I go to a movie or go rollerblading? Should I read a book or write a short story? More decisions follow from whatever choice you make. For example, if you decide to go to a movie, then you need to decide which movie and find a way to get there. If you choose to go rollerblading instead, you need to pick up your rollerblades and kneepads and decide where to go.

Visual Basic uses the same kind of logic to control the flow of a program from one section to another. You express choices in the form of an **If-Then** statement. Then, depending on the choice made, Visual Basic executes different sections of code. A "section of code" may mean a single line. An **If-Then** statement allows you to conditionally insert a line or many lines into the straight-line flow of a program, or run one section instead of another.

For example:

**If** *the unpaid balance is not zero* **Then**
    *Continue to make payments,*
**Else**
    *Stop making payments*
**End If**

**If** *the player's health points are zero* **Then**
    *The game is over, so end the game*
**Else**
    *Continue play*
**End If**

**If** *the pathway is blocked* **Then**
    *Choose an alternate path*
**Else**
    *Proceed along pathway*
**End If**

**If** *the last name is "McRae"* **Then**
    *Display full name*
    *Display address*
    *Display phone number*
**Else**
    *Continue name search*
**End If**

**If** *the time is 7am* **Then**
    *Ring the alarm*
**End If**

You can see that all of these statements are set up in the same way. The statements have the same general structure:

- The first line starts with **If**.
- The first line ends with **Then**.
- One or more statements follow the **If-Then** line. They will be executed provided that the expression after the word **If** is true.
- The word **Else**, if it is included, begins a new line.
- One or more statements follow the **Else**. They will be executed provided that the expression after the word **If** is false.
- The end of the statements following the **If-Then** or the **Else** is signaled by the words **End If**.

## KEYWORDS

The words **If**, **Else**, and **End If** are keywords, also known as reserved words. Basic sets these words aside and gives them special meaning that you cannot override. You cannot name a variable *If*, for example. Visual Basic recognizes its keywords when you type them, and it saves you effort by capitalizing them correctly. Thus, you can simply type *if*; Visual Basic will recognize its keyword, then substitute **If** as soon as you move the cursor onto another line. In fact, Visual Basic even recognizes *endif* as a shorthand for the keyword combination **End If**, and will expand it properly.

This structure is called the syntax of the **If-Then** statement. This section introduces three variations in the syntax of this type of statement. Each variation is appropriate for a specific range of situations that arise in programming.

## Writing an If-Then Statement

The simplest version of the **If-Then** statement uses this syntax (see Figure 4-1):

**If** *condition* **Then**
    *one or more statements, to be executed only if condition is True*
**End If**

The condition is typically an expression that gives a **True** or **False** result when the program evaluates it. Comparisons are the most common examples of such expressions. **True** and **False** are Basic keywords. **False** has the value of 0, while **True** has a nonzero value (specifically, -1). Consider the following three examples of conditions:

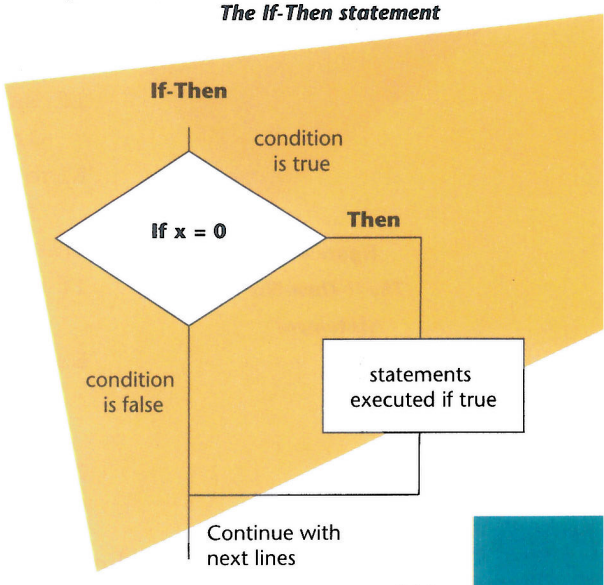
a) 2 * *x* = *y*
b) HealthPoints > 0
c) Payments < Years * 12

In a), the value of **2 * x** either is or is not equal to *y*. If it is, the condition is true, and the comparison expression has the value **True**; otherwise, the condition is false, and the expression has the value **False**. In b), the condition is true, and has the value **True**, if the value of HealthPoints is greater than 0. The expression is false, and has the value **False**, if the value of HealthPoints is less than or equal to zero. In c), the expression is true if the value of Payments is less than the number of Years times 12; otherwise, it is false.

Assignment statements, as you may remember, are limited to a single variable on the left-hand side (see Chapter 2). You are not limited in this way with the conditions in an **If-Then** statement. Visual Basic evaluates the left and right sides of the condition, then compares them. The comparison results in a **True** or **False** value.

```
If  x - y = 0 Then
   lblResult.Caption = "x and y are equal"
End If
If x2 - x1 <> 0 Then
   lblSlope.Caption = "The slope exists."
   m = (y2 - y1) / ( x2 - x1)
End If
```

***Figure 4-1***
***The If-Then statement***



This form of the **If-Then** statement executes instructions *only if* the condition is true (has a nonzero value). If the condition is false, the program jumps to the first statement following the **End If** keywords, which marks the end of the entire **If-Then** statement. The **End If** signals the end of the **If-Then** section of code.

After the **If-Then** line, you need to provide instructions for the program to follow if the condition is true. What is the code that the program should execute? As you can see in the last example, you can include more than one line of instructions to be executed.

## Writing an If-Then-Else Statement

Here is another useful variation on the **If-Then** statement (see Figure 4-2):

**If** *condition* **Then**
    *statements to be executed if condition is true*
**Else**
    *statements to be executed if condition is false*
**End If**

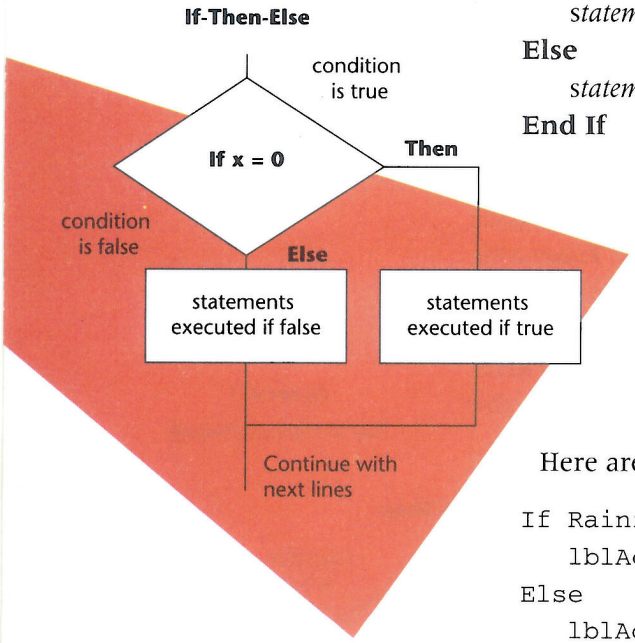



***Figure 4-2***
***The If-Then-Else statement***

In this variation, the program executes the first list of statements if the condition is true. So far, this is exactly like a simple **If-Then** statement. Now, though, you add a second list of statements after the **Else**. The program executes *this* list if the condition is false. If the first list executes, the second list does not. If the second list executes, the first does not.

Here are some examples:

```
If Raining = True Then
   lblActivity.Caption = "Go to the movie."
Else
   lblActivity.Caption = "Go to the beach."
End If
If x2 - x1 = 0 Then
   lblSlope.Caption = "No slope"
Else
   lblSlope.Caption = "Slope exists"
   Slope = (y2 - y1) / (x2 - x1)
End If
```

Indentation makes code easier to read by outlining its structure. You should indent the statements that the program executes in the True and

False branches. Visual Basic will not alter "whitespace" at the beginning of the line. Whitespace includes the spaces that indent the line, which you produce by pressing the Space bar or the Tab key.

However, Visual Basic also ignores whitespace and does not use it to determine how statements should be grouped. That is why the **End If** keywords are required to mark the end of the statements comprising the **If** or the **Else** sections of code.

### JUST SO YOU KNOW

In the original version of Basic, the **If-Then** statement was restricted to a single line. The syntax of this statement was:

**If** *condition* **Then** *one or more statements separated by colons (:)*

Visual Basic lets you use this form of the **If-Then** statement, but you will seldom have a good reason to do so. Using separate lines makes the code much easier to read and debug.

## Nesting Statements

Some decisions require complicated reasoning to choose the proper path through the program's statements. And, as you've seen, some decisions just lead to more decisions. For example:

```
If you go to a movie Then
    If you choose a romance Then
        If you forget your tissues Then
            wipe your tears on your sleeve
        Else
          blot your tears with tissues
        End If
    End If
End If
```

When one **If-Then** statement is contained in another, the statements are *nested*.

Another kind of nesting occurs when an **If** statement is contained in the **Else** branch of an **If-Then** statement. For example:

```
If you choose to rollerblade Then
    Put on rollerblades and pads
Else
    If you choose to bowl Then
        Take bowling ball and shoes to bowling alley
    Else
```

*(continued)*

```
        If you choose to ride a bike Then
            Get bike out of garage
        End If
    End If
End If
```

Nesting **If** statements can make the logical structure of the program difficult to understand, due to repeated indentation and a forest of **If**'s, **Then**'s, and **Else**'s. To reduce the confusion, Visual Basic uses the **ElseIf** statement.

### USING ELSEIF

If you nest **If-Then** statements, the indentation levels of the code can become very confusing. Visual Basic lets you use **ElseIf** as a way to prevent this problem.

For example, you could write:

```
If x = 0 Then
    do something
Else
    If x = 1 Then
        do something different
    Else
        do the default action
    End If
End If
```

VB lets you write a shorter form instead:

```
If x = 0 Then
    do something
ElseIf  x = 1 Then
    do something different
Else
    do the default action
End If
```

## Souped-Up Digital Clock

To experiment with the **If-Then-Else** statement, revisit the Digital Clock program and make it more eye-catching (perhaps even annoying). You will modify the Timer event handler so that the displayed time grows and shrinks continually, repeating a cycle every 10 seconds.

The code for the Timer event follows, and it provides a good example of why **ElseIf** is convenient (imagine what the code would look like if it used nested **If-Then-Else**'s) :

## INCLUDING COMMENTS

Programming statements can be self-explanatory, but more often they require some explanation. Programmers build explanations into their code by including comments. Comments are brief descriptive sentences or phrases that explain what's going on in the program. Comments can be included anywhere in a line. Start a comment with a single quote. When the default colors are active, Visual Basic will change the text of a comment to green.

Add this code for the Timer event, then run the program:

```
'_A modification to Digital Clock:
Sub Timer1_Timer ()
Dim nHeight As Integer
Dim n As Integer
'_Now is a function that returns the system date and
time in a single value.
'_Second gives a value between 0-59 equal to number
of seconds of current time.
n = Second(Now) Mod 10
If n = 0 Then
   nHeight = 8.25
ElseIf n = 1 Or n = 9 Then
   nHeight = 9.75
ElseIf n = 2 Or n = 8 Then
   nHeight = 12
ElseIf n = 3 Or n = 7 Then
   nHeight = 13.5
ElseIf n = 4 Or n = 6 Then
   nHeight = 18
Else      'n = 5
   nHeight = 24
End If
lblTime.FontSize = nHeight
lblTime.Caption = Time$           ' as before
End Sub
```

All but the last line are new. The label control (as well as the form) must be resized to accommodate 24-point text (the largest size that the displayed time achieves), and its Alignment should be set to Centered so that the horizontal expansion and contraction are more pleasant.

Every 10 seconds "on the 10 seconds," the time is displayed in 8.25 type. On each of the next 5 seconds, the size increases through 9.75, 12, 13.5, 18, and ultimately 24. On each of the next 4 seconds, it decreases, running down through the same sequence of point sizes before starting the cycle again.

## ACTIVITIES

1. Write **If-Then** statements for the following.
   a) If the value of the variable *x* is 2, then set *y* to 0.
   b) If the value of *LastName* is "McRae", then set the Caption property of label1 to the string "Found".
   c) If the value of *Month* is greater than 48, then set the Caption property of label2 to "The payments are over".

2. The Freefall program will give a run-time error if the user enters a non-numeric value into the time textbox and then clicks on the Calculate button. You can test for this, and take appropriate measures. Change the Click handler of the Calculate button to contain the following code:

```
If IsNumeric(txtTime) Then
   same statements as before
Else
   lblDistance = "?"
End If
```

3. Convert the following to the **If-Then-Else** syntax (the **mod** operator divides two whole numbers and returns the remainder):
   a) If $x = y$, then set $z$ to 34; otherwise set $z$ to 12 and $x$ to $x + 1$.
   b) If *Month* mod 12 = 0, then set *NewYear* to *NewYear* + 1 and *Month* to *Month* + 1.
   c) If *Prime* mod 2 = 0, then set lblPrime to " not "; otherwise, set *Divisor* to 3.
   d) If *LastName* is "Hanratty", then set *LastPlayer* to True.

4. Convert the following English sentences to Basic statements. Make up variable names where appropriate:
   a) If the year is 1994, then the balance is twelve thousand.
   b) If the computer is a "386", then the label lblWindowsCanRun is "OK"; otherwise, Windows is "no go".

c) If the bike is a "mountain" bike, then it will go off-road. If it isn't, it won't.

5. Make up three **If-Then** problems of your own. Follow the patterns used in Exercise 2 above.

6. Write nested **If-Then** statements for the following. Make up variable names where appropriate:
   a) If a player's name is "Decker", then the team playing is the "Marlins". Otherwise, if a player's name is "Sosa", then the team playing is the "Cubs".
   b) If a person's age is greater than 12, then if a person's age is less than 18, then *StudentAge* is true.


# The Cereal Program

The Cereal project gives you another opportunity to experiment with **If-Then** statements. You will use an **If-Then** statement to choose between two messages the program could display. In the process, you will work with different methods of changing focus within a program. Whatever object has the focus is the active object, the one where keyboard input goes. One of the methods you will use to change focus requires an **If-Then** statement.

## Starting Out

No one likes to run out of their favorite cereal. To prevent this from happening, you need to know how much cereal you have in your kitchen and the rate at which you are eating the cereal. With that information, you can predict when you need to add cereal to the grocery list.

You can set up a program to perform the necessary calculations. The program should prompt the user to enter:

- The number of boxes of cereal on hand
- The number of bowls of cereal eaten in a week

On the basis of this input, the program will calculate the cereal available. Depending on that calculation, the program then displays one of two messages:

- "Buy more cereal"

  OR

- "Cereal supply is OK".

You need a single form for this program. Think through the program, then see if your list of objects matches the list below. If not, why not?

- Two textboxes, one to enter the number of boxes of cereal on hand and another to enter the number of bowls eaten each week
- Two labels prompting the user to enter values
- Two labels displaying alternative messages about buying cereal
- Three command buttons to do the following:
  - Perform the calculation and display the result
  - Clear the textboxes
  - Quit the program

The form you are going to create is shown in Figure 4-3.

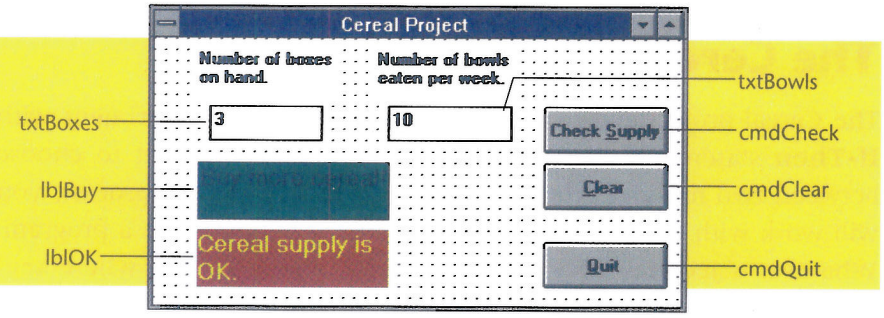



*Figure 4-3*
*The Cereal program*

## NOTE:

**By this point, you have had considerable experience placing textboxes, labels, and command buttons. The directions provided here are abbreviated. If you need more help, refer to Chapter 2.**

Now that you are ready to begin:

1. Start Visual Basic. If it is already running, select New Project from the File menu.
2. Change the caption of the form to "Cereal Project".

## Placing Labels and Textboxes

Your first step is to place the labels and textboxes, then change their properties.

1. Place two labels for user prompts on the form. Double-click on the Label icon in the toolbox. Position the labels near the top of the form.
2. Place two textboxes. You will use one for the user to enter the number of boxes of cereal and the other for the number of bowls of cereal eaten. Double-click on the textbox icon in the toolbar. Position the textboxes below the labels (Figure 4-4).

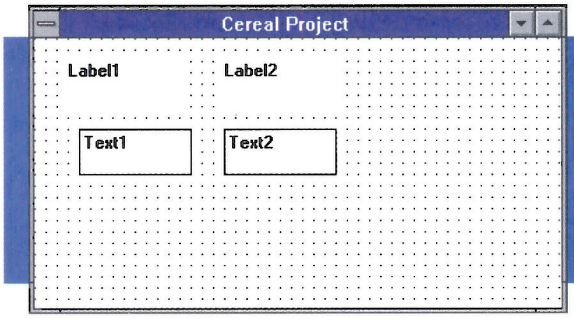



*Figure 4-4*
*Two labels and two textboxes placed*

3 Select the first label. In the Properties window for this object, change the Caption property to **Number of boxes on hand**.

4 Select the second label. In the Properties window for this object, change the Caption property to **Number of bowls eaten per week**.

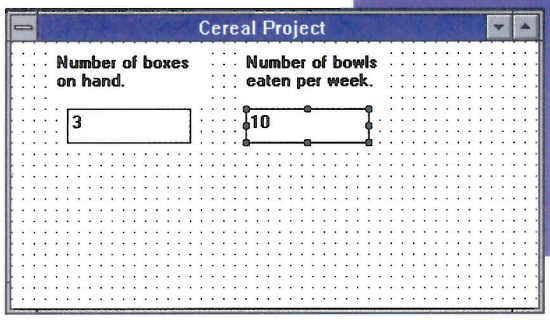
5 Click and drag to resize the label, then check the new size in the Properties window. Position the labels to leave room along the right side of the form for the command buttons.

6 Select the first textbox. In the Properties window for this object, change the following properties:
- Text property: Insert a default value. Replace Text1 with **3**, representing three boxes of cereal.
- Name property: **txtBoxes**
- TabIndex: 0

7 Select the second textbox. In the Properties window for this object, change the following properties:
- Text property: Insert a default value. Replace Text2 with **10**, representing 10 servings.
- Name property: **txtBowls**
- TabIndex: 1

The form should look something like that shown in Figure 4-5.

*Figure 4-5*
*Textboxes with default values*




8 Place two more labels. Select each of the labels in turn and change its properties as listed below.
*First label:*
- Caption property: **Buy more cereal**
- BorderStyle property: 1
- Name property: **lblBuy**
- Visible property: False
- BackColor property: Green

*Second label:*

- Caption property: **Cereal supply is OK**
- BorderStyle property: 1
- Name property: **lblOK**
- Visible property: False
- BackColor property: Red

Although you have set the Visible property to **False** for the last two labels, they will remain visible as you design the form. When the program is running, however, these labels will be invisible until the program sets the Visible property of one of them to **True**. By setting the TabIndex of the two textboxes to 0 and 1, you have established which control will have the focus when the program is started, and which control will gain the focus when the user presses the Tab key.

Setting the TabIndex of the first textbox to 0 ensures that the user can immediately enter the number of boxes on hand as soon as the program is running, without having to move the focus to that textbox by clicking on it or pressing the Tab key one or more times. Once the user has entered the number of boxes on hand, pressing the Tab key moves the focus to the control with the next highest TabIndex—namely, to the second box. Because we read from left to right, we tend to regard that direction as the more natural one for representing operations to be performed in sequence. Although we might not recognize why, the program would seem strange if the TabIndex of the two textboxes were reversed.

The user interface of the Cereal project should now look like the form shown in Figure 4-6.

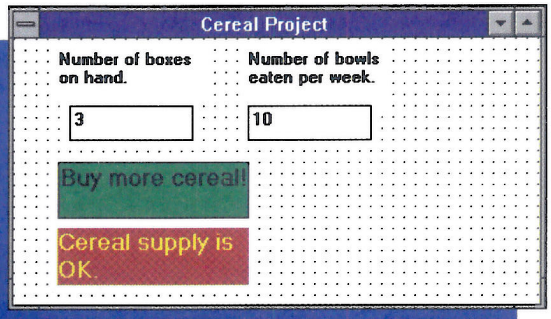



*Figure 4-6*
*Cereal project User Interface with additional labels*

## Adding Command Buttons

You are now ready to set up the three command buttons. As you do so, you should follow the Windows convention for command button captions. Most Windows programs allow users to select a button by pressing

a key combination or by clicking on the button. You let users know which key to press by underlining one letter in the button caption.

You can create an underline in a caption by including an ampersand (&) in the caption text. Visual Basic will then underline the character following the ampersand. When the program is run, the user can click on a button to execute an action. The user can also press the Alt key and the underlined character to run the code attached to the button.

To add the buttons:

1 Place the three command buttons. Double-click on the Command button icon in the Toolbox. Then click and drag on the objects to move them into the right positions.

2 Select a command button, then change its properties as shown. Repeat this step for all three buttons.

*First command button:*

- Caption property: **Check &Supply**
- Name property: **cmdCheck**
- TabIndex property: 2

*Second command button:*

- Caption property: **&Clear**
- Name property: **cmdClear**
- TabIndex property: 3

*Third command button:*

- Caption property: **&Quit**
- Name property: **cmdQuit**
- TabIndex property: 4

## Writing the Code

If you look at a typical box of cereal, you will see the number of servings listed as 12 per box. The real number is probably lower. You need to write code that will compare the available supply with the user's weekly consumption.

The program must make an evaluation as well. At what point, for example, do you want the program to tell the user to buy more cereal? When the available supply is gone? When there are ten servings left? Twenty?

In the code shown here, the cereal supply is considered to be OK if the supply is at least twice a single week's consumption. In other words, if you run this program every day, it will alert you to replenish the cereal supply beginning two weeks before you run out.

As an additional convenience to the user, you should also handle events associated with keyboard input in the textboxes and in two of the command buttons. When a key representing an alphanumeric character is pressed, the **KeyPress** event is fired, and any handler for that event is executed. If the key pressed is the Enter key, the ASCII code 13 is generated. The Enter key is usually pressed at the end of an entry signifying that the entry is complete.

The **SetFocus** method sets the focus to a specified control. You will use the **SetFocus** method within **KeyPress** event handlers for the textboxes so that pressing the Enter key moves the focus just as the Tab key does. For example, pressing Enter will shift the focus of the program from the txtBoxes control to the txtBowls control. Responding to the Enter key when a button has the focus requires a different approach. A button's Click event is fired when it has the focus and the user presses Enter. Because you will be writing code for each button's Click event anyway, you will only have to add another statement to shift the focus as desired.

To write the code for the Cereal project:

1. Double-click on the Check Supply button to open the Code window. In the Click event for this button, insert the code shown in Figure 4-7.

*Figure 4-7*
*Code for the Check Supply button*

CEREAL.FRM

Object: cmdCheck  Proc: Click

```
Sub cmdCheck_Click ()
  Dim Boxes As Integer, Bowls As Integer
  Dim Servings As Integer
  Boxes = Val(txtBoxes)
  Bowls = Val(txtBowls)
  Servings = Boxes * 12
  If Servings >= Bowls * 2 Then
     lblOK.Visible = True
  Else
     lblBuy.Visible = True
  End If
  cmdClear.SetFocus
End Sub
```

2. With the Code window open, select cmdClear from the Object drop-down list. Now insert the code shown in Figure 4-8 for the Clear button.

*Figure 4-8*
*Code for the Clear button*

CEREAL.FRM

Object: cmdClear  Proc: Click

```
Sub cmdClear_Click ()
  txtBoxes = ""
  txtBowls = ""
  lblBuy.Visible = False
  lblOK.Visible = False
  txtBoxes.SetFocus
End Sub
```

3 With the Code window open, select cmdQuit from the Object drop-down list. Now insert the following line of code for the Quit button:

```
End
```

4 With the Code window open, select txtBoxes from the Object drop-down list. Click on the downward-pointing arrow to open the Proc pop-down list. Choose the KeyPress event handler to replace the handler displayed by default (the Change event handler).

5 In the KeyPress subroutine for this textbox, insert the code shown in Figure 4-9.


*Figure 4-9*
*Code for the first textbox*

6 Select txtBowls from the Object drop-down list and enter the KeyPress procedure. Enter code similar to figure 4-9, shifting the focus to cmdCheck.

## Finishing the Program

Visual Basic programs can be written to solve almost any kind of problem. In the Cereal project, you've used three basic controls and some simple code to solve an important breakfast dilemma. Now you are ready to see how the program runs:

1 Run the program several times; use the Tab key and the Enter key to move from object to object. Be sure to run the program with data that will cause each of the labels to be displayed.

2 Save the program. Save both the form file and the project file. You may give each file the same name; Visual Basic will fill in the **.frm** extension for the form, and **.mak** for the project.


# The Triangle Inequality Project

Using **If-Then** statements is a powerful way to control the flow of a program. This program is designed to further your understanding of and expertise with this type of control statement in Visual Basic.

You will use the simplest form of the **If-Then** statement:

**If** *condition* **Then**
    *statements to execute if condition is true*
**End If**

You will also use the following variant:

**If** *condition* **Then**
    *statements to execute if condition is true*
**Else**
    *statements to execute if condition is false*
**End If**

You insert the first of these **If-Then** statements in the KeyPress event handler. In that subroutine, the statement works to change the focus in the program. You will use the ASCII code generated by pressing the Enter key to control the movement of the cursor by shifting the focus from textbox to textbox and command button.

The second form of the **If-Then** statement is inserted in the cmdCalculate Click event procedure. There, you use it to evaluate whether the three values the user enters could represent the sides of a triangle. You will use the And connective to combine three expressions into a single logical expression.

## Starting Out

The triangle inequality is a well-known relationship in mathematics. It states that the sum of the lengths of any two sides of a triangle is greater than the length of the third side. The common sense rendition of this fact says that the shortest distance between two points is a straight line. This relationship is illustrated in Figure 4-10.

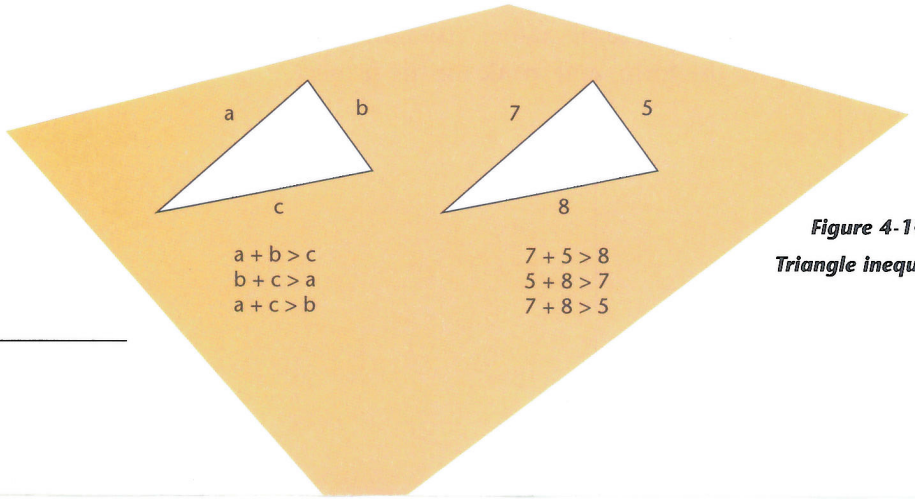



***Figure 4-10***
***Triangle inequality***

If the three sides fail to satisfy the triangle inequality, the sides can't be the sides of a triangle. For instance, if the sides measure 3, 2, and 7, the triangle inequality is violated. As you can see, 3 + 2 is not greater than 7 (Figure 4-11).

You are going to create a program to determine if three line segments can be used to form a triangle. The user will enter the lengths of the three segments, then click the Calc button. At that point, the program will test these values to see if they satisfy the triangle inequality. If they do, the program displays the label "It is a triangle!". If they do not, it displays the label "It is not a triangle!". As in the Cereal program, both of these labels are invisible until the program sets the Visible property of one of them to **True**.

On this form, you will need:

- Three textboxes
- Five labels
- Three command buttons

See Figure 4-12 for a look at the final form.

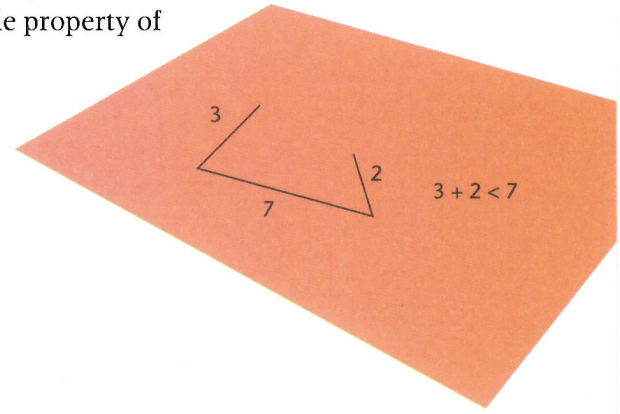



***Figure 4-11***
***No triangle!***

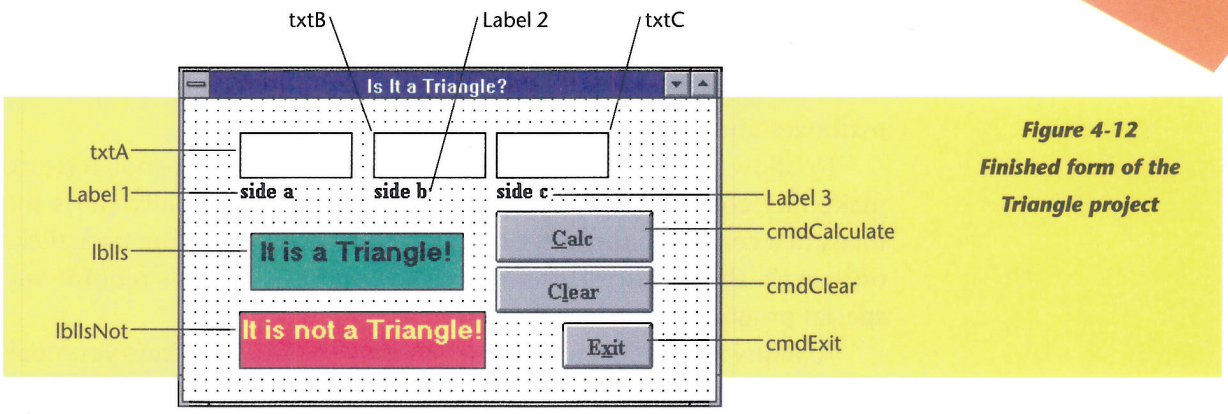



***Figure 4-12***
***Finished form of the Triangle project***

As always, start out by opening Visual Basic and changing the name of the default form. In this case, change the caption of the form to **Is It a Triangle?**

## Placing Objects on the Form

Follow these steps to place objects on the Triangle form:

1. Place three textboxes across the top of the form. These boxes will receive input from the user.
2. Name the boxes **txtA**, **txtB**, and **txtC**. Delete the text from the boxes.

3. Place labels below the boxes and change the captions of the labels to **side a**, **side b**, and **side c**.
4. Place three command buttons on the form. Change their names to **cmdCalculate**, **cmdClear**, and **cmdExit**. Change their captions to **&Calc**, **C&lear**, and **E&xit**.
5. Place two display labels on the form. Name them **lblIs** and **lblIsNot**. Change the caption of the first to **It is a Triangle!** Change the caption of the second to **It is not a Triangle!** Change the BackColor, ForeColor, and Font properties of the labels to make each distinctive. Change the Visible property of both labels to False.

## Controlling Focus

Once a value is entered in the first textbox, a user should be able to move to the next box in any of three ways. These include:

- Pressing the Tab key
- Clicking on the next box
- Pressing the Enter key

To allow users to press the Tab key, you need to change a property in the Properties window. You will change this property for all three textboxes and one command button.

To allow users to click on the next textbox, you do not have to make any changes in the program. Windows automatically shifts the focus to a control that can receive keyboard input when the user clicks on it with the mouse. This way of shifting focus doesn't require any special programming.

To allow users to change focus by pressing the Enter key, you need to make changes to a subroutine in the Code window. You will make changes to accomplish this for all three textboxes and for two command buttons.

### WORKING WITH THE TABINDEX PROPERTY

To control the order of the objects that the focus cycles through when the user presses the Tab key, you need to set the tab order for each control. As you saw with the Cereal program, this order is established by setting the TabIndex property of the controls.

To establish the tab order:

1. Select the first textbox (txtA). Set the TabIndex property for that control to 0.

2 Select the second textbox (txtB). Set the TabIndex property for that control to 1.

3 Select the third textbox (txtC). Set the TabIndex property for that control to 2.

4 Select the Calc button. Set the TabIndex property for that control to 3.

When the form opens during run-time, the focus will be on the first textbox. The control with TabIndex 0 automatically has the focus when a program is run.

## USING THE KEYPRESS EVENT

You can insert code into event handler subroutines so that users can move the focus by pressing the Enter key. Follow these steps for each of the textboxes:

1 Double-click on the txtA textbox to open the Code window.

2 Click on the downward-pointing arrow to open the Proc drop-down list (Figure 4-13).

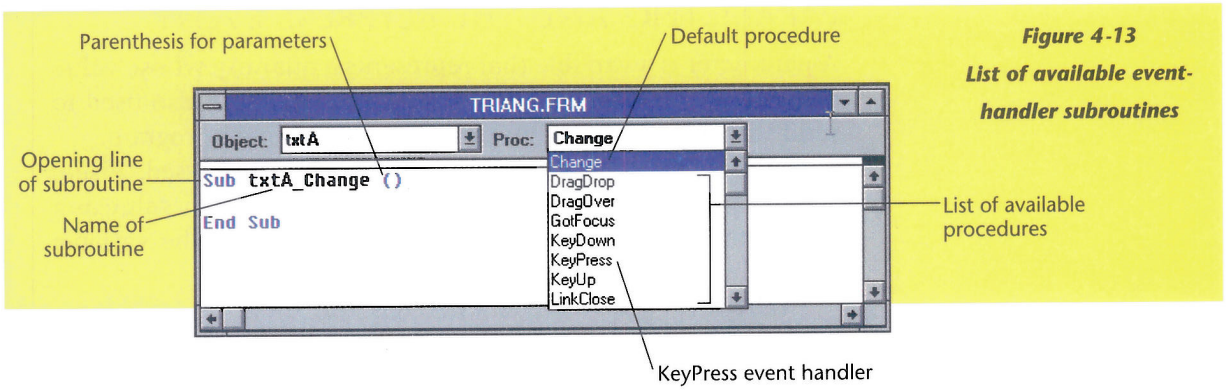



***Figure 4-13***
***List of available event-handler subroutines***

3 Change the displayed event handler from Change to KeyPress. To do this, select KeyPress from the Proc drop-down list. As you can see in Figure 4-14, the name of the event now appears as part of the name of the displayed subroutine.

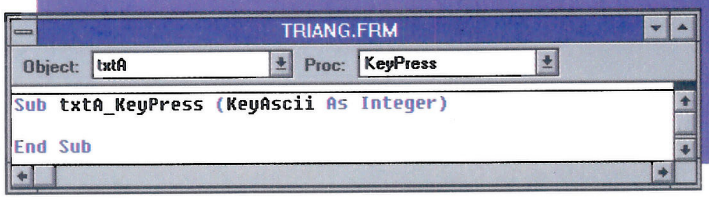



***Figure 4-14***
***Making KeyPress the procedure for the txtA textbox***

4 Insert this **If-Then** statement between the Sub line and the End Sub line:

```
If KeyAscii = 13 Then
   txtB.SetFocus
End If
```

Once you have done so, the name of the event appears in boldface type in the Proc drop-down list. As a result, you can see at a glance which event procedures have code in them. This routine shifts the focus to txtB after the user enters a value in txtA and presses Enter. See the textbox for more detail on this step.

5 Add similar statements into two more subroutines: txtB_KeyPress and txtC_KeyPress. Repeat the same steps outlined above for those two textboxes. The txtB_KeyPress event should shift the focus to txtC. The txtC_KeyPress event should shift the focus to the command button, cmdCalculate.

## PARAMETERS AND THE KEYPRESS EVENT

A parameter is a variable that represents a quantity whose value comes from outside the procedure. Parameters are also used to send values from the event procedure to the main program.

In this case, when an alphanumeric key is pressed, Visual Basic reads the ASCII code of the key, and passes that value as a parameter to the KeyPress event handler through the variable KeyAscii (see Figure 4-15).

Whenever the textbox txtA has the focus and the user presses a key, this procedure is executed. If you did not insert code into the KeyPress procedure, the program would do

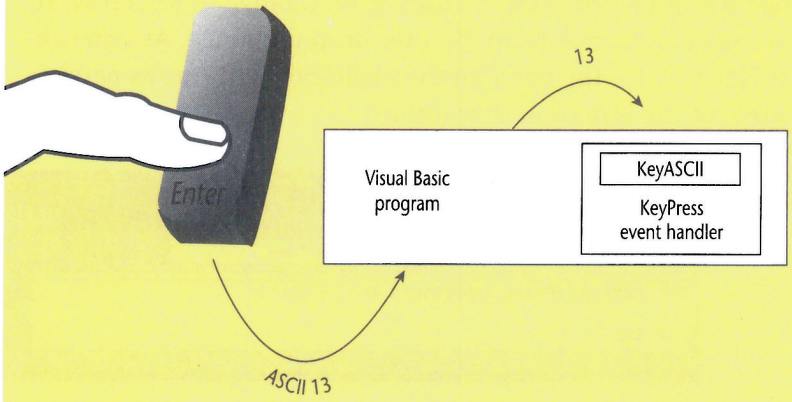



***Figure 4-15***
***Passing a value as a parameter***

nothing in response to the KeyPress event generated in the textbox. However, you are programming the event procedure to react to one key, and one key only. This is the Enter key.

When the Enter key is pressed, the following two ASCII codes are generated: 13 and 10. These codes stand for Carriage Return and Line Feed, respectively. These two actions normally reset the cursor to the beginning of the next line.

The ASCII code 13 is passed to the subroutine through the parameter. At this point, the program will evaluate the condition statement you inserted into the code. Its value is **True**, so the program switches the focus to the second textbox (txtB). If the user pressed any other key, the ASCII code passed to the subroutine would not be 13. Therefore, the program would evaluate the condition statement as **False**, and the focus would not be moved.

## Inserting Code

You need to insert code into the Click subroutine for each of the three command buttons. As you do so, you will work with another **If-Then** statement. You will also experiment with using Assignment statements to simplify Condition statements.

Sometimes condition statements can be very long. Long conditions can make it difficult to understand what is being tested. Parts of the condition statement can be assigned to descriptive variables, and those variables are then used in the **If-Then** statements. The shorter condition statements are easier to read and understand.

### CALC BUTTON

To write the code for this button:

1. Double-click on the Calc button to open the Code window. If the Code window is already open, select cmdCalculate from the Object drop-down list.
2. Enter this code in the Click subroutine for the button:

```
Dim a As Single, b As Single, c As Single
a = Val(txtA)
b = Val(txtB)
c = Val(txtC)
```

```
If a + b > c And b + c > a And a + c > b Then
   lblIs.Visible = True
Else
   lblIsNot.Visible = True
End If
cmdClear.SetFocus
```

There are no parameters sent to or from this procedure, so the parentheses after the procedure name are empty.

The **Dim** statement reserves memory for the variables *a*, *b*, and *c* and declares them to be of the Single type.

Visual Basic defaults are used in the statements that take care of the conversions. Referring to the textboxes only by name automatically accesses the Text property of each box.

```
a = Val( txtA )
b = Val( txtB )
c = Val( txtC )
```

Whenever you refer to the name of a textbox without a particular property specified, Visual Basic assumes you mean the Text property. The **Val** function ensures a proper conversion from the string type of the textbox's Text property to the appropriate numerical type. To remove any dependence upon defaults, or just to make explicit to a reader of your code that a conversion is being performed, you may use the **Val** function, as shown in the following statements:

```
a = Val( txtA.text )
b = Val( txtB.text )
c = Val( txtC.text )
```

Assigning these values to the three intermediate variables *a*, *b*, and *c* relieves you from forming the condition out of the much lengthier expressions on the right-hand side of the assignments. The resulting code is much more readable. These assignments have another, perhaps less obvious, benefit: the resulting code also runs faster. They perform the conversions of strings to numbers once and once only, for each side of the triangle.

If the condition were written out using the **Val** expressions, then a conversion would be performed as many as three times for each side of the triangle, for a total of nine conversions. With such a simple program, you will probably not notice any difference in speed between the two approaches. Nevertheless, the principle illustrated here—avoiding needless recomputation—is a good one to observe.

The next statement is an **If-Then-Else** with a complicated expression for the *condition* part of the statement. This expression tests all three parts of the triangle inequality at once. It can do so because the three parts are joined together by And connectives. Each of these subexpressions—$a + b > c$, $a + c > b$, and $b + c > a$—can be true or false (and have a **True** or **False** value).

For the expression as a whole to be true, all three subexpressions must be true:

True **And** True **And** True = True

If any one of the subexpressions is false, the entire expression is false.

As you can see in the code, the label "It is a Triangle!" appears if the condition has the value **True**. In other words, the line segments *a*, *b*, and *c* do represent three sides of a triangle. The label "It is not a Triangle!" appears if the condition has the value **False**.

In an **If-Then-Else** statement, only one branch is executed. If the statements after the **Then** branch are executed, the statements after the **Else** branch are skipped. If the statements after the **Then** branch are skipped, the statements after the **Else** branch will be executed (see Figure 4-16).

The final instruction in the code for the Calculate button shifts the focus to the Clear button.

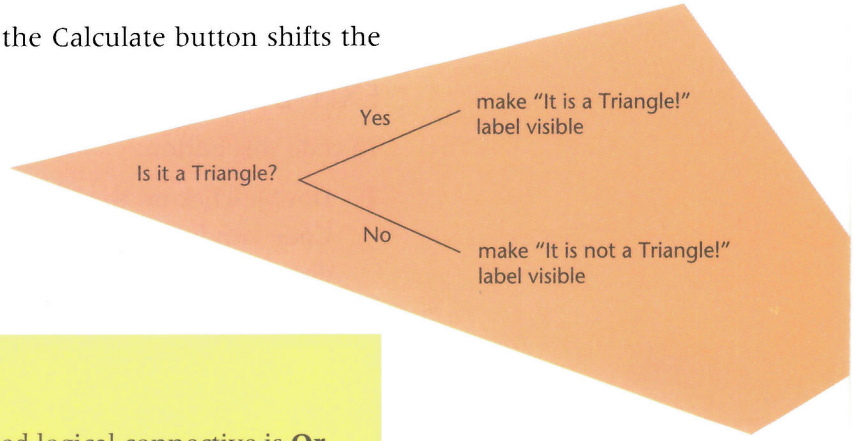



***Figure 4-16***
***Executing a branch of the If-Then-Else statement***

## OR CONNECTIVES

Besides And, the other commonly used logical connective is **Or**. You have already encountered examples of its use, in the revisited Digital Clock example earlier in this chapter. An **Or** expression has the value **True** if one or the other, or both parts, are true. The only condition under which an **Or** expression is **False** is when both the left- and right-hand expressions are false.

True **Or** True = True
True **Or** False = True
False **Or** True = True
False **Or** False = False

## CLEAR BUTTON

To write the code for this button:

1. Double-click on the Clear button to open the Code window. If the Code window is already open, select cmdClear from the Object drop-down list.
2. Enter this code in the Click subroutine for the button:

```
txtA = ""        ' clear the textboxes
txtB = ""
txtC = ""
lblIs.Visible = False   ' make both labels invisible
lblIsNot.Visible = False
txtA.SetFocus     ' shift focus to txtA
```

In this code, the textboxes are cleared by inserting an empty string, represented by the adjacent quotation marks, into the Text property. The empty string is displayed as you would expect: no characters appear in the textboxes, not even spaces. The two boxes indicating "It is a Triangle!" and "It is not a Triangle!" are both made invisible by setting their Visible properties to False. Finally, the focus is set to txtA to prepare for the next round of trial values.

## EXIT BUTTON

To code this button:

1. Double-click on the Exit button to open the Code window. If the Code window is already open, select cmdExit from the Object drop-down list.
2. Enter this code in the Click subroutine for the button:

```
End
```

## Working with the Finished Program

Try running the program, testing it for several values of *a*, *b*, and *c*. Does the program return the correct result? Is the program right when it says that a triangle can be created from three line segments? Check the math yourself.

Save the project and form files.

## QUESTIONS AND ACTIVITIES

1. What are three ways by which focus is changed in a typical Windows program?

2. When you double-click a textbox control to enter its event-handling code, what is the name of the event procedure entered? What causes the code to execute?
3. What is a parameter?
4. In the KeyPress event procedure, what is the purpose of the parameter?
5. Of the three ways for changing the focus in a Visual Basic program, which do you think is the easiest to understand and the most natural to use? Describe how that way operates and justify your opinion.
6. What happens to a textbox when it has the focus and a key is pressed, but there is no code in the KeyPress event? Assume that the KeyPress event handler has been selected for that object in the Code window, but that no code has been added to it.
7. Write the statement that would shift the focus from the txtX1 box to the txtY1 box.
8. Rewrite the following English descriptions in Basic notation:
   a) A number is less than 5 and is not -3
   b) A string is not the empty string
   c) A variable is either 7 or -2
   d) A variable is neither positive nor negative 8
9. Write an **If-Then** statement for each of these situations:
   a) If *x* is equal to 3, set *y* to 5
   b) If *y* is positive, let *c* equal *x* plus 9
10. Describe the purpose of the Visible property and how it may be changed.
11. **SetFocus** is a method and Visible is a property. What is the difference?
12. Make up three different English statements that use the word "and".
13. Under what conditions are the following statements True:
    a) Snow is falling or it is cold.
    b) The car is red and the wheels are flat.
    c) The city is near or the bus door is open.
    d) The cupcake is overdone, and the floor has been freshly waxed.
    e) It is your birthday, or it is not your birthday.
    f) It is Friday and the TV is not on.
    g) The statue is not beautiful or the road is torn up.
    h) John is happy but Mary is sad.

14. What is an empty string and can it be used?
15. Write a statement that doesn't depend on any defaults to assign the contents of the textbox txtNumber to the numeric variable *Number*.
16. State the triangle inequality.


*Section*

# Divisibility, Running Totals, and Loops

Repetition is something a computer does very well. A program can repeat a calculation over and over again with complete accuracy and reliability. A program loop is a section of code that repeats again and again. Loops are used in most computer programs. In this section, you will learn to program simple loops that you need for future projects.

Divisibility problems relate to finding whole numbers that divide evenly into other whole numbers. Divisibility comes up in problems ranging from mathematics to graphic displays.

Many programs need to add up long lists of numbers. Repeatedly adding values to a total is called "keeping a running total."

The techniques of determining divisibility and calculating running totals, as well as program loops, are used in the next project (finding prime numbers).

## Divisibility

As you write programs, you will often find yourself needing to test for divisibility. This is the process of checking to see whether one number is evenly divisible by another. For example:

15 is evenly divisible by 1, 3, 5, and 15
13 is evenly divisible by 1 and 13

Imagine that you are writing a program that displays a long listing on the screen. This material will scroll too quickly for the user to read it. The solution is to break the information up into chunks that fit on the screen. Because 24 lines of text fit on a screen, you could stop the display at 24, 48, and 72 lines. Setting this up to occur is a divisibility problem. The program will need to count each line as it is displayed. When the number of lines written is divisible by 24 (and is greater than 0), the program must pause the display.

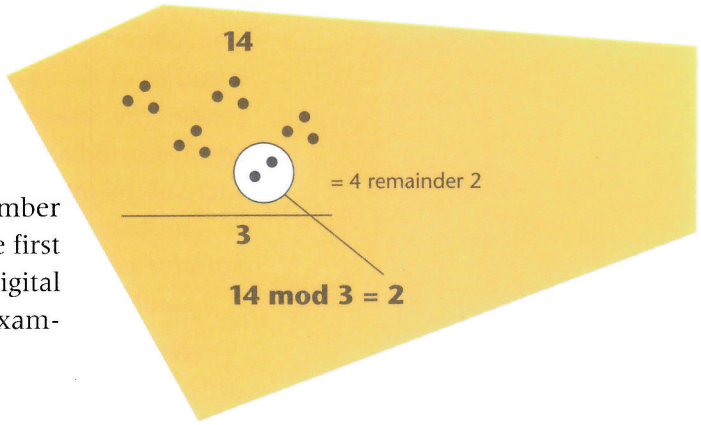
Another way you will use divisibility is to calculate a common factor or divisor. Most programming languages provide an operator to

make this easier. The **Mod** operator in Visual Basic divides one number by another and returns just the remainder of that division as its result (Figure 4-17). For example:

5 mod 3 is 2
21 mod 7 is 0
25 mod 2 is 1
24 mod 5 is 4

When the **Mod** operator returns 0, the first number is evenly divisible by the second. In other words, the first number is a *multiple* of the second. The enhanced Digital Clock example earlier in this chapter provides an example of testing for this condition.

***Figure 4-17***
***Mod operator***




## Using Counters and Running Totals

You can use assignment statements to count how often something happens. For example, you could count how often a section of code is executed or how many monsters a user killed in a game. Every time the event happened, the count would increase by 1.

You can also use assignment statements to keep a running total. For example, you could keep a running total of a checking account. Every time you deposit money, the total increases; every time you withdraw, the total decreases.

Assignment statements work for these purposes because they are not algebraic equations. For example, consider the following assignment statement:

```
C = C + 1
```

If you read this as an algebraic equation, you would be able to subtract *C* from both sides of the statement. This subtraction would leave you with 0 = 1, which does not make sense.

Now consider the statement as what it is—an assignment statement. This type of statement has the following syntax:

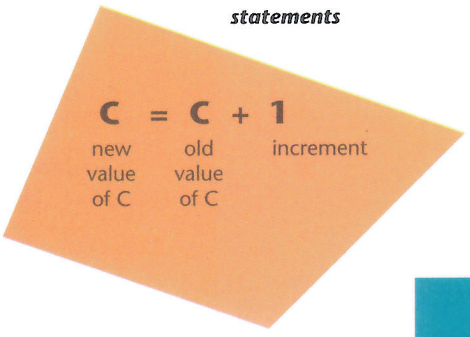
variable = expression

The statement now reads: "1 is added to the old value of *C*, and the result is assigned back to *C*. It has become the new value of *C*." This idea is shown in Figure 4-18.

Here is an example. You have initialized the value of *C* to 5 (*C* = 5). Initializing the variable means to give it a starting value. You now execute the assignment statement:

```
C = C + 1
```

***Figure 4-18***
***Reading assignment statements***



The statement fetches the current (soon to be old) value of *C*, adds 1 to it, and assigns the result back to *C*. The "new" value of *C* is 6.

To use a variable *C* as a counter, you initialize the value of *C*. Then you place the statement **C = C + 1** wherever you want to count the number of times an event happens. Every time the statement is executed, the value of the variable is incremented by 1. Incrementing means to add a constant to a value. When the increment is 1, the values increase by one each time through.

To keep a running total instead of a counter, you would use this variation in the assignment statement:

Total = Total + Value

With a running total, for example, you could add up monthly interest on a banking account. You could then choose to display annual interest by printing the total every twelve months.

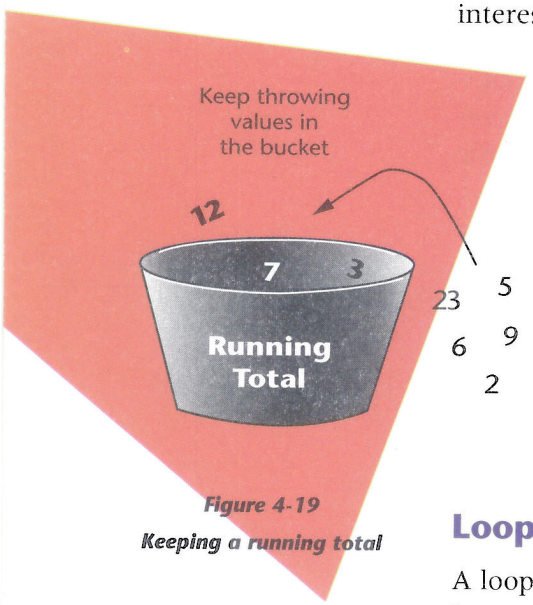



*Figure 4-19*
*Keeping a running total*

Keeping a running total is like filling a bucket with values. Start by making sure the bucket is empty (initialize the running total to 0). Keep adding values to the total in the bucket until all the values have been added. The total in the bucket is the running total of the values (see Figure 4-19).

As noted earlier, a checkbook is also a running total. The assignment statement in this case takes the form:

Bal = Bal + Transaction

Any deposit is added to the total and any check is subtracted. Deposits can be represented with positive amounts and checks with negative numbers.

## Loops

A loop is created by program statements that cause a section of code to be executed over and over again. Because a computer has the speed and accuracy to repeat the same operations quickly and accurately, problems that would defy a solution performed by hand can be solved. A bank processing checks is a good example of a repetitious task that would take too long if done by hand. The volume of checks would keep hundreds of clerks busy around the clock, processing transactions.

Problems involving running totals or counting are obvious candidates for loops. A **For-Next** loop has the following syntax:

**For** variable = expression **To** expression
    the "body" of the loop contains the block of repeated statements
**Next** variable

Figure 4-20 shows an example of a **For-Next** loop.

Here is an example of such a loop:

```
Dim x As Integer
For x = 1 To 10
   'statements to execute
in the body of the loop
Next x
```

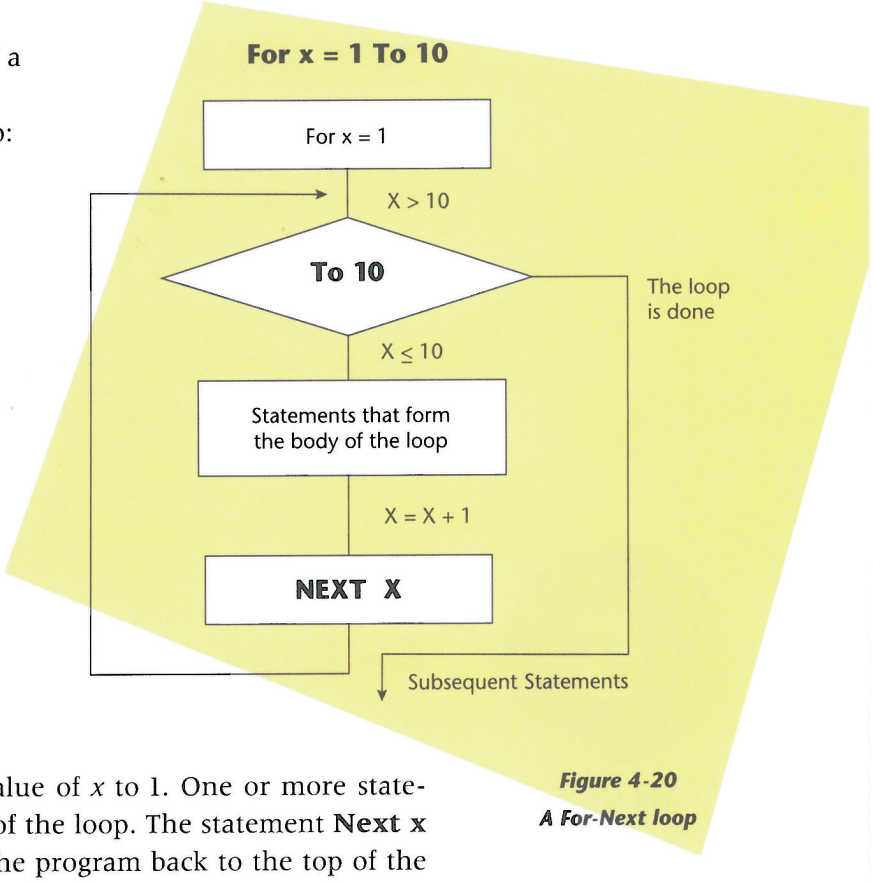



***Figure 4-20***
***A For-Next loop***

The loop starts by setting the value of *x* to 1. One or more statements can be executed in the body of the loop. The statement **Next x** adds 1 to the value of *x* and sends the program back to the top of the loop. The new value of *x* is compared with the limit of the loop, the value 10, following the word **To**. If the value of *x* is greater than 10, the loop ends, jumping to the next statement following **Next x**. The statements that make up the body of the loop can, and generally do, use the loop variable—in this instance, *x*. The actions that they perform can therefore depend on how many times the loop has already been passed through.

The following code uses the **Print** method to display a sentence ten times on the default form. The sentences are numbered, one through ten. To test this code, open a new project and copy the code into the Form_Click procedure. Run the program, then click on the form.

```
Dim x As Integer
For x = 1 To 10
   Print x; "My name is Joclyn"
Next x
```

## QUESTIONS AND ACTIVITIES

1. Write an assignment statement that represents the running total of test scores in math class. Use appropriate variable names.
2. You are keeping a running total of monthly interest. You want to print the running total at the end of every twelve months. If the number of months is represented by the variable *Months*, write an expression to test to see if *Months* is divisible by 12.
3. Write an expression to evaluate whether your age is divisible by 3.
4. Write a running total assignment statement that would keep track of a person's parking fines.

5 *Section*

# Finding Prime Numbers

The Prime Number program is designed to let you experiment with running totals and divisibility. You will also have the opportunity to use multiline textboxes. By changing a default property of a textbox, you can let it display more than one line. In a multiline textbox, text will automatically wrap around to the next line as needed, as in word processing programs. Multiline textboxes can be used to display lengthy passages of text to users while occupying a smaller amount of space on a form. They can also be used to let users enter notes and comments that your program can save and display again when it is next run.

## Starting Out

A prime number is an integer that is evenly divisible only by itself or 1. Examples include 3, 5, and 23. If you divide any of these numbers by any number other than the number itself or 1, a remainder results.

This program tests numbers entered by the user to see whether they are prime. The user is prompted to enter a number in a textbox. The program displays the divisors of the number in a multiline textbox. A textbox is set to multiline by selecting the MultiLine property in the box's Properties window. You then type **True** in the edit area of the window. Or, you can toggle from False to True by double-clicking on the property.

One way to display a large amount of data is to put it all into one long string. The string is then written into the textbox. If the MultiLine

property is set to True, the contents of the box are displayed on more than one line, if necessary. Note that although the multiline textbox can display several lines of text, it can only display one (possibly long) string at a time. It does not display multiple strings, one on each line.

If the number the user enters is prime, a label with the message "It is prime!" will be displayed. If it is not, the message "It is not prime!" will be displayed. The user can click on a command button to clear the first textbox and enter a second number to be tested. The code also instructs the program to leave the focus on the first textbox and turn off the message labels. To exit the program, the user clicks on a second command button.

You need the following controls for this program (see Figure 4-21):

- Two textboxes, one of which is MultiLine
- Four labels, including:
  - Two prompting labels
  - Two message labels
  - Two command buttons

**NOTE:**

**Multiline textboxes are not typically used for output-only fields. Later, you will learn a more typical way of displaying variable-sized lists.**

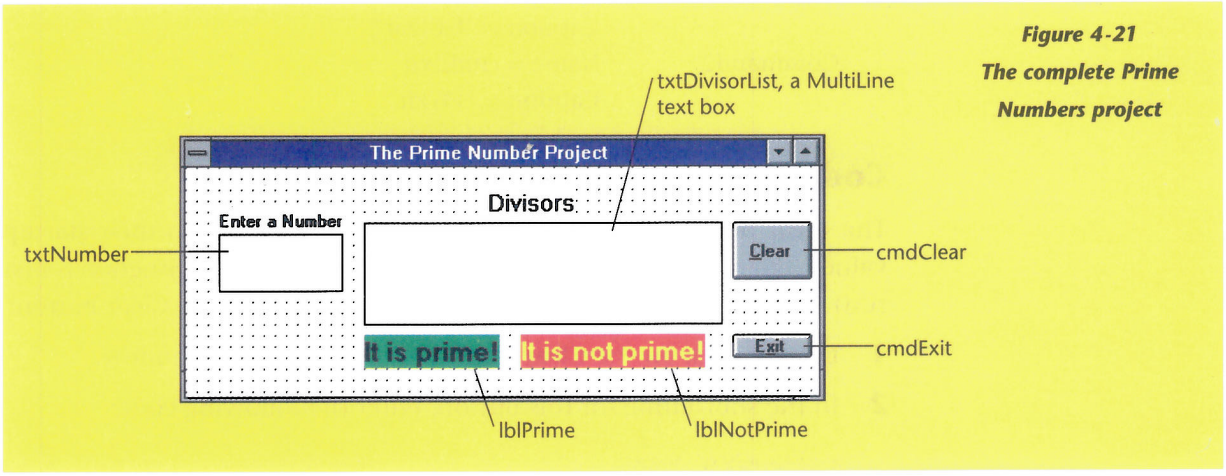



***Figure 4-21***
***The complete Prime Numbers project***

## Designing the Form

First, place all the objects on the form and change their properties as necessary. If you need more detail on the steps required, see Chapter 2.

1. Change the caption of the default form to The Prime Number Project.
2. Place the controls shown in the table on the form. Change the properties as listed.

| *Object* | *Change object properties to:* |
|---|---|
| Text1 | Name = txtNumber |
| | FontSize = 13.5 |
| | TabIndex = 0 |
| | Text = "" (empty string; clear this property) |
| Text2 | Name = txtDivisorList |
| | FontSize = 9.75 |
| | MultiLine = True |
| | Text = "" (empty string; clear this property) |
| Label1 | Caption = "Enter a Number" |
| Label2 | Caption = "Divisors" |
| Label3 | Name = lblPrime |
| | Caption = "It is prime!" |
| | AutoSize = True |
| | Visible = False |
| Label4 | Name = lblNotPrime |
| | Caption = "It is not prime!" |
| | AutoSize = True |
| | Visible = False |
| Command1 | Name = cmdClear |
| | Caption = "&Clear" |
| Command2 | Name = cmdExit |
| | Caption = "E&xit" |

## Coding for the Clear Button

The Clear button should restore the values of controls to their startup values (that is, the values that the controls have when the program is first run). To insert code into the Click event procedure for the Clear button:

1. Double-click on the Clear button to open the Code window.
2. In the subroutine for this button, enter the following code:

```
txtNumber.Text = ""
txtDivisorList.Text = ""
lblPrime.Visible = False
lblNotPrime.Visible = False
txtNumber.SetFocus
```

The first line of this code clears the text from txtNumber. The next line clears the box that contains the divisors (txtDivisorList). Line 3 makes the message label lblPrime invisible. Line 4 makes the message label lblNotPrime invisible. Finally, line 5 uses the SetFocus method to shift the focus back to the first textbox.

## Coding for the txtNumber Textbox

The code that does most of the work in this program is attached to the txtNumber textbox. When the user enters a number and presses the Enter key, this code instructs the program to find and display the divisors of that number. For this purpose, you will be using the KeyPress event as you have in earlier programs in this chapter.

To insert code into the event procedure for the txtNumber textbox:

1. Double-click on the txtNumber textbox to open the Code window. If the Code window is already open, select txtNumber from the Object drop-down list.
2. Click on the downward-pointing arrow to open the Proc drop-down list. Select KeyPress. This changes the event selected in the Code window from Change to KeyPress.
3. Now enter the following code in the txtNumber_KeyPress event procedure:

```
Dim Number As Long   ' declare the variables
Dim Sum As Long
Dim Divisor As Long
Dim DivisorList As String
If KeyAscii = 13 Then
   Number = Val(txtNumber)   ' read number
   Sum = 0      ' initialize sum of divisors
   DivisorList = ""   ' initialize list
   For Divisor = 1 To Number    ' loop of divisors
   If Number Mod Divisor = 0 Then     ' test
      Sum = Sum + Divisor      ' running total
      DivisorList = DivisorList & Str$(Divisor) & " "
   End If
   Next Divisor
   txtDivisorList.Text = DivisorList   ' display
   If Sum = Number + 1 Then    ' test if prime
      lblPrime.Visible = True
   Else
      lblNotPrime.Visible = True
   End If
   cmdClear.SetFocus    ' shift focus to Clear
End If
```

The code above makes use of a number of new programming techniques. The first four lines declare variables. The last of these declares a string variable. String variables represent sequences of characters. The program will use this string to assemble a list of divisors for the number entered.

This statement **If KeyAscii = 13 Then** begins the first **If-Then** statement. If the condition is true, the rest of the code will be executed. The code is not executed unless Enter is pressed (KeyAscii = 13).

The statement **Number = Val(txtNumber)** initializes the variable *Number* to the value the user enters in the textbox txtNumber. The program automatically converts the string type of the textbox to the long integer type of the variable.

The next line initializes the variable *Sum* to 0. This variable will be used to count the number of divisors (a running total). **DivisorList = ""** initializes the variable *DivisorList* to an empty string.

The line **For Divisor = 1 To Number** sets up a loop to repeatedly execute a series of statements. The computer can execute these statements far faster than you can. Because the *Divisor* variable has been initialized to 1, the program will test this divisor first. It will then test every number between 1 and the number entered by the user.

The line **If Number Mod Divisor = 0 Then** sets up a branch in the program with a second **If-Then** statement. The **Mod** operator divides *Number* by *Divisor* and returns the remainder of the division. If the remainder is 0, then *Divisor* divides *Number* evenly and the condition is true. In this case, the program takes the old value of *Sum*, adds the current value of *Divisor*, and assigns the result back to *Sum*.

The line **DivisorList = DivisorList & Str$(Divisor) & " "** joins strings together. The **Str$** function converts the integer value of *Divisor* into a string. The old value of the string variable, *DivisorList*, is joined to the string representation of *Divisor*, and the result is assigned back to *DivisorList*. The space within quotes joined to the end of the string inserts space between each number in the list. The ampersand operator(&) is used to join two strings.

The line **Next Divisor** adds 1 to *Divisor* and sends the program back to the top of the loop. The value of Divisor is initially set to 1. The second time through the loop, the value of Divisor will be 2. The program will continue to loop until the value of Divisor exceeds *Number* (which is the number entered by the user). At that point, the loop ends and the program jumps to the next statement outside the loop.

The line **txtDivisorList.Text = DivisorList** transfers the value of the display string to the Text property of the txtDivisorList box. This value will then be displayed.

The next line sets up a branch in the program with a third **If-Then** statement. The condition statement tests to see if the number the user entered was prime. If the number is prime, the sum of the divisors of the number is equal to the number itself plus 1. For any prime number, the loop will have found only two numbers that divide the prime number with a remainder of 0. Those are 1 and the number itself.

The program displays the lblPrime label (It is prime!) if the condition is True. Otherwise (the alternative or False branch of the statement), the program displays the lblNotPrime label (It is not prime!).

## Finishing Up

Enter the End statement in cmdExit.

Run the program and test several numbers to see if they are prime. Then save the project and form files.

## QUESTIONS AND ACTIVITIES

1. What, typically, is the last job tackled by a Clear command?
2. Describe the use of the MultiLine property of a textbox.
3. Write the statements that make a textbox appear and disappear. The name of the box is txtWarning.
4. Write the statement that will clear the display of the textbox txtNumber.
5. In what event will you find the partial statement:

```
If KeyAscii=13 Then
```

6. In you own words, describe what a loop does in a program.
7. Write a statement that will join together these components into a single string and assign the result to the textbox txtEquation.

| | |
|---|---|
| "y=" | a string literal |
| *m* | a variable representing a numeric value |
| "x + " | a string literal |
| *b* | a variable representing a numeric value |

The string displayed in the textbox should depend on the values of the variables *m* and *b*. For example, when *m* has the value 17 and *b* the value 3, the string displayed should be "*y = 17x + 3*".


The **If-Then** statement has three forms:

1. **If** *condition* **Then** *statements*
2. **If** *condition* **Then**
   *statements if true*
   **End If**
3. **If** *condition* **Then**
   *statements if true*
   **Else**
   *statements if false*
   **End If**

If the condition tested is true, the statements immediately following are executed. If the condition is false, versions 1 and 2 would not execute any statements. Version 3 would execute the statements after **Else**.

The KeyPress event procedure attached to a textbox is executed when the textbox has the focus and a text key on the keyboard is pressed. The ASCII code of the key pressed is passed to the event procedure through the variable (called a parameter) *KeyAscii*.

An And statement is true only if both the left and right expressions are true.

An Or statement is true if either the right expression is true or the left expression is true, or if both are true.

Divisibility of a number by a divisor is tested with the following statement:

**If** Number mod Divisor = 0 **Then**
  number is divisible by Divisor
**End If**

The **Mod** operator divides *Number* by *Divisor* and returns the remainder. If the remainder is 0, the *Divisor* divides *Number* evenly.

Running totals are kept with statements like this:

```
total = total + divisor
```

To count the number of times a statement is executed, use a counting statement such as:

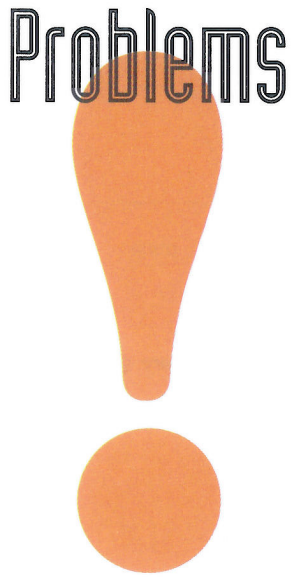
```
C = C + 1
```

The basic loop statement, the **For-Next** loop, may look like this:

```
For x = 1 To 100 Step 2
Next x
```

This statement initializes the variable *x* to 1, and increases the value by two's until the limit of 100 is exceeded.

The MultiLine property of a textbox automatically word wraps when the display line exceeds the width of the box.




1. The Movie Theater Problem

Write a program that uses a textbox to enter a person's age. If the person is 12 or older, they pay the adult fee at the movie theater, $6. If the person is less than 12, the fee is $4.00. Display the label **Adult Fee $6** or **Child Fee $4.00** based on the value entered.

Use the Visible property to turn the labels on and off. Use a button with the caption Test to test the age. Include a Clear button to clear the textbox and turn off the labels.

2. The Floor Problem

The **Rnd** function will return a random numeric value between 0 and 1. Use **Rnd** to make a random branch:

```
If Rnd < .5 Then
    'value is between 0 and .5'
Else
    'value is between .5 and 1
End If
```

Create a form with two labels and three command buttons. The captions of the two labels are **Fall Through the Floor**, and **The Floor Holds**. Name the labels **lblFall** and **lblHold**. Set the Visible property for both to False. The captions of the command buttons are **Test**, **Reset**, and **Exit**. Name the buttons **cmdTest**, **cmdReset**, and **cmdExit**. The first button, using the **If-Then** statement above, turns on one label or the other. The Reset button makes both labels invisible. The Exit button stops the program. Include a Randomize statement as the first line in cmdTest to generate a different random sequence each time the program is run.

3. The Couch Problem

**If-Then** statements are used to test strings as well as numbers. For example:

```
If txtName = "Jim" Then ...
```

will compare the contents of the textbox, txtName, with the string "Jim". If they are equal, the True branch of the **If** statement will execute.

Write a program with a textbox named **txtFurniture**. Put a label above the box with the caption **Couch or Chair?**

If the user enters "couch" or "Couch" (use an OR operator), display "The couch is $496" in a separate label. If the user enters "chair" or "Chair", display "The chair is $295".

4. What Kind of Triangle?

Write a program using textboxes to enter three lengths that may represent the sides of a triangle.

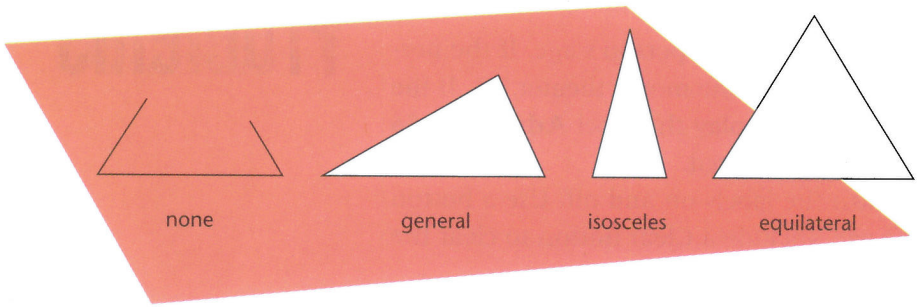



*Figure 4-22*

Use four **If-Then** statements and appropriate And statements to display a label with one of the following responses:

- None if it fails the triangle inequality.
- General if it is scalene.
- Isosceles if two sides are equal.
- Equilateral if three sides are equal.

5. Cooking a Turkey

A stuffed turkey should be cooked for 21 minutes per pound. An unstuffed turkey should be cooked for 17 minutes per pound. Display a table that shows:

Column 1: Weight of the turkey in pounds, ranging from 12 to 30
Column 2: The time to cook the unstuffed turkey
Column 3: The time to cook the stuffed turkey

The time should be printed as minutes, or hours and minutes. Use the **Mod** operator to convert the number of minutes to hours and minutes. The backslash operator divides two numbers and gives a whole number result. For instance, 7 \ 2 is 3. Minutes \ 60 is the number of hours. Minutes Mod 60 is the number of left-over minutes.

6. The Garden Fencing Problem

Write a program to calculate the cost of a rectangular wooden fence. Use two textboxes to enter the dimensions of the garden: the length and width. The amount of fence needed is the perimeter of the garden. Display the cost of fence. Each eight foot section costs $24.95.

7. The Garden Fencing Problem (Part II)

Add a textbox to the problem above. Label the box so the user will enter a "1" if the fence is rectangular, and a "2" if the fence is circular. If the fence is circular, use the "length" textbox to enter the radius of the garden. Display the cost of the fence and whether the garden is rectangular or circular. Each eight-foot-curved section of fence costs $29.95.

8. The Paint Problem

Write a program to estimate the quantity of paint needed to paint a room.

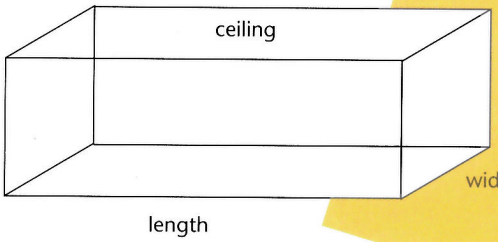
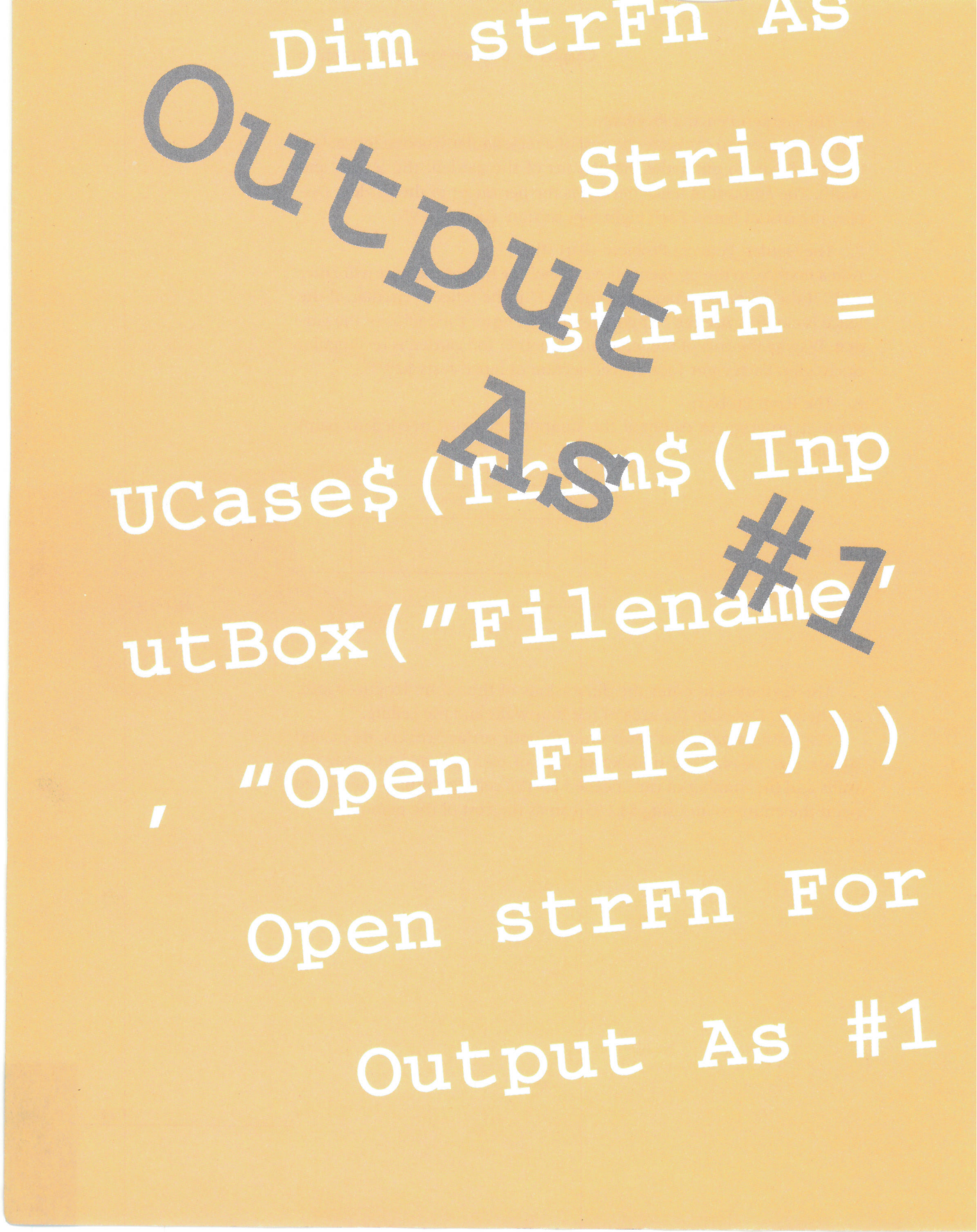



*Figure 4-23*

Use textboxes to enter the dimensions of the room: length, width, and height. Calculate the area of the four walls and the ceiling.

Assuming a gallon of paint on a smooth surface covers about 400 square feet, display the number of cans of paint needed to cover the walls and the number of cans needed for the ceiling. Display the cost to paint the entire room using $12.95/can as the cost of the paint.


Dim strFn As
String
strFn =
UCase$(Trim$(Inp
utBox("Filename
", "Open File")))
Open strFn For
Output As #1
Output
As #1