Chapter 8

Arrays and Chaos

G O A L S

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

Use the Checked and Enabled properties in menu designs.

Implement menu control arrays.

Use the **Select Case** statement in programs.

Program with arrays.

Understand the idea of chaos in a dynamic system.

Write programs to illustrate the principles of chaos inherent in the non-linear difference equation: $p_2 = r * p_1 * (1 - p_1)$.

O V E R V I E W

You were introduced to menus in Chapter 5. In this chapter, you will explore some additional features of menus, including the Enabled and Checked properties. These properties let you guide the user who is running your programs. The Enabled property lets you turn menu commands on and off. A disabled, or grayed, menu command is unavailable to the user. A checked menu command indicates that the command has been chosen, or the feature the command selects has already been turned on.

A control array allows you to handle several menu commands with a single event procedure. Regular arrays—arrays that hold numbers or strings—let you process data again and again. Values in arrays persist until the program ends.

Chaos theory is a branch of mathematics that came into being in the mid-1970s. We are just now learning the many areas to which Chaos theory can be applied. The advent of the computer has allowed the discovery and subsequent development of the methodology of Chaos theory. That methodology uses a computer to calculate the hundreds, thousands, or millions of pieces of data required to build a picture of a physical system.

This application of computers has only recently (in the last ten years) come into its own. Scientists have always used computers to analyze data, comparing the results of experiments with the results that their theories predict. Now, however, instead of manually examining large tables of numerical data for patterns, scientists enlist the graphical capabilities of computers in their search for patterns.

The Population Biology program makes predictions about population variations over generations using a simple mathematical relationship.

Menus

This section introduces some new features of menus. The Enabled and Checked properties of menu commands give feedback to users to help them make appropriate menu choices. Menu control arrays are also defined and used.

Both users and programmers benefit if commands are enabled only when it makes sense to choose them. The typical edit commands behave in this way: the Paste command is grayed (disabled) unless something has been cut or copied to the Clipboard. If you leave commands enabled even when it is inappropriate for a user to choose them, users could become confused and you will have more programming to complete. And you have an unpleasant choice to make: do you want your program to do nothing in response to an inappropriate choice, or do you want to inform the user that the choice was inappropriate?

If you decide to have nothing happen when a user clicks on an inappropriate command, a user may think your program is broken, or that a mistake has been made. If you display messages when a choice is inappropriate, then the user will soon feel frustrated and less than happy with your program. Could you defend this choice? Why let users make a choice that they cannot carry out?

Finally, if you enable commands only when appropriate, then you don't have to clutter the code that implements those commands with logic to test whether the program should respond to the user's selection. By isolating the test for appropriateness and performing it separately, the code that implements commands (the command handlers) can always be written with the assumption that the program is going to perform the action associated with that menu command.

On the other hand, a dimmed command could be frustrating if the user doesn't understand why it is dimmed. At times, leaving a menu command enabled and displaying an error message explaining why the command is currently inappropriate is a better choice than disabling the command. As you gain more experience with programming, you will learn when to disable commands and when to display error messages.

Placing checkmarks next to menu commands gives clues to users about their current settings and how to change them. Some commands toggle a condition on and off; placing a checkmark next to such a command indicates that the condition is currently on.

In this section, menu control arrays are also defined and used. Menu control arrays let you write a single command handler for a group of commands. These control arrays benefit you, the programmer, only; they don't offer users any immediate benefit. A running Visual Basic program provides no visual clue that it uses menu control arrays.

Checked and Enabled Properties

When a menu command is enabled, it can be chosen from the menu. If the menu command is disabled, it is dimmed and cannot be clicked.

Professional programmers typically disable commands when users cannot select them. Setting a command's Enabled property to True enables it. Setting the property to False disables it.

Typically, you do not disable a command during project design. If a user can never select the command, why include it in the program? Commands are disabled as the program runs, if a certain condition is not met.

For example, suppose that you open the Edit menu in Microsoft Word. Unless you have placed some material on the Clipboard that could be pasted, the Paste command is dimmed. Dimming the command tells you, the user, that there is no material ready to be pasted. As soon as you cut or copy some text, however, the Paste command is enabled.

To display a check next to a command, set the command's Checked property to True. To remove a check from a command, set the command's Checked property to False.

Suppose that you had a command on a View menu for the Toolbar. This command is a toggle. Choosing it shows or hides the toolbar. When the toolbar is displayed, the menu command appears with a checkmark; choosing it hides the Toolbar and unchecks the menu command. When the toolbar is not shown, the menu command appears unchecked; choosing it shows the toolbar and checks the menu command.

CONTROL ARRAYS

You haven't used control arrays in your programs yet, but this may not be the first time you've seen the term. If you have ever given the same name to more than one control, Visual Basic has prompted you to create a control array. In a control array, more than one control shares the same event procedure. The controls in a control array all have the same name; they are distinguished by a numerical index, much like the elements of a data array declared with the **Dim** statement.

To use a control array, you have to be able to distinguish among the elements of the array. The solution is to give each control in a control array a different number, or index. The Index property serves this purpose. Elements of control arrays have unique index values. Controls in a control array share the same event procedures, which have an extra parameter: the Index of the control in the array that generated the event.

Menu Control Arrays

In a menu control array, you give two or more commands the same name. Commands with the same name share the same Click event handler. Each menu item receives a unique index number, which you assign in the Menu Design window. Commands assigned the same name must be at the same level (such as in the same submenu) and next to each other. The captions of the menu commands—the names by which the user knows the commands—can be whatever you want them to be. Only the Name property of the commands—the means by which you refer to the items in code—must be the same.

When the program is running and a user clicks on a menu command in a control array, the index number is passed to the subroutine as an index parameter. This number is used to determine the program's action.

What Shall I Wear?

This program answers that tough morning question: What shall I wear?

A menu displays a selection of bottoms: three shorts and two pants. When a user clicks on a command, a list of appropriate shirts is displayed. The program makes the assumption that the clothes are dirty after a person has worn them. As each menu item is clicked, it is both checked and disabled.

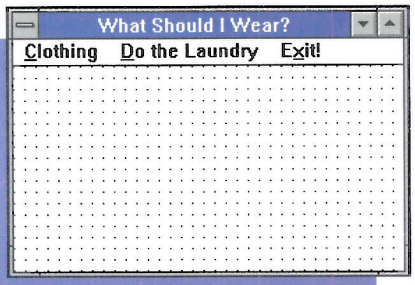


Figure 8-1
The What Shall I Wear? form

When the laundry is done, the menu lets the user reset all the choices, enabling and unchecking each item. Figure 8-1 shows the completed form for the project.

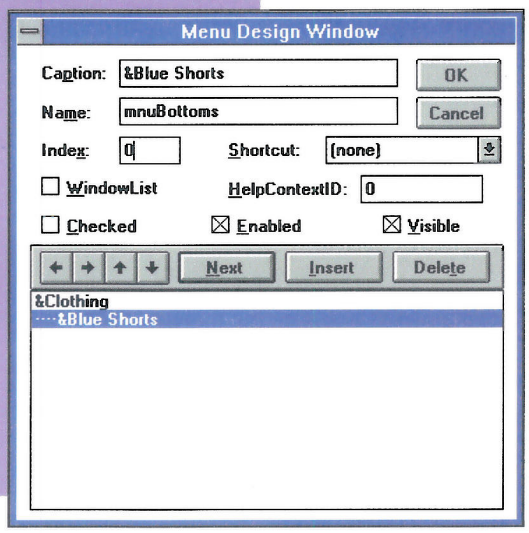
DESIGNING THE MENU

Follow these steps to create the menu.

1. Start Visual Basic. If Visual Basic is running, select New Project from the File menu.
2. Change the caption of the default form to **What Shall I Wear?**

3. Select the form. Click on the Menu Design button to open the Menu Design window.
4. For the first menu command, enter the caption **&Clothing** and the name **mnuClothing**.
5. Click Next and the right arrow to create a submenu.
6. Enter the caption **&Blue Shorts** and the name **mnuBottoms**. For the Index entry, enter 0. Note the Checked and Enabled properties. These will be set while the program is running. See Figure 8-2.

Figure 8-2
Beginning the menu design



7. Click Next and enter the caption **B&lack Shorts** and the name **mnuBottoms**. Enter an Index value of 1.
8. Click Next and enter the caption **&Plaid Shorts** and the name **mnuBottoms**. Enter an Index value of 2.
9. Click Next and enter the separator, a single hyphen. Though this is not a real menu item, it must have the same name as the other menu items. It must also have an index number. Enter **mnuBottoms** for a name and 3 for the Index value. See Figure 8-3.
10. Click Next and enter the caption **B&rown Pants** and the name **mnuBottoms**. Enter an Index value of 4.

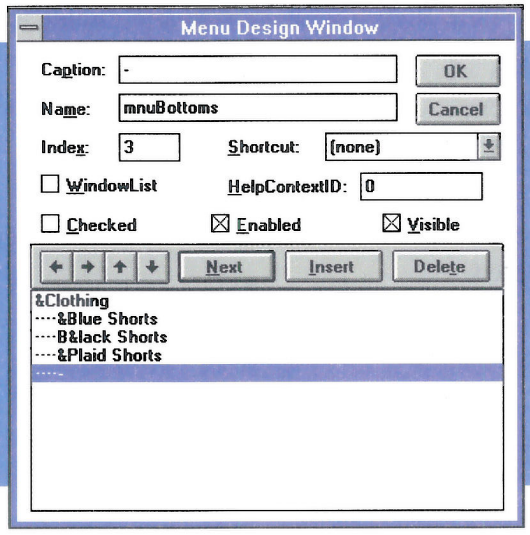


Figure 8-3
More entries in the menu

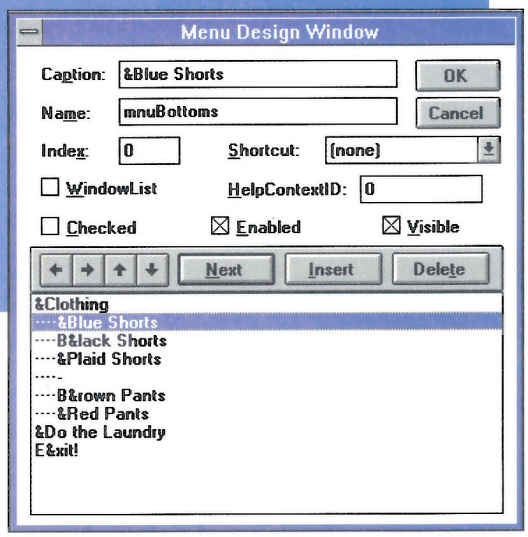


Figure 8-4
Completed menu

11 Click Next and enter the caption **&Red Pants** and the name **mnuBottoms**. Enter an Index value of 5.

12 Click Next and the left arrow. Enter the caption **&Do the Laundry** and the name **mnuLaundry**.

13 Click Next. Enter the caption **E&xit!** and the name **mnuExit**. An exclamation point is added to menu bar commands that perform tasks, rather than displaying a submenu.
Figure 8-4 shows the completed menu.

ENTERING THE CODE

The following steps create a fully functioning menu. By the end of the steps, the program will still not prompt the user regarding what shirt to wear with the selected bottom. You will finish the program in the exercises at the end of this section.

Follow these steps to enter the code.

1 Click Clothing in the main menu bar.

2 Click any of the entries shown in the submenu. The items share the same event procedure, the Click event.

3 The top line of the form shows one of the effects of creating a menu control array. *Index* is passed as a parameter to the event handler. The value of *Index* corresponds to the numbers assigned in the Menu Design window.

```
Sub mnuBottoms_Click (Index As Integer)
```

4 Enter lines to mark the item clicked with a check and disable it:

```
'—Mark clicked item as checked, and disable
mnuBottoms(Index).Checked = True
mnuBottoms(Index).Enabled = False
```

5 Enter lines to handle the only choice appropriate when all the clothes have been worn: Do the Laundry. Each of the menu items is unchecked and enabled:

```
'—Do the laundry
Dim x As Integer
For x = 0 To 5
    '—Get rid of check marks and enable each item
    mnuBottoms(x).Checked = False
    mnuBottoms(x).Enabled = True
Next x
```

6 In the mnuExit command handler, enter the command **End**.

7 Run the program. Click on Plaid Shorts. Click on Clothing. Note the item Plaid Shorts is dimmed and checked.

8 Try to click on Plaid Shorts again.

9 Click on each of the other menu items.

10 Click on Do the Laundry. Note the results.

11 Try to click on the separator.

12 Save the project and form files.

QUESTIONS AND ACTIVITIES

1. Use an **If-Then** statement in the **mnuBottoms** menu handler of the form:

   ```
   If Index = 0 Then ...
   ```

   to display at least one appropriately colored shirt for each bottom in the menu. Use a message box to display the color of the shirt to wear with each bottom.

2. Add a submenu to the Blue Shorts command:
   a) Select the form.
   b) Open the Menu Design window.
   c) Select the command below Blue Shorts.
   d) Click on the Insert button, then click on the right arrow.
   e) Enter the caption **White Shirt**, the name **mnuShirt**, and an Index of 0.

f) Click on the Insert button, then click on the right arrow (if necessary) to put this entry in the same submenu as White Shirt.
g) Enter the caption **Blue Shirt**, the name **mnuShirt**, and an Index of 1.
h) Without running the program, click on Clothing. Click on Blue Shorts. Click on White Shirt.
i) Run the program. Click on Clothing. A run-time error is generated: a command with a submenu cannot display a checkmark.
j) Delete the submenu of Blue Shorts.

3. Rename the Clothing menu to Bottoms. Create a new menu called Tops. In the Tops menu, create a submenu with the following entries:

   White Shirt
   Blue Shirt
   Black Shirt
   Green Shirt
   Reset

   Give each of the first four commands in the submenu the same name and index values 0 through 3. As each command is clicked, disable it until the Reset item enables all of the commands again. Give Reset its own name and event procedure.

The Planet Demo Program

The Planet Demo Program calculates what a person's weight would be if measured on another planet of the solar system. The program uses a menu control array to choose a particular planet. In the Click event procedure, you will use a **Select Case** statement to identify which planet was chosen. The person's name and weight are entered through a dialog box, which is another form that you will design and display. This project is the first in which you will use more than one form.

The main form of the program has a very simple appearance (see Figure 8-5).

The menu bar shows three commands: Person, Planet, and Exit. When a user clicks on Person, a dialog box opens, prompting the user to enter the name and weight of a person. The user can enter their own weight, or that of someone else.

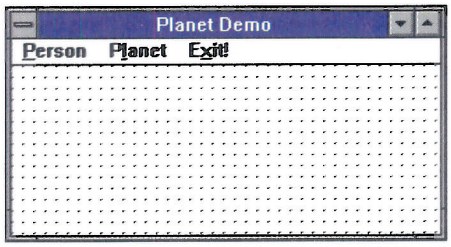


Figure 8-5
Finished Planet Demo program

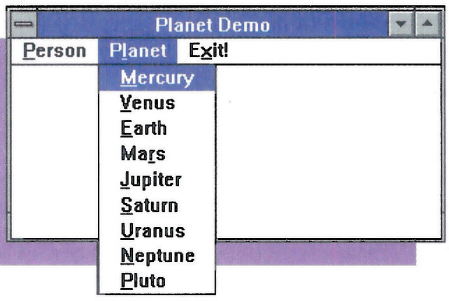


Figure 8-6
Planet submenu for the Planet Demo program

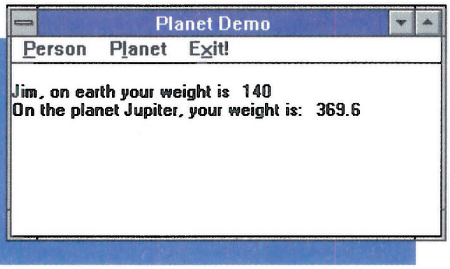


Figure 8-7
Displaying values calculated by the Planet Demo program

Next the user clicks on Planet. A submenu listing nine planets opens (see Figure 8-6).

Once the planet has been picked, the program multiples the person's earthly weight by a factor that adjusts the weight for the chosen planet. The two values are displayed on the main form, as shown in Figure 8-7.

Starting Out

Before you build the simple form for this project, you should focus on the Visual Basic features you will use. These include the **Select Case** statement, dialog boxes, showing and hiding forms, and the **Print** method.

SELECT CASE STATEMENT

This program has nine possible branches, one corresponding to each planet. One way to handle this is with a series of **If-Then** statements, one for each planet. You might write one of the statements this way:

```
If Index = 1 Then
   '—Mercury
   W = Weight * .27
End If
```

Another way to handle this is to use a **Select Case** statement. This statement evaluates an expression and, based on the calculated value, branches to a particular set of statements. The basic syntax for the **Select Case** statement is:

Select Case *expression*
  Case *value1*
    statements executed if value1 matches expression
  Case *value2*
    statements
  ....
  Case Else
    statements executed if the expression doesn't match any of the Values
End Select

In the Planet program, the *expression* is the value of the index parameter. Each valuelist is of the simplest form—a single integer constant giving the index for a planet. The case statements perform the conversion from earth weight to weight on that planet.

The **Select Case** statement permits even greater flexibility than the above syntax indicates. For example, any *value* can be an expression taking either of these forms:

```
expression1 To expression2          ' (for example, 17 To 25)
comparison-operator expression1     ' (for example, > "John")
```

You may also use a list of values after the **Case** keyword, as an alternative to using multiple **Case** groups with identical statements. Finally, the **Case Else** part is optional. For the details of the **Select Case** statement, consult the Visual Basic Help system.

The **Select Case** statement doesn't let you express anything that you could not already express using only **If-Then-Else**. However, **If-Then-Else** is a very general tool, whereas **Select Case** is tailored for choosing among a range of alternative actions based on the current value of a single expression. In such situations, the **Select Case** statement lets you write more concise code that is easier to read.

DIALOG BOXES

A dialog box is a kind of window that is typically used to collect a specific kind of information from the user, or to perform a specific function. Unlike the main window of a program, dialog boxes appear only when needed, and are closed when they have fulfilled their limited purpose. When you save a program, for example, a dialog box appears (see Figure 8-8). Dialog boxes such as this one contain labels, command buttons, textboxes, and comboboxes.

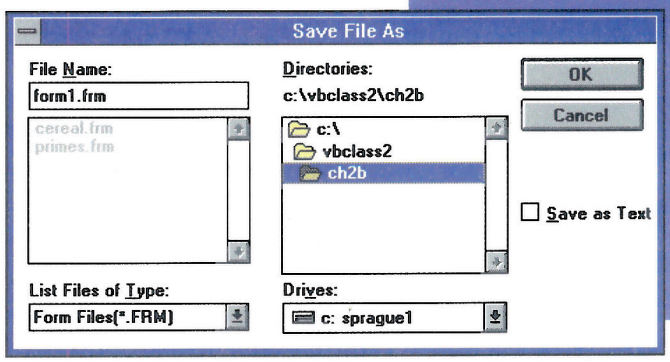


Figure 8-8
The Save As dialog box

While this dialog box is open, you cannot use any other window of Visual Basic; you must first click OK or Cancel to close this dialog box before you can resume work. A dialog box that requires you to complete your interaction with it before you can resume working with a program is called a modal dialog box. Not all dialog boxes are modal. Those that are not are called modeless dialog boxes. For example, when you choose the Find command in Microsoft Word or Windows Write to search for text, the Find dialog box lets you work in the application while it is open. You close the dialog box when you are finished searching.

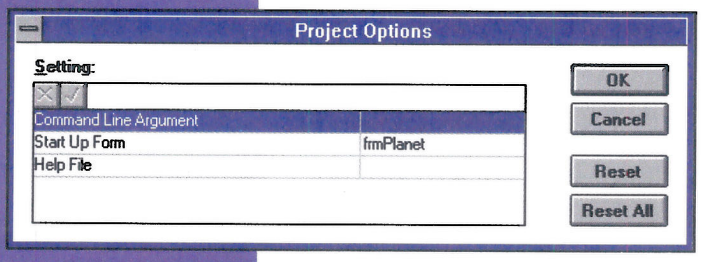
The InputBox and MessageBox functions create dialog boxes that you can display in Visual Basic. These dialog boxes are not forms; you do not design their layout, which is very general but also very simple. Often, however, your programs will need to collect from the user more than one piece of information before they can perform a task. Displaying a large number of InputBoxes in succession would annoy the user. To address this need, Visual Basic lets you design and use forms that behave like either modal or modeless dialog boxes. The Planet program uses a form as a modal dialog box to collect the person's name and weight.

SHOWING AND HIDING FORMS

The form that is displayed first and automatically when a Visual Basic program is executed is called the start-up form. If your project contains more than one form, the start-up form is by default the first form added to the project.

You can change the default start-up form if necessary. To do so, select Project from the Options menu. For example, in Figure 8-9, the form frmPlanet is identified as the start-up form. This is the first form a user will see when the program is run.

Figure 8-9 Identifying the start-up form



To make any other form appear while the program runs, it must be loaded and displayed. Both are accomplished by the **Show** method. In the Planet program, you want the dialog box frmPerson, which is a form, to appear first. Therefore, you enter this line of code:

```
frmPerson.Show 1
```

The argument 1 indicates that the form frmPerson should be displayed in a modal way: the main form of the Planet program cannot be accessed until the frmPerson form is closed. (If you do not know what the word "modal" means, read the previous section about dialog boxes.) Using an argument of 0 instead of 1 would tell Visual Basic to show the frmPerson form in a modeless way: Visual Basic would make frmPerson visible and active, but the user would be able to switch back to the main form while frmPerson was still open.

To close the frmPerson dialog box and return to the original form, you execute this command:

```
frmPerson.Hide
```

The **Hide** method removes the form from the screen, but does not remove it from memory. The next time your program executes the **Show** method of the frmPerson form, the dialog box will appear on the screen more rapidly, because it does not have to be loaded from disk and constructed in memory by Visual Basic. If your program had many different dialog boxes, most of which were not often used, it would be wasteful for them to occupy memory just because they had been used once. To remove the frmPerson from memory, you would use this statement:

```
Unload frmPerson
```

This statement causes the form's QueryUnload event and then the Unload event to be generated before the form is removed from memory. For more information about the Unload event, consult the Visual Basic Help system.

THE PRINT METHOD

The output for the procedure is displayed directly on the default form using the **Print** method. A statement that uses this method has the syntax:

object.**Print** *string or value*

If you do not specify an object, the **Print** method prints on the default form. You can use the **Print** method to print on forms and picture boxes, as well as in the Debug window. The **Print** method uses the current color and font for the object, and writes its output at the coordinates given by the object's CurrentX and CurrentY properties.

The **Print** method is used to display successive lines of text. It is in many ways a throwback to the era of character mode programs. After the above statement using the **Print** method is executed, CurrentY will be increased by the value of *object*.TextHeight so that a new line can be written beneath the one just printed.

Creating the Menu

The key parts of this program are the menu bar and the dialog box. Let's start with the menu:

1. Start Visual Basic. If Visual Basic is running, choose New Project from the File menu.
2. Change the caption of the form to **Planet Demo**. Change the name of the form to **frmPlanet**.
3. Select the form and open the Menu Design window.

4 For the first command, enter the caption **&Person** and the name **mnuPerson**. This command calls the dialog box, which is then used to collect the name and the weight of the person.

5 Click Next and enter the caption **P&lanet** with the name **mnuPlanet**. This command displays a submenu of the nine planets.

6 Click Next, then the right arrow to create a submenu. Enter the caption **&Mercury** and the name **mnuPlanets**. Set the Index to 0. Note the plural form of Planet.

7 Click Next and enter the following captions, names, and Index values. Because all the entries in the submenu have the same name, you have created a menu control array.

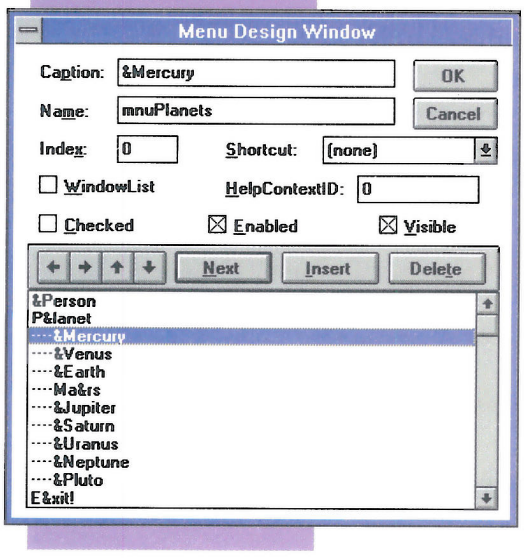
Caption	*Name*	*Index*
&Venus	mnuPlanets	1
&Earth	mnuPlanets	2
Ma&rs	mnuPlanets	3
&Jupiter	mnuPlanets	4
&Saturn	mnuPlanets	5
&Uranus	mnuPlanets	6
&Neptune	mnuPlanets	7
&Pluto	mnuPlanets	8

8 Click Next, then the left arrow. Enter the caption **E&xit!** and the name **mnuExit**. See Figure 8-10 for the complete menu.

9 Save the project and form files.

Menu control arrays, just like all other arrays, have subscripts in parentheses. Although the parentheses don't appear in the name box of Figure 8-10, they do appear in the listing of objects in the Code window (see Figure 8-11).

Figure 8-10
Completed menu for the Planet Demo program



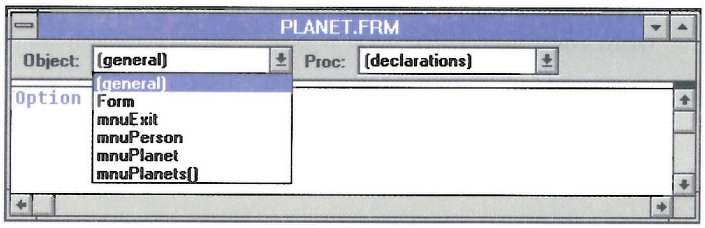


Figure 8-11
Identifying menu control arrays in the Code window

Creating a Code Module

To build a dialog box for the Planet program, first determine what information is to be collected, then determine how that information is to be communicated to the rest of the program.

The dialog box collects a person's name and weight. The name is a string, and the weight is an integer. To communicate between the dialog box and the Planet form, the variables are declared in a code module. A global declaration in a code module makes the value of a variable available to all forms in a project.

Create the code module by following these steps.

1. Select New Module from the File menu.
2. Enter the lines to declare the two global variables (the *g* prefix indicates a global variable):

```
Global gstrName As String
Global gWeight As Integer
```

3. When the project is saved, the module will be saved separately with a name you pick. Before it is saved, the module window uses its default name for a caption. See Figure 8-12.
4. Save the code module.

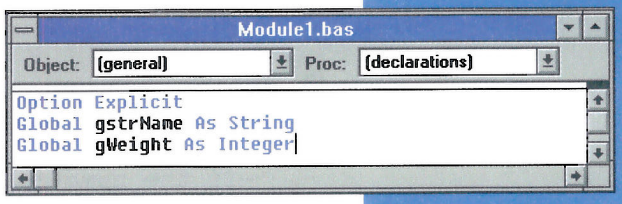


Figure 8-12
Caption of the module window is the name of the module file

Setting Up the Dialog Box

To create the dialog box form, start by adding a form to the project. The dialog box requires two textboxes, two labels, and a command button.

1. Select New Form from the File menu.
2. Change the caption of the form to **Person's Name and Weight**. Change the name of the form to **frmPerson**.
3. Add two textboxes, **txtName** and **txtWeight**. Clear their text.
4. Add two labels with the captions **Enter the name:** and **Enter the weight:**. Set the FontSize to 9.75. See Figure 8-13.

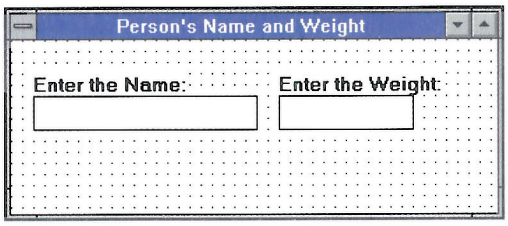


Figure 8-13
Adding labels to the dialog box

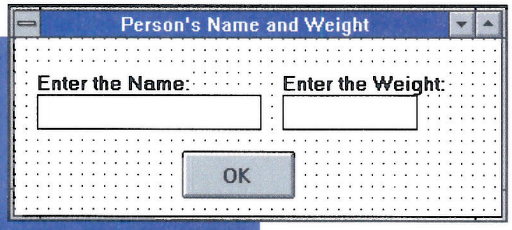


Figure 8-14
The completed form

5 Add a command button. Change the name of the button to **cmdOK** and the caption to **OK**. See the complete form in Figure 8-14.

6 Save this form.

Adding Code to the Dialog Box

The code in the command button reads the textboxes and assigns their values to the global variables. It also manages shifting focus back to the start-up form and enabling the Planet menu command. Follow these steps to add code to the command button:

1 Double-click on the command button cmdOK.

2 Enter these lines to read the values of the textboxes:

```
'—Read the name and weight boxes
gstrName = txtName
gWeight = Val(txtWeight)
```

3 If either the name or weight is blank, the dialog box hasn't done its job. Focus is returned to txtName. Enter these lines:

```
'—If the name is blank or the weight is blank
'set the focus back to txtName
If gstrName = "" Or gWeight = 0 Then
   txtName.SetFocus
   txtWeight = ""
```

4 If a user provides values for both name and weight, the program hides the dialog box and focus returns to the main form. In addition, the Planet command on frmPlanet, the start-up form, is enabled.

```
Else
   frmPerson.Hide
   frmPlanet.mnuPlanet.Enabled = True
End If
```

5 Save the form.

The code in the fourth step can access a menu command from the other form by including the form name in the statement. The syntax of the statement is:

FormName.MenuItem.Enabled = True

Writing the Code for Form Planet

The code in this form has four jobs.

- Declare an array for the planet names. These names are used in the display.
- Initialize the elements of the planet array to the names of the planets.
- Manage the appearance and disappearance of forms and menu items.
- Calculate and display the person's weight on the planet clicked.

Follow these steps to write the code for frmPlanet:

1. Select frmPlanet in the Project window. Click on the View Code button.
2. In the general declarations section, add this line to declare an array for the planet names:

```
Dim PlanetName(0 To 8) As String
```

   The subscripts match the index numbers assigned in the menu control array. Index 0 corresponds to subscript 0, Mercury.

3. Open the Form_Load procedure for frmPlanet and enter these lines to initialize the PlanetName array:

```
'-Initialize planet names
PlanetName(0) = "Mercury"
PlanetName(1) = "Venus"
PlanetName(2) = "Earth"
PlanetName(3) = "Mars"
PlanetName(4) = "Jupiter"
PlanetName(5) = "Saturn"
PlanetName(6) = "Uranus"
PlanetName(7) = "Neptune"
PlanetName(8) = "Pluto"
```

4. You don't want users choosing a planet before they enter a name and weight. That means you need to enter a final line in the Form_Load procedure that disables the Planet command in the menu bar:

```
mnuPlanet.Enabled = False
```

5. The menu command mnuPerson displays the dialog box. You display the box and shift focus to it by using the **Show** method. Enter this line in the mnuPerson_Click() procedure:

```
frmPerson.Show 1
```

6. Clicking on the **Exit!** command should stop the program. To have this happen, insert **End** in the mnuExit_Click() procedure.

7 Open the Code window and select mnuPlanets() from the Object drop-down list. The parentheses at the end of the name indicate a menu control array. Enter the line declaring a variable for the adjusted weight:

```
'—Declare variable for adjusted weight
Dim AdjWeight As Single
```

8 Add the **Select Case** statement in the same subroutine. The value of *index* corresponds to a menu item. Each menu item is a planet name. Each branch in the **Select Case** statement calculates an adjusted weight for one planet:

```
'—Select Case statement to pick out planet
Select Case index
'—For each planet, calculate weight and
'check the menu item.
   '—Mercury
   Case 0
      AdjWeight = gWeight * .27
   '—Venus
   Case 1
      AdjWeight = gWeight * .86
   '—Earth
   Case 2
      AdjWeight = gWeight
   '—Mars
   Case 3
      AdjWeight = gWeight * .37
   '—Jupiter
   Case 4
      AdjWeight = gWeight * 2.64
   '—Saturn
   Case 5
      AdjWeight = gWeight * 1.17
   '—Uranus
   Case 6
      AdjWeight = gWeight * .92
   '—Neptune
   Case 7
      AdjWeight = gWeight * 1.44
   '—Pluto
   Case 8
      AdjWeight = gWeight * .33
End Select
```

9 Once the calculation is made, you want the planet name checked on the menu. Enter this line to check the planet name:

```
'—Check the planet selected
mnuPlanets(index).Checked = True
```

10 The results are displayed directly on the form. You use the **Print** method to print the person's name, earth weight, clicked planet, and equivalent weight on that planet. Enter these lines to display the results:

```
'—Display results on form
Cls
Print
Print Trim$(gstrName);
Print ", on earth your weight is "; gWeight
Print "On the planet " & PlanetName(index) & ", your weight is: "; AdjWeight
```

The form is cleared, a line is skipped and the person's name is printed. The **Trim$** function deletes any extra spaces around the name, and the **Print** method prints the result on the form. The semicolon means the display stays on the same line. Normally, it would automatically shift to the next line. The earth weight is printed. On the next line, the planet name is printed from the *PlanetName* array, initialized in the Form_Load procedure. Finally, the adjusted weight is printed.

11 Run the program. Click on Person. Enter a name and a weight. Click OK.

12 Click on the Planet menu. Click on a planet. Click on each of the planets to check your program.

13 Save the project file, the form files, and the code module file.

QUESTIONS AND ACTIVITIES

1. Explain some of the important properties of a menu item.
2. How is a menu command like a command button? To what events does a menu command respond?
3. How are the different menu commands differentiated from each other in a menu control array?
4. People between the ages of 0 and 15 cannot receive a license to drive in the state of Illinois. People ages 16 and 17 can receive an Illinois license with their parent's permission. People above the age

of 18 can apply for a license without parental consent. If the age of an applicant is represented by the variable *Age*, write the **Select Case** statement that displays an appropriate message box for each of the three possibilities. Are there other possibilities? If so, make sure your program segment handles those.

5. Rewrite the code for mnuPlanets to eliminate the **Select Case** statement and replace it with nine **If-Then** statements.
6. Rewrite the code for mnuPlanets to eliminate the **Select Case** statement and replace it with a single assignment statement. Declare an array of Single to hold the multipliers for each planet. Initialize the array in the Form_Load procedure, just like the *PlanetName* array.
   In the mnuPlanets code, multiply the person's weight by the multiplier in the multiplier array and assign it to *AdjWeight*.
7. Remove the **Trim$** function from the mnuPlanets code and run the program. Note the change in the display.
8. Enlarge the frmPlanet and add a listbox. Rewrite the mnuPlanets code to display the results in a listbox instead of on the form. Don't clear the listbox between each choice.
9. Change the mnuPlanets code to disable each planet in the menu when it is clicked.

Section

Arrays

One of the data structures implemented on the earliest computers was the list. Lists are everywhere. A computer is dependent on lists. There are lists of files in a computer's directories, lists of commands in programs, lists of data in memory cells. Many processors have special commands that facilitate working with lists. Programming languages have special list-handling capabilities.

A phone book is a list. A file of recipe cards is a list. An individual recipe is a list of ingredients and cooking instructions. It's easy to turn real-life lists into computer lists.

Sometimes a distinction is made between internal lists and external lists. An external list is a list maintained in a file on a disk. An internal list is a list maintained in random access memory. An internal list is

called an array. Arrays have already used in previous programs: most often they have been referred to as tables.

Here is a typical statement declaring an array:

```
Dim List(100) As String
```

In this statement, the variable *List* represents 100 string values (the array). This array can store 100 items, which are called elements or members. The elements in this example are of the String data type.

To access a specific element in an array, you need to be able to refer to it. To do so, you use a subscript, which is the number in parentheses. The subscript is a whole number that indicates the position of the element in the array.

The first element of the array is referred to as List(0). If an array is declared (or dimensioned, Dim for short) with a single value, the starting subscript is 0. The eleventh element of the array (starting from 0) is List(10).

Another way to declare an array is to specify the first and last subscript:

```
Dim List(51 To 100) As Single
```

In this example, the array holds 50 elements. The valid subscripts range from 51 to 100. You can also specify subscripts that are negative:

```
Dim List2(-5 To 5) As Integer
Dim List3(-10 To -1) As Single
```

To place a string into the forty-third element of an array, you would use the statement:

```
List(43) = "Here is a sample string."
```

To put the twenty-first element of an array into a textbox named txtDogName, you would use the following statement:

```
txtDogName = List(21)
```

QUESTIONS AND ACTIVITIES

1. Name five household management projects that could be handled with an array.
2. Write the statements to declare the following arrays of values:
   a) 30 whole numbers
   b) 31 whole numbers starting with a subscript of -15
   c) 20 strings starting with a subscript of 21
   d) 5 currency amounts starting with a subscript of 10

3. Describe the following arrays in words:
   a) Dim Shoes(1 To 50) As String
   b) Dim Names(20) As String
   c) Dim Fees(1 To 5) As Currency
   d) Dim Adjustments(10 To 20) As Single

The School Cafeteria Project

This program features arrays. Thirty students have been polled about the quality of cafeteria food. They have ranked the food 1 for *horrific* to 5 for *excellent*. Their rankings are stored in an array. The program reads the array and tallies the rankings. The tallies are kept in an array.

The program illustrates the counting principle ($C = C + 1$) described in Chapter 4 in the context of arrays.

Planning the Program

To plan the project, break it down into its component parts. You need to:

- Initialize the array of student rankings. Typically, this information would be read from a file. In this program the array is initialized in the Form_Load procedure.
- Display student rankings in a listbox.
- Tally the results.
- Display the results. The results are displayed in a bar graph, drawn in a picture box. The **Line** method draws the rectangles needed to build the graph.

  Figure 8-15 displays the completed program.

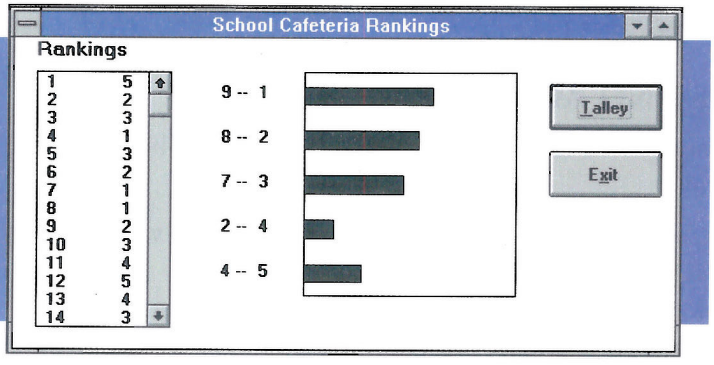


Figure 8-15
Completed School Cafeteria program

Setting Up the Form

Follow these steps to set up the form.

1. Start Visual Basic. If Visual Basic is running, choose New Project from the File menu.
2. Change the name of the form to **frmCafeteria**. Change the caption to **School Cafeteria Rankings**.
3. Place a List box on the form. Change the name to **lstRankings**. This box displays the student responses to the poll.
4. Put a Picture box on the form. Change the name to **pctGraph**. This box displays a bar graph of the tallies.
5. Put two command buttons on the form. Change the captions to **&Tally** and **E&xit**. Change the names to **cmdTally** and **cmdExit**. See Figure 8-16.

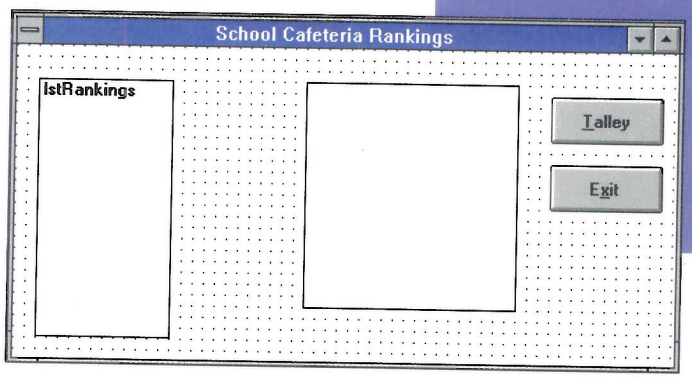


Figure 8-16
The form in progress

6. In the space between the List box and the Picture box, create a control array of labels. Draw a single label, set AutoSize to True, and position it between the boxes in the upper fifth of the space. Change the name to **lblTally**.
7. Select the label and copy with Ctrl+C. Paste with Ctrl+V. When Visual Basic prompts you about control arrays, click on Yes.
8. Paste again and again until there are five evenly distributed labels. These labels will display the ranking, 1 to 5, and the number of responses in each. See Figure 8-17.

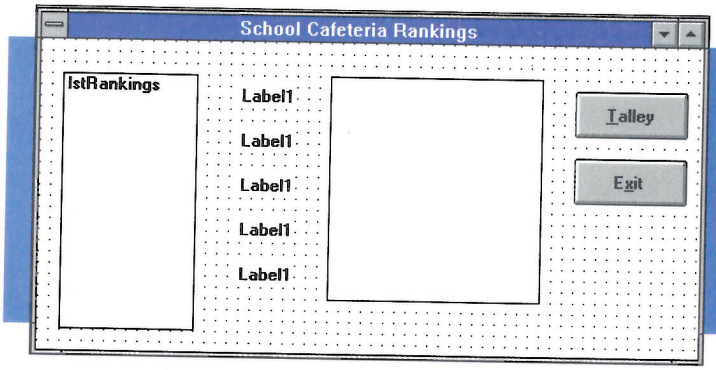


Figure 8-17
Placing the ranking labels

9 Finish the form by putting a label above the List box. The caption should read **Rankings**.

10 Save the project and form files.

Entering the Code

As you write the code, start with the smaller routines:

1 Click on the View Code button in the Project window.

2 In the general declarations section, enter the following declarations:

```
Dim Ranking(1 To 30) As Integer
Dim Tallies(1 To 5) As Integer
```

3 In the Form_Load procedure, enter these lines to initialize the Tallies arrays declared above.

```
Dim x As Integer
'—Initialize the tallies
For x = 1 To 5
   Tallies(x) = 0
Next x
```

4 To initialize the Ranking array, enter a single line:

```
Ranking(1) = 5
```

5 Select the line. Press Ctrl+C to copy the line. Press Ctrl+V 29 times to paste the line into the routine. Move the cursor to each line in the routine and change the subscripts and the values assigned to each array element. The subscripts should range from 1 to 30. The rankings should be random whole numbers between 1 and 5.

```
'—Initialize the 30 student responses
Ranking(1) = 5
Ranking(2) = 2
Ranking(3) = 3
Ranking(4) = 1
...
Ranking(28) = 1
Ranking(29) = 2
Ranking(30) = 2
```

The middle lines have been omitted.

6 Enter the **End** statement in the code for cmdExit.

7 Open the cmdTally_Click procedure. Enter the lines to declare a local variable for the loop and clear the listbox for the rankings:

```
Dim x As Integer
lstRankings.Clear
```

8 The next section of code does two jobs. It displays the rankings in the List box. It tallies the rankings using a special counting statement. Each ranking, a whole number between 1 and 5, becomes the subscript to the Tallies array. The ranking is used to determine which tally to add one to. Enter these lines:

```
For x = 1 To 30
   lstRankings.AddItem Str(x) & Chr(9) & Str(Ranking(x))
   Tallies(Ranking(x)) = Tallies(Ranking(x)) + 1
Next x
```

9 Set up the Picture box with a user-defined coordinate system for the graph. A sketch of the coordinate system used is shown in Figure 8-18.

Enter the line to set up the coordinate system:

```
pctGraph.Scale (0, 0)-(15, 5)
```

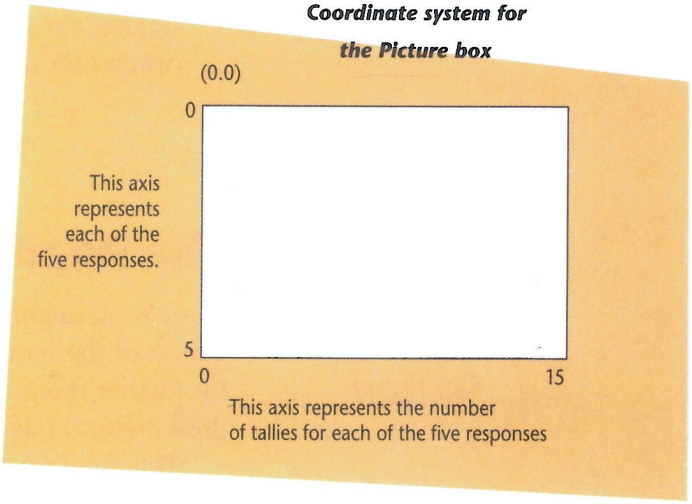


Figure 8-18
Coordinate system for the Picture box

10 The second loop uses the value of each ranking to draw a filled rectangle on the graph. The value is also displayed in the label control array:

```
Dim y As Integer
Const TopEdge = .7
Const BottomEdge = .3
For y = 1 To 5
   pctGraph.Line (0, y - TopEdge)-(Tallies(y), y - BottomEdge), , BF
   lblTally(y - 1) = Str$(Tallies(y)) & " — " & Str$(y)
Next y
```

The length of the bar is given by *Tallies(y)*. The top edge of the bar is *y - TopEdge*. The bottom edge of the bar is given by *y - BottomEdge*.

The actual tally along with the ranking number, 1 through 5, is assigned to the label control array. The control array starts with subscript 0. The value of *y* is decreased by 1 to give the subscript of the label control array, whose elements have indexes ranging from 0 to 4.

Try running the program, then save the project and form files.

QUESTIONS AND ACTIVITIES

1. Experiment with the values of the constants *TopEdge* and *BottomEdge*. Change the values by tenths and run the program with each change.
2. Alter the **Line** statement to change the color of the bars. Look up where to put the color in the Help system.
3. Alter the **Line** statement to draw empty rectangles, not filled rectangles.
4. Use the Notepad to create a file with 30 random entries between 1 and 5. Save the program in the Temp directory with the name **C:\Temp\Response.txt**. Put one value per line. Replace the 30 assignment statements in the Form_Load procedure with code to open the file for input, read the 30 values from the file, and initialize the array.
5. Change the coordinate system of the picture box and the **Line** commands to draw a vertical, not horizontal, graph.

Section

The Population Biology Program

There is an underlying order to nature and the events of nature. Observation of the natural world reveals relationships and patterns that beg for further research and explanation. One way to explain and to study these natural relationships is to create mathematical models.

The program you will build in this section is designed to model the fluctuations in the population of fish in a lake. The application of mathematics to the study of biological populations is called population biology.

Biological systems are enormously complex. The interaction of every part of a system on every other part is very difficult to model. Our ability to do such modeling has been enhanced enormously by using computers. As you will see with this program, the application of simple mathematical models to complex biological systems can sometimes have surprising results.

Background

In order to find a pattern in the fish populations of Butler Lake, you will need a simple mathematical model. This background section introduces both linear and nonlinear models. As you will see in this section, a nonlinear model fits your purposes better than a linear model.

LINEAR MODELS

Many events in life seem to conform to a linear model. A linear model assumes that a change in one event causes a proportional change in an associated event. For instance, if a 10-pound turkey should be cooked for 170 minutes, then it is logical that you would need to cook a 20-pound turkey for (2 x 170) or 340 minutes. See Figure 8-19.

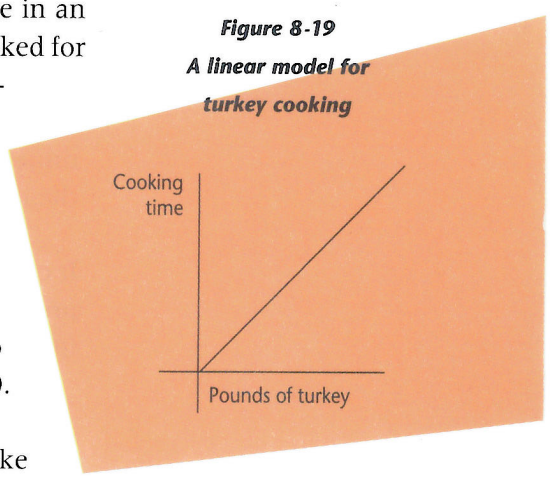


Figure 8-19
A linear model for turkey cooking

If an investment of $100 earns $5, you might think that an investment of $200 will earn $10. Should studying for one hour a night lead to a grade increase of 10%, if studying for half that time leads to a grade increase of 5%?

Actually, beyond a certain ratio of study time to class time, studies have shown that grades *don't* improve. And there is no guarantee that investing $200 will earn $20 dollars or even $10. These events do not conform to a linear model.

Now, try to figure out whether the populations of fish in a lake would conform to a linear model. How many fish there are depends on how much food is available. If the fish population reaches a stable level, neither increasing or decreasing significantly, an increase in food ought to lead to an increase in the population. More food means more fish. Does it?

To test this question, start with a simple linear model:

$p_2 = r * p_1$

Instead of using actual figures for populations, which would make the model specific to a particular pond, in this model p_1 is the level of fish population at time 1 expressed as a number between 0 and 1. p_2 is the level of fish population at time 2. The growth factor that transforms one population number into the next is represented as r.

By this equation, the population grows each year if the growth factor is greater than 1. If the growth factor is less than 1, the population decreases each year.

Growth factors are based on past data and predictions about the future. If the population two years ago was 1000 and the population last year was 1500, the growth factor is 1500/1000 or 1.5. It might be reasonable to assume a population of 1500*1.5 or 2250 for this year.

Often, you compare populations from one year to the next because many animals in the wild have young once a year. You might not use years if the population you are examining reproduces every few days or months. Predicting the growth of a population entails finding a good value for the growth factor. If conditions for growth are right, pick a high value (1 or more) for the growth factor. If food is scarce and predators are many, pick a low value (less than 1).

APPLYING A LINEAR EQUATION

To experiment with this linear equation for the fish population of a lake, start with a population of 0.5 and a growth factor, *r*, of 1.1. These numbers represent the current fish population and the conditions for growth. To find the population for the next year, p_2, multiply 1.1 times 0.5. The result, 0.55, is the prediction for next year's population. To continue the predictions, use 0.55 as p_1 and multiply again by 1.1 to predict the next year's population.

An equation applied in this manner is called a difference equation. It calculates the difference between one year and the next. The value it predicts becomes the input used to predict the value of the population for the year after that, and so on, as shown in Table 8-1.

Table 8-1 Fish Population Year to Year

year 1	p_1	p_2
2	0.5	0.55
3	0.55	0.605
4	0.605	0.6655
5	0.6655	0.73205
6	0.73205	0.805255
7	0.805255	0.8857805
8	0.8857805	0.97435855
9	0.97435855	1.071794405
10	1.071794405	1.1789738455

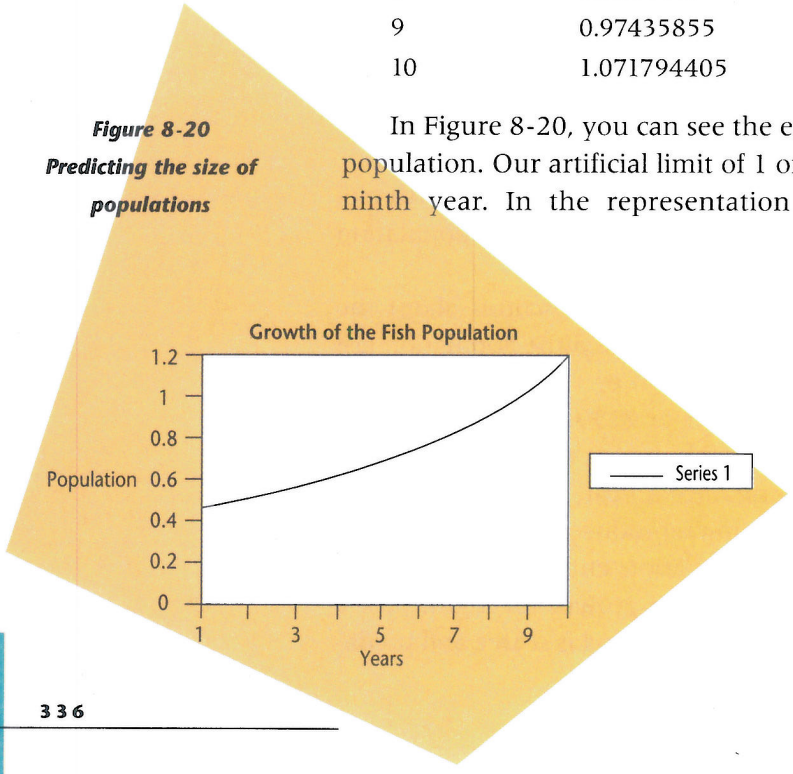


Figure 8-20
Predicting the size of populations

In Figure 8-20, you can see the effect of unrestrained growth on the population. Our artificial limit of 1 on the population is exceeded in the ninth year. In the representation of population figures as values between 0 and 1, 1 represents the theoretical maximum population. This allows you to represent the extremes of population figures without actual numbers.

A very similar process goes on when school districts try to predict enrollments. In a growing community with reasonably priced housing, a school administrator would pick a high value for the growth factor, *r*. If the community is landlocked, prices

are high, and the population is aging, the administrator would pick a low growth value.

The only problem with this simple model is that it doesn't work. It might work for a year or two, but almost invariably after a few years the model breaks down. One way to explain the breakdown is to say there were factors that weren't taken into account, or new factors that couldn't have been anticipated. But maybe there is a need to modify the model.

THE NONLINEAR MODIFICATION

Even with virtually unlimited resources, the fish population in Butler Lake cannot grow without limit. There is no straight-line relationship that can explain how many fish will be in the lake five years from now. Eventually, the number of fish is just too large for the lake. When the food runs out, the fish start to die.

You need to add a nonlinear modification to your linear model, then see if that works for the fish in the lake. A nonlinear system doesn't assume a straight-line relationship between events. Try changing the linear equation like this:

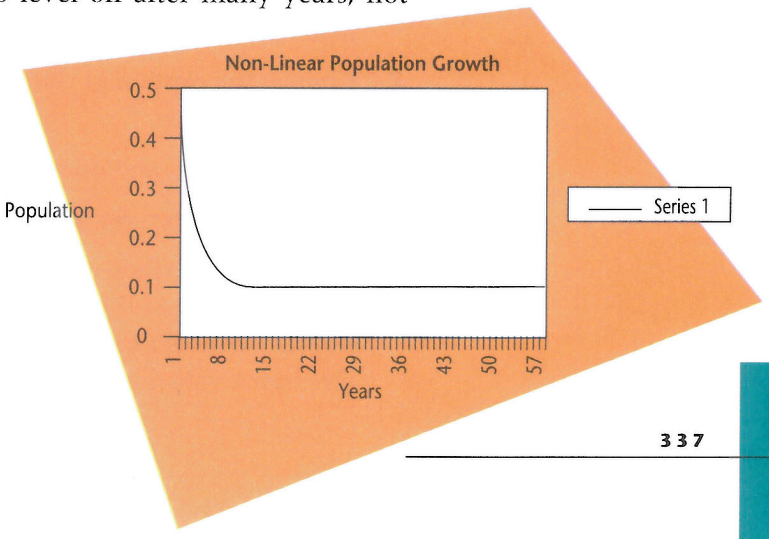
$p_2 = r * p_1 * (1 - p_1)$

Remember, the population is expressed as a number between 0 and 1. The added factor, $(1 - p_1)$, is large, near 1, when the value of p_1 is small. Because it is nearly 1, it affects the product on the right side of the equation very little. As p_1 gets larger (as the fish population grows), the factor $1 - p_1$ starts to shrink. As it shrinks, it affects the growth rate. It acts as an inhibiting factor to unrestrained growth. As p_1 approaches 1, the factor $1 - p_1$ approaches 0. The resulting population in the subsequent year, p_2, is quite small.

If you use a growth value of 1.1 with the nonlinear equation, you will find that the population seems to level off after many years, not growing or shrinking by very much. The size of the population when it levels off is called an equilibrium point. In this case, the equilibrium is 0.090909. See Figure 8-21. Scientists have observed such equilibrium points in natural populations.

What's interesting about this model is what happens when the growth factor, r, is increased. For many values of r, the population

Figure 8-21
Equilibrium points for fish populations



reaches an equilibrium point. As *r* increases, the equilibrium point increases. At a certain point, when the value of *r* increases, a single equilibrium point is not reached. The population fluctuates year after year, first up, then down. If you continue to increase *r*, strange things start to happen. The population predictions don't reach equilibrium and they don't bounce back and forth between two or even four values; they become chaotic. The population for each successive year is unpredictable.

When chaotic values were first noticed, scientists assumed that the model had been strained beyond its limits and the results were rejected. Now, a new way of explaining these results, called Chaos theory, explains why the results happen. The next section discusses Chaos theory in more detail.

Starting Out

This program illustrates what happens when a nonlinear model is used to simulate the population growth of the fish in Butler Lake. The program uses a menu to pick values for *r* and a graph to plot the growth of the population.

The graph is drawn directly on the form. This eliminates the need to specify an object for the graphics commands. The graphics commands used in this program are **PSet**, **Cls**, and **Scale**.

The program declares one global variable, *r*, in the general declarations section of the program.

Creating a Menu

Build a menu that allows the user to select a value of the growth factor, *r*, from a list. The values form a menu control array. Follow these steps:

1. Start Visual Basic. If Visual Basic is running, select New Project from the File menu.
2. Change the caption of the default form to **The Fish Population in Butler Lake**.
3. Select the form and open the Menu Design window.
4. Enter the caption **&r Value**. Enter the name **mnuRValue**.
5. Click on Next. Click on the right arrow to create a submenu. For each entry in the submenu, assign the same name e.g., mnuGrowthFactor. This creates a menu control array. Give successive index values to each entry. Click on Next after each entry.

Growth Factor	*Index*
1.1	0
2.0	1
2.5	2
3.0	3
3.2	4
3.4	5
3.6	6
3.8	7
&You Choose	8

6 Click on Next. Click on the left arrow to return to the level of the entries in the main menu bar. Enter the caption **&Graph!** and the name **mnuGraph**. Disable the item by clicking on the Enabled property.

7 Click on Next. Enter the caption **E&xit!** and the name **mnuExit**. See the complete menu design in Figure 8-22.

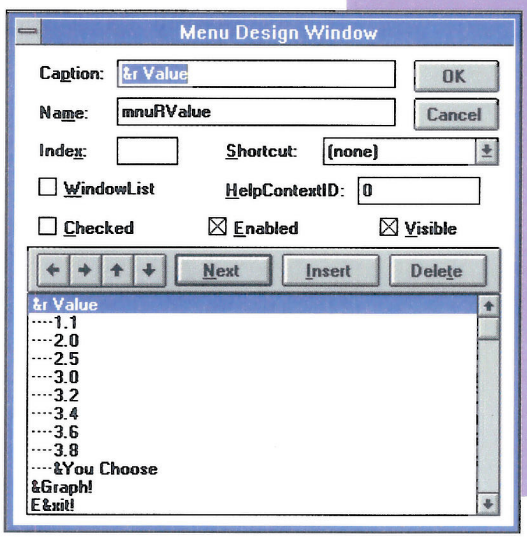


Figure 8-22
Completed menu

Adding an Object

The form has a simple appearance, with just the menu bar displayed (see Figure 8-23):

To finish the form:

1 Draw a label in the center of the bottom of the form.

2 Set the FontSize to 9.75, BorderStyle to 1-Fixed Single. Set the Alignment property to Center. Delete the caption. Name the form lblGrowthFactor.

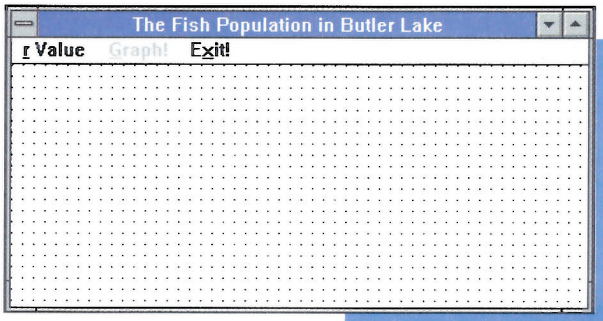


Figure 8-23
Form with menu bar displayed

Entering the Growth Factor Code

The code for mnuGrowthFactor handles assigning a value to the global variable *r*, the growth factor. To complete this code:

1 In the Project window, click on View Code. Open the general declarations section and enter the following declaration:

```
Dim r As Single
```

2 Open the procedure for mnuGrowthFactor and enter the line declaring a string variable for a message:

```
Dim Msg As String
```

3 Enter the **Select Case** statement that handles the assignment of values to *r:*

```
Select Case index
   Case 0
      r = 1.1
   Case 1
      r = 2.0
   Case 2
      r = 2.5
   Case 3
      r = 3.0
   Case 4
      r = 3.2
   Case 5
      r = 3.4
   Case 6
      r = 3.6
   Case 7
      r = 3.8
   Case 8
      Msg = "Choose a value for r between 1 and 4"
      r = Val(InputBox(Msg))
End Select
'—Check the value
mnuGrowthFactor(index).Checked = True
'—Enable the graph menu item
mnuGraph.Enabled = True
```

Entering the Graph Code

The form itself will display the graph. Follow these steps to enter the code:

1 Click on View Code in the Project window. Open the mnuGraph_Click() procedure.

2 Enter the declarations for the local values:

```
Dim yr As Integer, p1 As Single, p2 As Single
```

3 Clear the form and set the coordinate system. The population values range from 0 to 1. The graph will plot 50 years of population predictions.

```
'—Clear the form
Cls
'—Set the coordinate system
Scale (0, 1)-(50, 0)
```

4 Set the DrawWidth:

```
'—Set the DrawWidth for a thicker point
DrawWidth = 2
```

5 Set the initial population. Set the label to show the current value of *r*.

```
'—Initial population
p1 = .5
'—Set caption
lblGrowthFactor.Caption = "r is " & Str$(r)
```

6 Enter the lines to set up a loop that generate population predictions for 50 years. Generate the new population, plot the point, and update the population variables.

```
'—Fifty years of predictions
For yr = 1 To 50
   p2 = r * p1 * (1 - p1) ' new population, p2
   PSet (yr, p2)
   p1 = p2                ' reset population
Next yr
```

7 Enter the command **End** in the mnuExit procedure.

8 Save the project and form files

9 Run the program. Click values for *r*. Click on Graph!

10 Experiment with values for *r* entered through the Inputbox. Values near 4 give interesting results. See Figure 8-24.

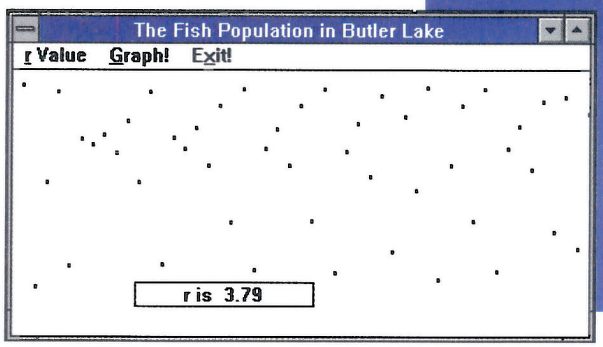


Figure 8-24
Results from the Population Biology program

What's Next

Looking at each graph individually is tedious. Imagine the difficulty of producing several graphs without the help of the computer. The computer makes this kind of modeling very easy.

It may have occurred to you to build a program to summarize the data. Imagine a graph that holds data for hundreds of values of r from 1 to 4!

QUESTIONS AND ACTIVITIES

1. What factors would be considered when determining a reasonable growth factor for a wolf population in Yellowstone?
2. During the 1970s, the price of gold was on an upward spiral. One month the price was $200 per ounce, the next it was $250. If this were a linear relationship, what growth factor would you calculate from the data given?
3. Change the initial value of p_1 to various values between 0 and 1. Run the program and note any changes.
4. Alter the program to prompt the user to enter a value for the starting population when the program is run.
5. Modify the graph routine to connect the dots with a **Line** method. Change DrawWidth to 1.

Section

Chaos Theory

SCIENTIFIC THEORY

Scientific theories seek to explain and make predictions about events. If a theory is good, its predictions, confirmed by experiments, come true. Over a period of time, theories either evolve or are replaced by new theories. What happens is that, one by one, anomalies in a theory pop up. An anomaly is an unpredicted and, therefore, unexpected event. The anomalies grow in number until they demand explanation. At that point, a whole scientific theory may be scrapped and replaced with another.

The new theory has its own language and its own assumptions about truth. It makes predictions. Although the words used in the new theory may seem familiar, the meanings change so that old words are used in new ways.

In the early 1960s, a mathematician-turned-meteorologist named Edward Lorenz put together a weather simulation on his computer. The machine itself would be unrecognizable today. It had very few of the external parts you think of as belonging to a computer. It did have a processor and memory and could be programmed and display its results. Lorenz used it to put together a weather simulation that did a good job of mimicking real weather patterns.

Lorenz's computer began with a set of initial conditions, which are the conditions that are present at the start of the simulation. From there, he calculated a sequence of weather patterns. One day, he tried to recreate a sequence of weather patterns from the middle of the sequence. He took values from the middle of a run, then entered those values into his program by hand and ran the program again. He expected the second run to match the first. To his astonishment, after a short while, the runs differed dramatically. Soon there was no way to tell the two runs were related at all.

Lorenz entered a value to initialize the system accurate to three significant figures, thinking that weather patterns change in broad strokes, impervious to small changes. That particular number in the original run was calculated to an accuracy of six significant figures. The number Lorenz entered was different by 1 part in 1,000, yet the sequence of weather patterns produced were, after a short while, completely different.

It was then that Lorenz realized that long-range weather prediction would be impossible. The physical systems that govern weather are incredibly sensitive to initial conditions. We cannot know the initial conditions of a physical system well enough to make accurate long-range predictions. This odd result in the weather program was the beginning of a new science. This science finds order behind chaos and is named Chaos theory.

You may have experienced the order behind chaos yourself in your own kitchen. Liquids flow smoothly and predictably until, suddenly, the flow becomes turbulent, like the rapids of a river. Although plumbing systems are designed to prevent this, you may have heard your pipes knocking when water is flowing. Sometimes you can adjust the flow from a kitchen faucet until it starts to knock.

The knocking occurs because even in the midst of chaos (turbulence), order can emerge. The turbulence of the water resolves into a rhythm that becomes knocking.

The water flowing through the pipes of your house, the wind flowing over the wing of an airplane, electrons flowing through conductors

and semiconductors—all of these things affect our everyday lives. Understanding the flow of these liquids is necessary for designing systems to handle them.

In the next section, you will be combining the individual graphs of the Population Biology project into a single graph. In that graph, you will see order dissolve to chaos. Order then reemerges, just before chaos takes over again.

QUESTIONS AND ACTIVITIES

1. How do anomalies figure into the development of scientific theories?
2. Give an example of a simple system in which, if you know the initial conditions of the system and the rules that govern the system, you can predict what will happen over a period of time. For instance, a pool table may contain such a system.
3. How did an anomaly occur in Lorenz's experiments?
4. Why is accurate long-range prediction of weather impossible?
5. What does your kitchen faucet have to do with Chaos theory? Try to adjust the water flow in your faucet at home to cause knocking.
6. The swirling of a drop of antifreeze in water is an example of a chaotic system. Write down two examples, not listed in the text, of systems that might be chaotic.

Section

The Population Biology Program Revisited

The first population biology program predicted the equilibrium state(s) (if they existed) for populations of fish in Butler Lake. Equilibrium states are population values that keep recurring. If there is one equilibrium point, the same population value recurs each year. In other words, the size of the population is stable.

Intuitively, you would think the fish population ought to be stable. The number of fish that die from old age or lack of resources (like food) should equal the number of fish born. If the conditions are very good in Butler Lake, the equilibrium population should be high. If the conditions are bad, the equilibrium population should be low.

The results from the program, though, did not match that hypothesis. When you tried different values for the growth factor *r*, you didn't always find a single equilibrium point. Instead, the graph sometimes showed two equilibrium points. In other words, your data identified two different equilibrium values, and these values interchanged each year. One year, the population would reach *p1;* the next year, the value was *p2;* the third year, you were back to *p1*. See Figure 8-25.

Scientists have found the same results in fish population studies. Does this mean our model is wrong? The nonlinear model you have used so far showed that for certain values of *r*, the growth factor, the population values went wild. See Figure 8-26. Should you throw your results out because they don't square with your intuition? In this second time through the fish population program, you will answer that question.

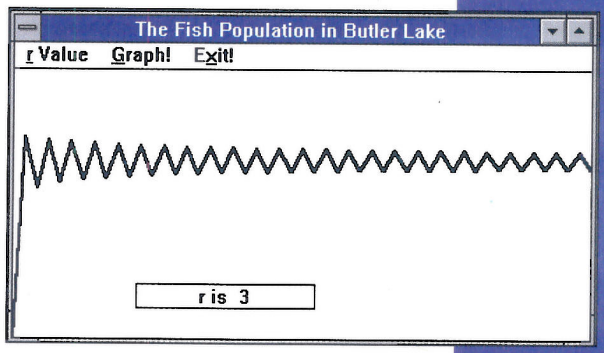


Figure 8-25
Fluctuations in population size in Butler lake

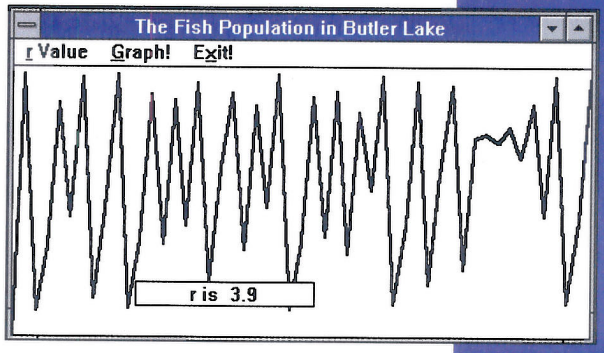


Figure 8-26
Predictions of population size in the lake with a different growth factor than used in Figure 8-25

MISTAKEN PREDICTIONS... OR ARE THEY?

Maybe our model's prediction of unpredictability isn't a mistake or misinterpretation. Maybe the nonlinear model that seems to predict chaos is correct.

This is a significant result. It means that fish populations may be unpredictable. If fish populations are unpredictable, what about school enrollments?

First Program Revisited

Remember, valid values for the population figure are decimal numbers between 0 and 1. Values for the growth factor *r* run between 1 and 4. A value for *r* less than 1 results in a population of 0. When the value of *r* is greater than 4, the population exceeds the upper limit of 1. Assume

the population starts at 0.5 and generate each successive population figure from the current one:

```
p2 = r * p1 * ( 1 - p1 )
```

When you applied this equation to successive generations of fish for small values of *r*, eventually the fish population stopped oscillating and settled down to a single equilibrium value. When you pushed *r* a little higher, the graph showed two equilibrium points. Then at some point, the steadily increasing value for *r* caused the resulting graph to bounce all over the place. See Figures 8-26 and Figure 8-27.

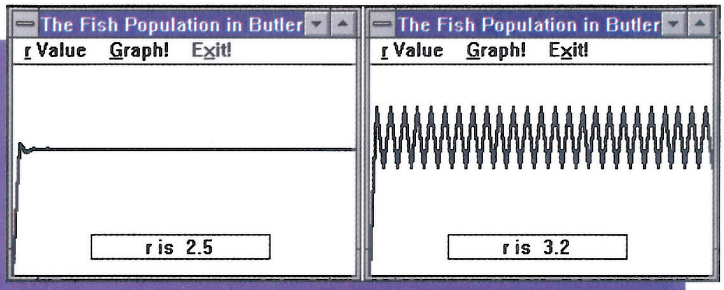


Figure 8-27
Predicting the size of the fish population

The value of the population variable in Figure 8-26 never settles down to a single value, or even two values.

Starting Out

To see patterns in your population data, you need a way to put all the information from the individual graphs you produced in the first population biology program into a single graph.

The Summary Program you are about to build displays this summary graph on the default form. You will use a loop to generate values for *r* that span the range from 2.5 to 4.

Each column of the graph represents one value of the growth factor, *r*. For each of these values, the population figures for 150 generations of fish are calculated. You use the **Pset** method to place these population values on the graph in the column for that growth value.

For values of *r* that produce one equilibrium point, almost all the 150 points representing 150 different population figures will be at about the same place. As the value of *r* increases, the single equilibrium point will split to two, then to four. Each successive column plots the data for a single value of the growth factor. After several doubling events (a single point, two points, four points), the graph bursts into 150 seemingly random positions all over the column.

The resulting graph shows the equilibrium points for many points in the range from 2.5 to 4.0. See Figure 8-28.

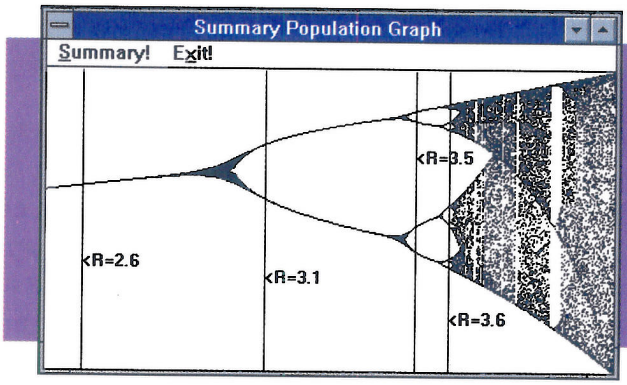


Figure 8-28
Summary graph of fish populations

Note that the first 20 generations for each value of *r* are not plotted. During the first 10 or so generations, the population figure moves around a lot before settling on its equilibrium point(s). The starting population value is 0.5.

Figure 8-28 can be divided into several regions. In the first region, the population value settles on a single equilibrium point. The column *r = 2.6* is typical of this region. The graph branches to two in the next region. This region shows two equilibrium points. The column *r = 3.06* is typical of the region. The column *r = 3.49* is typical of the region with four equilibrium points. *r = 3.56* is in a region where there are 8 equilibrium points. You can glimpse eight doubling to 16. But then, you find chaos. Finally, the pattern resolves again. From chaos emerges three values. These double and soon dissolve again into chaos.

Building the Menu

In Figure 8-28, you could see the menu bar above the graph. Follow these steps to create that menu.

1. Start Visual Basic. If Visual Basic is running, select New Project from the File menu.
2. Change the caption of the form to **Summary Population Graph**.
3. Select the form and open the Menu Design window.
4. Enter the caption **&Summary!**. Enter the name **mnuSummary**.
5. Click on Next. Enter the caption **E&xit!**. Enter the name **mnuExit**.
6. Save the project and form files.

Planning the Code

Before you start writing the code, you need to think through how to calculate the step, set the scale, and use the MouseDown event.

CALCULATING THE STEP

The values for the growth factor, *r*, are set by a loop. The loop steps up each time through just enough to fill the form with dots. If the step is too broad, there are blank vertical lines between each column. If the step is too small, the same column may be drawn more than once.

To make the step just right, you should first set the ScaleMode to 3. This sets the form to the pixel coordinate system. Save the value of the ScaleWidth property in the variable *WidthInPixels*.

This value is divided into the difference 4.0-2.5 to determine the step for the loop.

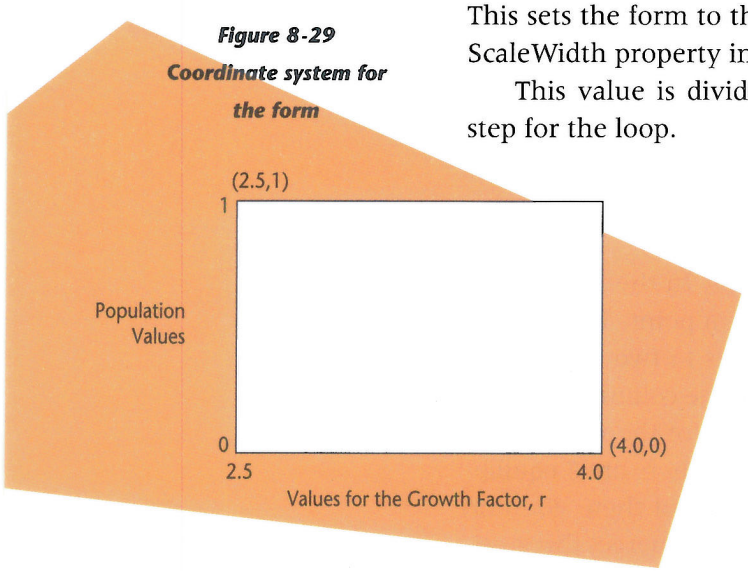


Figure 8-29
Coordinate system for the form

SETTING THE SCALE

The values for the growth factor range from 2.5 to 4.0. The population values range from 0 to 1. The coordinate system for the form is shown in Figure 8-29.

The **Scale (2.5,1)-(4,0)** command sets the coordinate system.

PLANNING THE MOUSEDOWN EVENT

You will use the form's MouseDown event to draw a labeled vertical line on the graph. When the graph is complete and a user clicks on the graph, the program draws a vertical line. Next to the line is the value of *r* for that column. This value comes from the *X* parameter to the MouseDown event.

The next place to plot a point or print a message is determined by the CurrentX and CurrentY properties of the MouseDown event. The message displaying the value of *r* is positioned at the mouse click by setting *CurrentX* to x, and *CurrentY* to y.

Entering the Code

Follow these steps to enter the code for the Summary Population Graph program.

1 In the mnuSummary_Click() procedure, enter these lines to declare the local variables.

```
'—Declare variables
Dim Stp As Single, r As Single
Dim Pop As Single, Generation As Integer
Dim WidthInPixels As Integer
Const StartGen = 20
Const StartPop = .5
Const TotalGen = 150
```

Stp is the step value for the loop. *r* is the growth factor. *Pop* is the current population value. *Generation* controls the loop that calculates the successive populations. *WidthInPixels* saves the width of the form in pixels. *StartGen* is a constant. The graph begins for each value of *r* at *StartGen + 1*. *StartPop* is the starting value for the population. *TotalGen* is the total number of generations calculated.

2 Enter the lines to find the width of the form in pixels and set the custom coordinate system.

```
'—Set the coordinate system
ScaleMode = 3
WidthInPixels = ScaleWidth
Scale (2.5, 1)-(4, 0)
```

3 Enter the lines to clear the form and calculate the step value:

```
'—Clear the form
Cls
'—Process step from menu
Stp = (4 - 2.5) / WidthInPixels
```

4 Enter the lines to set up the outer loop and initialize the value of *Pop*:

```
For r = 2.5 To 4 Step Stp
   Pop = StartPop
```

5 Enter the lines to set up the inner loop and calculate and display the population values:

```
For Generation = 1 To TotalGen
   Pop = r * Pop * (1 - Pop)
   If Generation > StartGen Then
      PSet (r, Pop)
   End If
Next Generation
Next r
```

6 Enter **End** in the procedure for mnuExit.

7 Save the project and form files.

8 Run the program. Click on **Summary!**

9 Enter these lines in the MouseDown event procedure of the form:

```
Line (x, 0)-(x, 1)
CurrentX = x
CurrentY = y
Print "<R=" & Format$(x, "#.0")
```

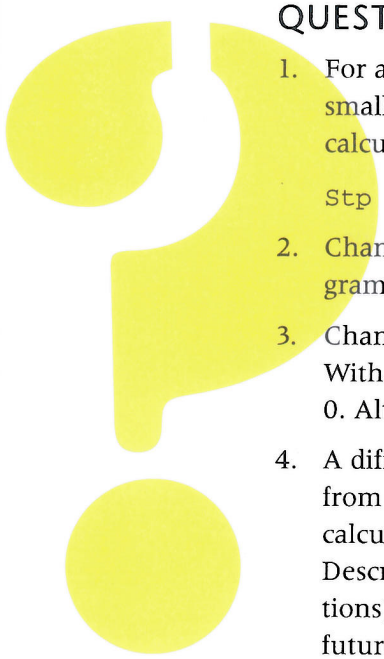
10 Run the program. Click on **Summary!**. When the graph is finished, click on key points of the graph to display the *r* value.

11 Save the project and form files.

The computer is a science laboratory. It is a place to run experiments and test predictions. The computational and graphic power of the computer make it an indispensable tool for doing science.

The Population Biology programs reveal patterns that can be examined and explored in minutes. By hand, such work would take years. Use problem 8 in the Problems section as a guide to perform real explorations of the information created by the Summary Population Graph program.

QUESTIONS AND ACTIVITIES



1. For a less dramatic graph, but faster execution, experiment with smaller values for *Stp*. Alter the program by changing the statement calculating *Stp*.

   ```
   Stp = 2 * (4 - 2.5) / WidthInPixels
   ```

2. Change the value of *TotalGen* to 50 and *StartGen* to 0. Run the program and observe the results.

3. Change the value of *StartPop* to 0.1, then to 0.9. Run the program. With *StartGen* = *20*, what effect does *StartPop* have? Set *StartGen* to 0. Alter the value of *StartPop*. What changes do you observe?

4. A difference equation is an equation used to calculate a prediction from a known value. In the fish problem, you used an equation to calculate next year's fish population from this year's population. Describe two other situations (don't worry about the actual equations) in which you could use a difference equation to predict future values.

5. In the fish population problem, some values for the growth factor *r* resulted in two equilibrium points. Describe in words why that

might happen. What conditions of the system would cause the succeeding year's population to decrease one year and increase the next?

We say the difference equation:

```
p2 = r * p1 * (1 - p1)
```

is a nonlinear equation. What makes this equation nonlinear? *(hint: multiply out the right side)*

A menu command that is checked shows a check mark at the beginning of the item. A menu item that is enabled can be clicked to generate a Click event. A menu command whose Enabled property is set to **False** is dimmed and does not generate a Click event when clicked.

Giving two or more menu items the same name creates a menu control array in which all items execute the same event procedures. The items are differentiated within the procedure by the value of the index parameter sent to the procedure.

The **Print** method is used for displaying text and values directly on a form.

The **Select Case** statement replaces a series of **If-Then** statements. Its syntax is:

Select Case *expression*
  Case *value1*
    statements if value1 = expression
  Case *value2*
    statements if value2 = expression
  ....
  Case Else
    statements executed if the expression doesn't match any of the Values
End Select

An array is a list of values in memory. Arrays have a single variable name that refers to the entire list. A subscript is a whole number that selects a single item from the list.

A non-linear difference equation of the form:

```
p2 = r * p1 * (1 - p1)
```

is used as a simple mathematical model for many systems. *p1* is the initial state, *p2* is the final state, and *r* is the growth factor.

1. Twenty Random Numbers

Use the **Rnd** function to generate an array full of 20 random numbers. Print the numbers on the default form.

2. The Length of a Name Problem

Use a string array to store 10 names. Use the **Len** function to search the array for the longest name and display the length of the longest name. Use a **For-Next** loop to cycle through the 10 names and a single **If-Then** statement to calculate the length of each string and compare the length with the value of an integer variable. Initialize the integer variable to 0. If the length of a name is larger than the value in the integer variable, replace the integer variable's value with the length of the longer string.

3. The Licensing Problem

Write a program using a menu to display information about various licensing fees. This table summarizes the information necessary.

Vehicle licenses		*Pet licenses*	
automobile	$25	neutered dog	$10
motorcycle	15	dog	$25
van	25	neutered cat	$12
commercial	100	cat	$30
airplane	250		

Use two menu control arrays with the items listed above as submenu items. The column headings should appear on the top menu bar. Inside the event procedures, use a **Select Case** statement to choose the correct fee. Once the fee has been determined, use a message box to display the fee.

4. The Portable Stereo Problem

Write a program to help diagnose a problem with a portable stereo. The program should allow the user to pick symptoms from a menu and display a message about what to do to solve the problem. Use the information below as a guide.

Stereo won't play	*Tape moves, but no sound*
Are batteries installed?	Do headphones work?
Are batteries charged up?	Are headphones on head?
Is tape jammed?	

5. The Linear Enrollment Problem

Use a linear difference equation ($p_2 = r * p_1$) to print a table of values based on the following information:

Year	*Enrollment*
1984	2150
1985	2175

Calculate the growth factor, *r*, from the information given above. Display a table that predicts the enrollment for the years from 1984 to the year 2000.

6. Linear Enrollment Revisited

Rewrite the program above but use the non-linear equation, $p_2 = r * p_1 * (1 - p_1/4000)$. Write the program so different values for *r*, the growth factor, can be entered.

7. Store Rebate

Write a program using a **Select Case** statement to solve the following problem. A store offers a rebate on a purchase based on the amount of the purchase.

Amount Spent	*Rebate*
0 - 100 dollars	2%
101 - 1000	3%
1001 - 5000	3.5%
5001 - 10000	4%
10001 and over	5%

Write a program to enter an amount spent, then calculate and display the rebate. Use an integer to represent the amount spent. The **Select Case** statement can designate a range of values with the following syntax:

Case *low* **To** *high*
    statements

8. More Population Biology

Modify the Summary Population Graph program to allow the user to enter values for the range of *r* values. Currently, the values 2.5 and 4 are coded directly into the program. Use the MouseDown event to define new values. Replace the old MouseDown code with this new code. The first click should be used as the starting value of *r* and the second click as the ending value. This allows the user to examine a specific range of values in detail. The automatic scaling done for the *Stp* variable comes in handy here.

```
Put #1, phoneText(x)
Open "c:\plist\phone.dat" For Random Access Write As #1 Len = 55
```