WEB 2

Using Visual Basic with Project 2002

In this chapter

WHY USE VISUAL BASIC MACROS?

There are several reasons you might want to use Visual Basic to create macros. This chapter covers some of the most common reasons and gives examples so that you can start creating your own.

Macros were originally used to collect a series of keystrokes that performed a given command or series of commands so that you could run the macro instead of typing the string of commands. This is very useful when the work you are doing is repetitive, such as setting a particular format or printing a particular report. This original functionality evolved and now, instead of just recording and replaying keystrokes, macros can do much more. Microsoft includes Visual Basic for Applications (VBA), a fully functional programming language, in Project as well as other Microsoft Office products. Having a programming language rather than a macro language allows you to program Project to do things that cannot be done by using simple macro functionality. It also allows you to have Project work with other Office applications to achieve the result you want. For example, using VBA, you can have Project send data to Excel and because Excel also uses VBA, you can tell Excel to chart the data in a specific way.

There are four basic ways to use VBA in Project:

- **Automation**—Many activities in Project require that you perform a number of separate actions in a distinct order. Printing a customized report is a good example of this. You need to select the report type, set the options the way you like, and print. This requires navigating through several menus and dialog boxes, making a variety of choices as you go. Because you typically want reports to be in the same format for each reporting period, this process can be automated so that you don't have to go through those steps individually each time you want to print a report. When the process is automated, the formatting of the printed reports is the same each time.
- **Configuration**—Configuration of Project is another good use for VBA. You can use VBA to configure the toolbars, views, and other customizable elements of Project so that you can always have it set up the way you like. You can also use VBA to create a common configuration that you can distribute to a group of users.
- **Interrelationships**—VBA enables Project to work with other software tools. VBA enables you to open and control other applications, and it enables Project to be controlled by other applications as well. Project and Excel are often used together, to take advantage of the graphing and analysis capabilities of Excel.
- **Extended functionality**—You can use VBA to extend the functionality of Project. Although Project is packed full of functions, there are some things it just does not do. With complete access to the Project object model, via VBA, you can add functionality to Project. Simple things such as automatically renaming a group of tasks and complex things such as tracing the dependencies to a selected task are all possible with the use of VBA.

This chapter covers a very simple example of recording a macro as an introduction to the concepts and definitions related to VBA, and then it moves on to the Project object model and some typical programming structures that you can adapt for whatever you need to develop.

GETTING STARTED WITH VISUAL BASIC MACROS

One of the first things that macros were designed for was recording a sequence of keystrokes so that they can easily and accurately be replayed. One of the most useful display options in Project is the ability to zoom the display of the Gantt chart so that it shows the entire project. The command to do this is fairly well hidden in the View menu, and it takes a few clicks to get there. This section uses that as an example and defines some terms that you will need to understand later in the chapter.

Before you create a macro, you need to walk through the needed so that you can eliminate any false or unnecessary ones. On a simple macro like the one described in this section, this might not be necessary, but for macros that have many steps, it is very important. When recording a macro, you must do things by selecting them. Using shortcuts such as the Control key will not be recorded properly.

Walking through the steps is easy: Select View, Zoom, Entire Project, and then click OK. The Gantt chart timescale will be reset so that the entire project is displayed.

To record a macro that zooms the display of the Gantt chart so it shows the entire project, follow these steps:

1. Select Tools, Macro, Record New Macro. This brings up a dialog box that gives you the opportunity to name the macro, give it a command key, give a description, and determine where Project should store it. For now, store it in the Global template file because you want it to be available whenever you are working in Project.
2. Give it the name ZoomAll and add any notes that you want. When that is done, click OK, and the recorder begins recording.
3. Select View, Zoom, Entire Project, and then click OK. When you are finished, click on the Stop button to stop the recorder.

NOTE

While the recorder is recording, a Stop button pops up on the screen. Click it to stop the recording when you are done. If the button is not visible for some reason, select Tools, Macros, Stop Recording.

4. It is a good idea to test the macro. For a simple macro like this, the easiest way to test it is to run it. To do so, open a different file, set the timescale so that not all of the project is showing, and then run the macro by selecting Tools, Macros, Run Macro and then selecting the macro. If it is working properly, the macro should zoom with no problem. Always save your work before testing. If your file has been saved, you can always go back if your macro does something unexpected or destructive.

5. If the macro does not pass the test and does not work correctly, you do not have to delete it and start over. The next section covers how to modify a macro.

TIP

When you are satisfied with the macro, you can add a button to the toolbar for the macro so that you can simply click the button to run the macro.

CREATING A MACRO THAT WORKS IN DIFFERENT SITUATIONS

The macro you recorded in the preceding section will work in every situation because the objects it operates on are not specific to any one project. Zooming for an entire project will work for any project. With a simple change in the options, however, you can create a macro that can be recorded with no problems but does not work in all cases. This section walks through a similar example and shows how to correct the problems that occur when you record a specific macro.

This time, rather than using the Zoom, Entire Project command, you will use Zoom, Selected Tasks. This might be a more useful option when you want to look at a set of particular tasks you are working on and the duration of those tasks is short compared to the entire project.

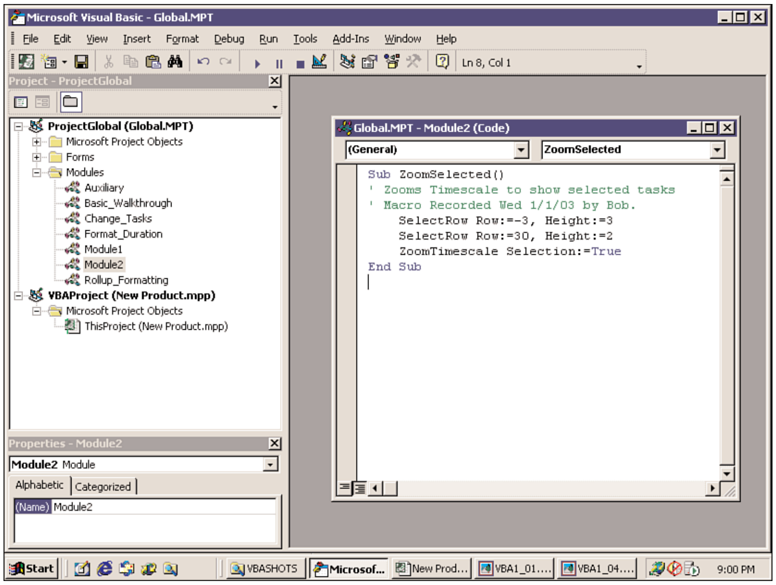
The process is the same as recording the previous macro:

1. Start the macro recorder and name the macro `ZoomSelected`. Store it in the Global template as before.
2. This time, there is an additional step. Because the command acts on a selection, you must first select the tasks you are going to zoom. Select a few tasks by dragging your cursor across them so that they are highlighted, and then select View, Zoom, Selected Tasks, and then click OK.
3. Stop the recorder.

To test the new macro, reset the timescale so that the whole project is showing, and then select a few tasks (select different ones from the ones you selected while recording the macro) and run the new macro. Project zooms, but it zooms to show the tasks you selected when recording the macro and ignores the tasks you have just selected. Because this macro is expected to zoom the timescale for the selected tasks and it ignores your current selection, it is clearly not working correctly.

The problem with this macro is what the recorder is recording. To see what is recorded, open the Visual Basic Editor (VBE). To open the VBE, press Alt+F11 or select Tools, Macros, Visual Basic Editor.

When you open the VBE, you see something similar to Figure 1. In the upper-left corner is the Project Explorer. Because you saved your macro in the `GLOBAL.MPT` file, you need to expand that file in Project Explorer and look in the `Modules` folder. Project creates a module for each macro you record. The highest-numbered module is the last one you recorded. If this is your first time recording macros, you should see `Module1`, which contains the first version, and `Module2`, which contains the second version. Double-click the highest-numbered module (it is likely to be `Module2`) to open it to display the code window.



```
Sub ZoomSelected()
' Zooms Timescale to show selected tasks
' Macro Recorded Wed 1/1/03 by Bob.
    SelectRow Row:=-3, Height:=3
    SelectRow Row:=30, Height:=2
    ZoomTimescale Selection:=True
End Sub
```

Figure 1
The default view of the Visual Basic editing screen is divided into three main areas–the Project window and Properties window on the left and the Code window on the right.

TIP

Project does not do a good job of naming modules for you. It supplies names such as `Module1` and `Module5`. If you have some macros you want to use or work on later, you should change the name of the module to something more appropriate so that you can easily find it. You can't change the name in the Project Explorer window, but you can change it in the Properties window. It opens by default just under the Project Explorer window, but if you don't see it, you can display it by pressing F4 or selecting it in the View menu. To rename the module, just type over the existing name.

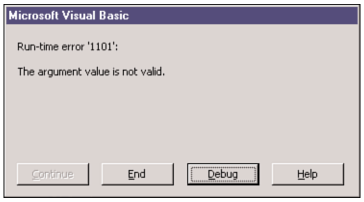
When a module opens, you see what was recorded by the recorder. A number of separate actions are grouped together as a macro or, more precisely, a procedure. Each line in that procedure is a comment, a single statement, or both. Every macro you record begins with a line stating that it is a *subroutine* (that is, a type of procedure), followed by the name of the macro and any parameters that the macro operates on. In the case of the `ZoomSelected` macro, you see the following code:

```
Sub ZoomSelected()
' This macro will zoom the timescale to show all the selected tasks
    SelectRow Row:=1
    ZoomTimescale Selection:=True
End Sub
```

The first line contains the name you gave the macro. Immediately after this line is the description that you entered or that Project entered for you. The description is in the form of a comment, so it begins with an apostrophe (`'`). Everything following on that line is interpreted as a comment. You can create as many comments as you like, and a comment can be on the same line as another statement, but it must follow the statement. Following this are the actions to occur and at the end, a line states `end sub`, which indicates the end of the macro. You can see from this code that you selected a task and then performed an action on the timescale based on that selected task.

The problem here is that you made the selection of the task part of the macro, so whenever the macro is run, that task is again selected. Although you could go back and rerecord the macro, the VBE enables you to simply delete the parts of the macro you don't want. In this case, you need to delete the line that starts with `SelectRow`.

After you make changes in the VBE, switch back to your project, select a few tasks, and run the macro again. At this point it should work fine. Try not selecting any tasks, and then run the macro again. Project generates an error (see Figure 2). In this case, click End. Debugging is covered later in this chapter.



```
Run-time error '1101':

The argument value is not valid.
```

Figure 2
Error windows present you with some detail about what the error is and allow you to end, debug, or go to help.

The problem is that you are asking Project to perform an action, but you haven't selected anything that it can use to perform that action.

The Object Model

Visual Basic works with a set of objects. *Objects* are the elements of the application such as tasks, resources, views, or specific cells. Groups of objects are called *collections.* A set of selected tasks is a *collection* of tasks, but a collection does not need to be made up of similar items. The `application` object is a collection of tasks, resources, assignments, and other objects.

When you are writing statements, you must first specify the object you are going to do something to or with, and then you state what you want to do with it. Most commonly you will want to add or delete an object or change something about it. The various aspects of the object are called *properties*.

Whereas a property of a car would be the color or the weight, a property of a task would be the duration, the cost, the task ID, or another value that could be associated with a particular task. In most cases you can change the properties of an object, just as you can change the duration of a task in Project. However, in some cases, you cannot change the properties of an object because they are the result of a calculation that Project has done. Total Slack is an example of a read-only property. You cannot change it by setting it to a specific value because the value of Total Slack is the result of a calculation by the scheduling engine. (In general, if you can directly edit the value of a field in one of the project tables, then you can set that same value by using VBA.

Objects also have *methods.* If we use the car analogy again, starting a car would be a method, and so would accelerating. Using a method with an object generally has an effect on one or more properties. For example, accelerating a car would increase the speed, which is a property. Some important methods handle the creation and destruction of the objects themselves. You add a new task to a project by using the `add` method of the `Project` object.

Knowing the object model is the key to becoming proficient at programming in VBA.

Fixing a Macro

In your code, the method `ZoomTimescale` requires some object to act upon:

```
ZoomTimescale Selection:=True
```

If nothing is selected, Project does not know how to set the timescale.

One way to fix this is to make the macro a bit more intelligent than it currently is. Using the car example again, you can see that if the car could be made to test the existence of the road ahead before it accelerated, it could prevent itself from driving off a cliff. You can do a similar thing in code by using a basic test. The object that you want to test is `Selection`. Here is the code you use to test it:

```
Sub ZoomSelected()
If Not ActiveSelection.Tasks(1) Is Nothing Then
    ZoomTimescale Selection:=True
End If
End Sub
```

The line `If Not ActiveSelection.Tasks(1) Is Nothing Then` tests to see if there is a valid selection. If there is a selection, the `ZoomTimescale()` method is executed. If there is not a selection, you do nothing.

Fine-tuning the Macro and Giving User Feedback with Message Boxes

If a user runs the macro when nothing is selected, nothing happens and the user doesn't know why nothing happened. You can improve that by telling the user what happened. The easiest way to do this is to pop up a message box that tells what happened (see Figure 3). To do this, you simply add two lines of code, as shown here:

```
Sub ZoomSelected()
If Not ActiveSelection.Tasks(1) Is Nothing Then
    ZoomTimescale Selection:=True
'add the following two lines
Else
    MsgBox ("No Tasks Selected")
End If
End Sub
```

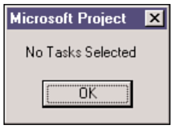


Figure 3
The message box displays whatever text you have specified within brackets.

Another way to make your code friendlier is to anticipate what choice the user would make if what he or she is trying to do is unsuccessful. In this case, you can simply zoom to show the entire project. Be careful about what you assume. It is wiser to do nothing and tell the user that than to do the wrong thing. The following code illustrates the change:

```
Sub ZoomSelected()
If Not ActiveSelection.Tasks(1) Is Nothing Then
    ZoomTimescale Selection:=True
'add the following two lines
Else
    MsgBox ("No Tasks Selected")
    ZoomTimescale Entire:=True End If
End Sub
```

TIP

It is best to record a macro when the objects and actions you are using will be the same every time. The following are some activities that fall into this category:

- Printing reports
- Switching views (while using Project's built-in views)
- Inserting a predefined task or tasks

However, you can also record macros in order to figure out how to write code to do other things you want to do. You can use the code that the macro recorder develops as building blocks to construct more complicated macros.

THE VBE

You learned earlier in this chapter how you can use the macro recorder to generate Visual Basic code, but for more complicated macros, you will want to start in the VBE. You saw earlier how to open the VBE (by using Alt+F11) and use it to modify a macro. The following sections describe at some of features the VBE offers and how to work with them.

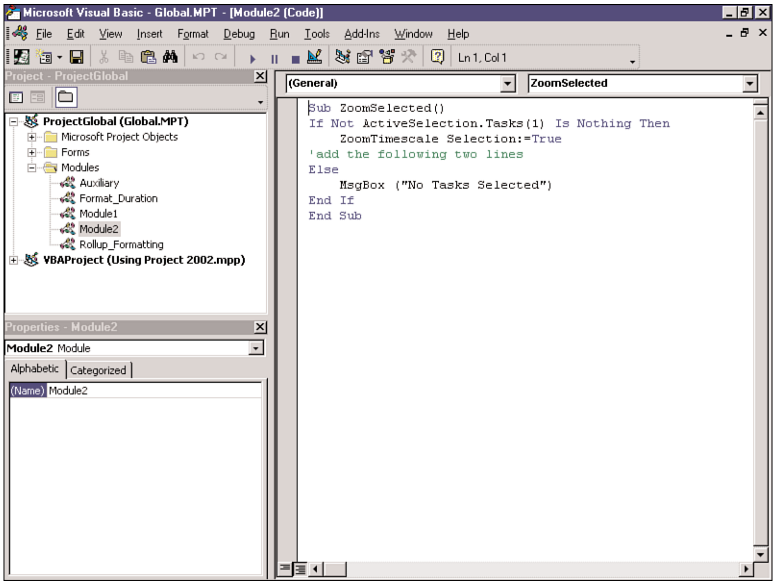
THE PROJECT EXPLORER

You saw the Project Explorer window earlier, when you were looking for the module where the macro you created was stored. The Project Explore is typically docked in the upper-left corner of the screen. The top project is usually `GLOBAL.MPT`, and this is a good place to put macros you are working on or that you want to use across several projects.

TIP

It is good practice to back up your Global template file occasionally, especially if you have put a large amount of work or customisation into it. You can back it up by simply copying it and storing it in a safe location.

Each project that is open in Project is shown in the Project Explorer window. You can copy modules between projects by dragging them from one project to the other. This serves the same purpose as using the Organizer within Project, but it does not require you to leave the VBE (see Figure 4).


Microsoft Visual Basic - Global.MPT - [Module2 (Code)]
File Edit View Insert Format Debug Run Tools Add-Ins Window Help
Ln 1, Col 1
Project - ProjectGlobal
ProjectGlobal (Global.MPT)
Microsoft Project Objects
Forms
Modules
Auxiliary
Format_Duration
Module1
Module2
Rollup_Formatting
VBAProject (Using Project 2002.mpp)
Properties - Module2
Module2 Module
Alphabetic
Categorized
(Name) Module2
(General)
ZoomSelected

```
Sub ZoomSelected()
If Not ActiveSelection.Tasks(1) Is Nothing Then
    ZoomTimescale Selection:=True
'add the following two lines
Else
    MsgBox ("No Tasks Selected")
End If
End Sub
```

Figure 4
When you are developing code, it can be helpful to store versions of the code in individual files rather than have them all exist in the Global template.

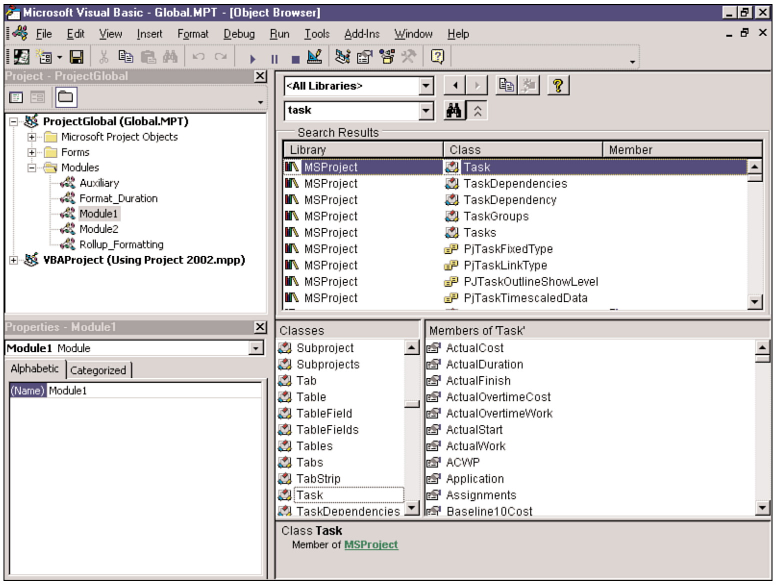
The Properties Window

The Properties window is positioned below the Project Explorer window, and it displays the properties for the item that is currently selected. You can edit properties in this window, and you can also use it when you want to name a specific module.

The Object Browser

The object browser is another essential window in the VBE. It displays all the objects that are available to you, including those from Office as seen in Figure 5. You can use it to browse through objects or search for them.

Figure 5
The Properties window is on the lower left, and the object browser is on the right.



Working with Tasks

Tasks are at the heart of Project, and you often need to make changes across a large number of tasks. Using a macro to do this is far more efficient than making all the individual changes by hand. Because of how common and powerful such macros can be, understanding how to cycle through all the tasks in a project and perform or not perform an operation on them is an essential skill.

For example, imagine a project in which several components are being developed. The workflow for each component is the same, so you would like to create just one schedule and then cut and paste for the other components. This works well when you keep the original outline structure, but you might sort, filter, or otherwise manipulate the tasks so that the original context is lost. To solve this, you can create a macro that joins the component name (which is in the summary task) to the task name so that the task name is a complete description.

Before you start coding, you need to think about the steps involved. You want to concatenate the name of each task's parent task with the task's own name and then store it in a text field (for example, `text20`). However, you don't want to do this on tasks that are summary tasks or external tasks (that is, those that are linked to other projects). More importantly, you don't want to do it on blank lines because the macro would fail if it tried to operate on a task that isn't there.

When you finish, you want to display a message stating that the macro has finished and how many changes were made. Giving feedback like this helps you and other users figure out what is happening and whether things worked as expected.

To create the `TaskSummaryName` macro, follow these steps:

1. Open the VBE by pressing Alt+F11.
2. Select the location for the module; in this case, use the `GLOBAL.MPT` file because you are just developing this and want to keep it available for all your projects, including test projects you might want to create.
3. Select the project and then select the Insert menu and insert a module.
4. Go to the Properties window and edit the name so that it is something you will remember. This example uses the name `TaskSummaryName`.
5. Double-click on the new module to open the code window for it.

TIP

As you learned earlier, a *module* is a container for one or more macros. Because a module can be a collection of different macros, it is good practice to use modules to organize the different macros you have created or are working on.

6. In the code window, type the following code:

```
Option Explicit
Sub TaskSummaryName()

'Name the variables
Dim T As Task
Dim Ts As Tasks
Dim intTcounter As Integer

'Set the variables
Set Ts = ActiveProject.Tasks
intTcounter = 0

'Step through each task
For Each T In Ts

'Only perform the operation on non-blank tasks
If Not T Is Nothing Then
    'Only perform on non-external tasks with Parents
    If Not T.ExternalTask And T.OutlineLevel > 1 Then
        'combine the parent and task names and store in Text20
        T.Text20 = T.OutlineParent.Name & "_" & T.Name
        intTcounter = intTcounter + 1
    Else
        T.Text20 = T.Name
        intTcounter = intTcounter + 1
    End If
End If
Next T
'Display a message that it is complete
MsgBox ("Macro complete" & Chr(13) & intTcounter & " Tasks Changed")
End Sub
```

The code is now complete.

TIP

As you type this code, you will notice some of the features of the VBE. When you press Enter after typing the `Sub` statement (the first line), the VBE automatically adds an `End Sub` statement at the end of the code. All code between the `Sub` and `End Sub` is kept together. (`Sub` is short for *subroutine*.)

After you type **`Dim T As`**, a drop-down list appears, displaying all the possible entries you can type next. Typing **`T`** for Task is enough to uniquely select Task. Press the Tab, Enter, or spacebar key to have the VBE fill in the rest of the command for you. This will then happen all through your code. Whenever there is a limited set of commands, the VBE displays that list. This feature is very helpful in assisting you to spell the command correctly or to choose the right option. If the choices are not what you expect, you are not specifying the right object or using the right method.

The example defines three variables:

```
Dim T As Task
Dim Ts As Tasks
Dim intTcounter As Integer
```

Variables are essential to using VBA because they allow you to store and manipulate data. They are temporary storage locations for whatever you are working on or for information you want to carry over to a later operation. Variables are reserved in memory when they are defined or first used. Because you might want to store different types of information, variables have what is called a *type*. Some common types are strings, which hold alphanumeric data; integers, which hold integer numerical data; and tasks, which hold the necessary information about tasks, including all the properties of a task.

VBA is very flexible in allowing you to create variables at any time and without being strict about determining what type of variables they are. Although this increases the chances that you will get your code to run the first time, it also increases the chances that the code won't run the way you want it to. For example, say you have spelled a variable one way in one place and then spelled it a different way in another place. Visual Basic does not know that you meant the same variable when you typed these different words, and you end up with two separate variables. When you refer to what you think is the correct variable, you might not get the result you expected. The following code will not work the way it should because of the misspelling in the third line:

```
Sub MixUp()
MyTask = ActiveProject.Task(1)
MyTaks.Name = "New Name"
Msgbox (MyTask.Name)
End sub
```

One way to avoid having VBA create extra variables is to have the VBE apply stricter rules. The way to do this is to start a module with the words `Option Explicit`. This tells the VBE to require you to explicitly define the variables before you use them. If this option is set, when you try to run the code for the first time, you get a warning that the variable has not been defined. It highlights the code where the problem has occurred so that you can fix it. Having the VBE check your work like this can save you extra work later, when you are debugging your code.

Because you are going to perform the same operation on all the tasks, you need a variable to be the temporary location for that task. Use `T` and define it as a task as shown here:

```
Dim T As Task
```

You also need a variable to hold the collection of tasks that you are working on. For this you can use `Ts` and define it as a collection of tasks as seen here:

```
Dim Ts As Tasks
```

NOTE

It is not necessary to define either of these variables because you will be working on only one task and one collection of tasks at a time. However, if you were working on more than one open task or project, it would be necessary to use these variables.

The final variable should hold the count of the tasks. You can call this `intTcounter` and define it as an integer because you are sure that there will be no fractional values:

```
Dim intTcounter As Integer
```

If there were, you would use another numeric type, such as single, long, double, or currency.

Each statement begins with the word `Dim`, which is short for *dimension*. Following that is the variable name, which can be whatever you choose, as long as it is not a name used by another object and is not a reserved word. Finally, you specify the variable type. You can use `Dim` with many variables in a single statement by separating them with commas:

```
Dim myTask, yourTask, anotherTask as Task
Dim strMessage, strName, strWords as String
```

Initializing Variables

After you define a variable, you can set its values. Setting values is done in different ways for different types of variables. For example, `intTcounter` is a simple number and it has no properties other than its own value, so you set its value using a simple equals sign:

```
intTcounter = 0
```

`Ts` is a collection of tasks, an object variable, so you need to set it equal to another object. You do this using the `set` keyword. You set `Ts` to be the collection of tasks in the active project (the one that is active in the application you are working on):

```
Set Ts = ActiveProject.Tasks
```

The VBE alerts you if you do this incorrectly by giving you an "object required" error if you try to set a numerical or string variable equal to an object. It gives an "invalid use of property" error if you forget to use the `set` keyword when defining an object.

Setting the variable `T` is done implicitly:

```
For Each T In Ts
```

Because you are using a `For…Next` construction, Visual Basic uses `T` to represent the task it is currently working on and then reassigns it to the next task when it is through with the first. This structure is discussed in more detail in the next section.

TIP

Variables can be declared anywhere in code, as long at they are declared before they are used. However, it is easiest to work with them when they are grouped in blocks at the beginning of code.

The next part of the macro is the heart of it. You want to operate on each task. You could name each task individually and then operate on it, but then your code would have a line for each task. To do this efficiently, there are a number of control structures you can use. In this case, you can use a `For Each...Next` structure:

```
'Step through each task
For Each T In Ts

'Only perform the operation on non-blank tasks
If Not T Is Nothing Then
    'Only perform on non-external tasks with Parents
    If Not T.ExternalTask And T.OutlineLevel > 1 Then
        'combine the parent and task names and store in Text20
        T.Text20 = T.OutlineParent.Name & "_" & T.Name
        intTcounter = intTcounter + 1
    Else
        T.Text20 = T.Name
        intTcounter = intTcounter + 1
    End If
End If

Next T
```

This structure enables you to step through each task in a collection of tasks without having to name them individually. It begins with the `For Each` and ends with `Next`. Each time through the loop, the operations within the loop are performed on a task, and when the code reaches the `Next` statement, the next task is selected and the operations are performed. This continues until all the tasks have been operated on. A few different operations occur within the loop.

TESTING AND NESTING

Another control structure in this macro is the `If...Then...Else` structure. This structure enables you to perform operations only if certain criteria are met. For example, you don't want to do anything to tasks that are external tasks or to tasks that don't have parent tasks. You also don't want to try to do anything on blank lines because they do not have names, and if you try to set their properties, you will get an error:

```
'Only perform the operation on non-blank tasks
If Not T Is Nothing Then
    'Only perform on non-external tasks with Parents
    If Not T.ExternalTask And T.OutlineLevel > 1 Then
        'combine the parent and task names and store in Text20
        T.Text20 = T.OutlineParent.Name & "_" & T.Name
        intTcounter = intTcounter + 1
    Else
        T.Text20 = T.Name
        intTcounter = intTcounter + 1
    End If
End If
```

This code shows that you can nest these statements. The inner statement is executed only if the outer statement is executed. If the outer `If…Then` statement is not true, the statements with the loop are not executed.

TIP

Blank lines are one of the main causes of macro failure. Typical control structures treat a blank line as a task, but it is actually a task defined as "nothing," and thus it has no properties, such as name and number. Fortunately, you can test to see if a task is nothing before you attempt to perform an operation that would cause an error. Testing whether a task is nothing should be the first step whenever you are dealing with a collection of tasks that might contain blank lines.

Control structures appear within other control structures, so following `For Each`, you immediately use an `If` statement to test whether the task is nothing. If it is nothing, you skip to the end of that `If` statement. Typically, programmers indent each level of nesting so that it is easy to find the start and end of each statement. The `End If` should be at the same level of indentation as the `For Each` that begins that statement.

`If` statements are written to be true or false and can contain almost any form of Boolean logic (for example, combining true or false statements by using `AND`, `OR`, or `NOT`). In this case you want to work on a task only if it is not nothing.

Next, you have a more complicated `If` statement that checks two conditions, and then if both conditions are true, you perform an operation. If both conditions are not true, you specify an alternative operation, using the `Else` clause. You could have constructed another `If` statement by using different criteria, but you can keep your code simpler and in one place by using an `Else` clause. `End If` terminates this loop, and the next `End If` terminates the loop that checks whether the task is nothing.

Working with the Object Properties

You do the real work in your macro in the loop. You need to set one of the properties of the task you are working with to be a value that is the concatenation of the name of the parent task, an underscore, and the task's own name. Concatenation works with any string values. A string is a character string, a series of alphanumeric characters that will be treated as text. If a string is being referenced as a property of an object (for example, `T.Name`), quotes are not required around it. However, if you have some text that needs to be recognized as a string, it must be surrounded by quotation marks. Without them, Visual Basic treats the text as an object or a variable and generates an error message. You join strings together by using an ampersand (&) between the items you want to join. Here are two examples of concatenation:

```
myString = "This" & " and That"
myString = "Today is " & Date
```

In the first example `myString` will be `"This and That"`. In the second example `myString` will be `"Today is 6/1/2002"` or whatever the current date is.

The second thing you do in the loop is to increment the counter. Each time the loop iterates, it sets the value of `intTcounter` to `intTcounter + 1`:

```
intTcounter = intTcounter + 1
```

This might seem odd at first, but as long as you remember that the new value is always to the left and the previous value is used on the right side of the equation, it will become second nature to you.

Reporting to the User

At the end of the macro, you want to tell the user that the macro has finished. This is more important with a long macro than with a short one, but the mechanism is the same, regardless of the length of the macro: You use the MsgBox function:

```
MsgBox ("Macro complete" & Chr(13) & intTcounter & " Tasks Changed")
```

A message box can display messages. To create a message for a message box, you need to concatenate some fixed strings and the value that has been stored in the `intTcounter` variable. A box pops up on the user's screen when the macro reaches this point.

Debugging Code

It is not often that code will be completely free of error the very first time you write it, so it is inevitable that you will have to fix errors. *Debugging* is the term given to finding and fixing errors in code.

There are two primary ways that code can fail. The first is that instructions might be written incorrectly. They might use the wrong syntax or have spelling errors. The second is that the instructions you have written are the wrong instructions. They might have perfect syntax, but the way they are ordered or structured may cause them not to do what you expect them to do. The VBE will try to help prevent the first type of error through built-in syntax checking. It assists in the second type by allowing you to set breakpoints and watches to closely monitor what the code is doing step-by-step.

Syntax Checking

When you enter code and move the cursor off a line, the VBE automatically checks the syntax of the line. For example, if you omit `Then` from the end of an `If` statement, the VBE generates a compile error with the message `Expected Then or GoTo`. This feature of the VBE eliminates many errors that might otherwise be hard to find, before you even start testing. The VBE can detect a large number of syntax errors.

Breakpoints, Watches, and the Immediate Window

When you have code with a large number of steps and you know only the initial state and the outcome, it is difficult to figure out where the root of your problem lies. The VBE provides the ability to view your code as it executes and to check the values of your variables. The main tools to do this are breakpoints, watches, and the Immediate window.

The VBE lets you step line-by-line through code, but if you have a large amount of repetitive code, it could take a long time to get to the point in which you are interested. To facilitate this, you can set breakpoints in code. Breakpoints allow code to run freely up until the specific line in which the breakpoint is set. The code then stops, and you can evaluate the status of different variables or use the breakpoint to start stepping through the code line-by-line.

To test code as part of debugging, follow these steps:

1. In the VBE, click anywhere on the `For Each` instruction line and select Debug, Toggle Breakpoint, or press F9. This highlights that row of code in red and places a red dot in the margin to the left.
2. Select Run, Run Sub/UserForm, or press F5. This executes the subroutine the cursor is in. The red line turns yellow, to indicate that it is the next instruction to be executed. A yellow arrow is displayed in the left margin. Project stops executing the code when it reaches your breakpoint.
3. Press F8 once. This makes Project single-step through the code. The `For Each` statement turns red again, and the first `If` statement turns yellow.
4. Hover your mouse over `T` in `(T Is Nothing)`. You should see a ToolTip with `T = 1` in it. This means Task ID 1 is the current active task (each object has a default property; the default property for a task object is its ID).
5. Click and drag with your mouse to select `T Is Nothing`, and then move the mouse over the selected (blue) area. Now the ToolTip says `T Is Nothing = False`.
6. Hover your mouse over each property (`OutlineLevel`, `Summary`, `OutlineParent.Name`, and so on) and see what values they have.
7. Press Ctrl+G. This activates the Immediate window and places the cursor there. You can use this window to investigate all the properties of any active object. You can also use it to test any single lines of code, to assist you in getting them right and deciphering exactly what is happening. For example, if `T` is pointing to each task in turn, you can find out which task it's pointing to when stopped at a specific breakpoint by typing **`?T.Name`** or **`?T.Id`** in the Immediate window and then pressing Enter. The name or ID of the task pointed to by `T` is then displayed.
8. Type **`?T.Name`** and press Enter. `?` is shorthand for `Debug.Print`. The value of the task's name should appear in the next row. The `?` allows you to have the value of any variable or property printed to the Immediate window. A line without the `?` is used to execute a statement.
9. In a blank line, type **`T.Name="Test"`** (because `Test` is the task name displayed by `?T.Name`) and press Enter.

10. Scroll back to and click on the `?T.Name` line and press Enter again. The task name should now be `Test`.
11. Press Alt+Tab to return to Project 2002 and confirm that the task name has changed to `Test`. Use Alt+Tab to get back to the VBE, and then reset the task name by using another `T.Name=` line in the Immediate window.

When you're testing code, you can also try typing **`?Activecell.Task.Name`**. `This` can be very useful, but it only has a few properties. By using the `Task` property in this example, you can access all the task information for the task in the selected row.

Other common and very useful properties of tasks that you need to work with are `Work` and `Duration`. In the Immediate window, go to a blank row and type **`?T.Duration`** and then press Enter. The answer should be a rather large number that looks nothing like what you see in the Duration column of Project 2002.

Project stores all `Duration` and `Work` values in minutes. To convert these to days, divide by 60 and then by the number of hours per day listed in the Calendar tab of the Options dialog box. The default number of hours per day is 8, so `?T.Duration/60/8` should display the correct number of days' duration.

TIP

`ActiveProject.HoursPerDay` should give you the number of hours per day to use in the formula instead of 8. Try typing **`?T.Duration/60/ActiveProject.HoursPerDay`** in the Immediate window to confirm this.

You have already learned that the variable `T` is `Nothing` until the `For Each` loop, when it points to each task in turn. You also need to be aware that the variable has no value at all in the Immediate window if the macro hasn't stopped at a breakpoint.

If you type `?T.Name` in the Immediate window when the code hasn't stopped at a breakpoint, an error is generated. `T` only points to a task after the `For Each T In ActiveProject.Tasks` statement has been executed. After `End Sub` is reached, `T` has no value, and any attempt to use it in the Immediate window fails.

TIP

Remember that if you want more information on a property or object, you can click it and press F1 to call up the help system.

When the code has stopped at a breakpoint, notice that, in the gray border on the left of the VBE, the yellow line has a yellow arrow and the breakpoint has a red spot. Click the red spot to toggle the breakpoint off or on, or click anywhere on the line's code and press F9 again.

You can also click and drag the yellow arrow to move the yellow line (changing which line will be executed next). Try moving it back to the `For Each` row. Press F8 again and again, and watch the code being stepped through. At any time you can press F5 to have it run continuously to the next breakpoint or the end of the subroutine, whichever comes first.

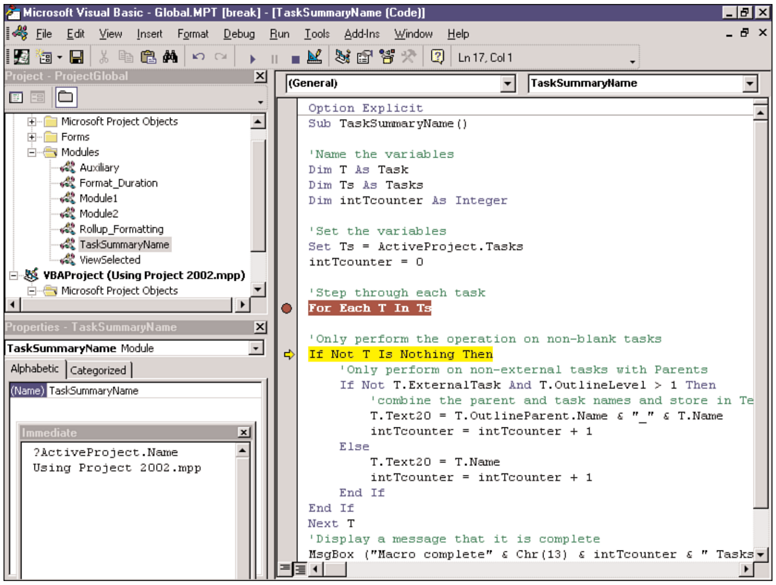
Enter a line like `Set T=ActiveProject.Tasks(1)` in the Immediate window and press Enter, and the line is executed. This is a good way to get a line of code working properly before you copy it back to the editing pane.

Remember that when you click any instruction and press F9, the code can be made to stop. You then hover your mouse over any variable to get a ToolTip that displays its value, or you can use `?` to print values in the Immediate window. You can also single-step through the code to see exactly what is happening, correcting the code as you go. You can also use Alt+Tab to swap between the VBE and Project (or any other program you are controlling from Project VBA) to look at what the code has done.

There are two alternatives to using the Immediate window. The first is the Locals window, which you display by selecting Locals from the View menu. The Locals window contains the names, values, and properties of the objects that are being evaluated in code. Because it shows every possible piece of information about the objects, it can be a bit difficult to navigate through, but it is helpful if you aren't exactly sure of the specific property you are looking for.

The second alternative is the Watch window, which you display by selecting Watch from the view menu. This window first appears with no watches set. To set a watch, highlight the item in the code that you want to watch, and then go to the Watch window, right-click, and select Add Watch. You can set a watch to display the value of an expression (just as you can get the value by hovering the mouse above the variable). You can also set a watch to break when the value of an expression you are watching turns either true or false. Figure 6 shows the code stopped where the expression `"T is Nothing"` has become true.

Figure 6
The Watch window shows watches for an expression `"T is Nothing"` and for a variable.


Microsoft Visual Basic - Global.MPT [break] - [TaskSummaryName (Code)]
File Edit View Insert Format Debug Run Tools Add-Ins Window Help
Ln 17, Col 1
Project - ProjectGlobal
Microsoft Project Objects
Forms
Modules
Auxiliary
Format_Duration
Module1
Module2
Rollup_Formatting
TaskSummaryName
ViewSelected
VBAProject (Using Project 2002.mpp)
Microsoft Project Objects
Properties - TaskSummaryName
TaskSummaryName Module
Alphabetic
Categorized
(Name) TaskSummaryName
Immediate
(General)
TaskSummaryName

```
?ActiveProject.Name
Using Project 2002.mpp
```

```
Option Explicit
Sub TaskSummaryName()

'Name the variables
Dim T As Task
Dim Ts As Tasks
Dim intTcounter As Integer

'Set the variables
Set Ts = ActiveProject.Tasks
intTcounter = 0

'Step through each task
For Each T In Ts

'Only perform the operation on non-blank tasks
If Not T Is Nothing Then
    'Only perform on non-external tasks with Parents
    If Not T.ExternalTask And T.OutlineLevel > 1 Then
        'combine the parent and task names and store in Te
        T.Text20 = T.OutlineParent.Name & "_" & T.Name
        intTcounter = intTcounter + 1
    Else
        T.Text20 = T.Name
        intTcounter = intTcounter + 1
    End If
End If
Next T
'Display a message that it is complete
MsgBox ("Macro complete" & Chr(13) & intTcounter & " Tasks
```

If you know what you are looking for, using the Watch window is the simplest way to display the values you want.

THE `TraceDependencies` MACRO

VBA is most useful when it does something for you that is really difficult or time-consuming for you to do by hand. The `TraceDependencies` macro discussed in this section demonstrates some useful control structures and concepts.

When schedules get large, it can become difficult to trace the relationships between activities. You can either try to trace the lines in the Gantt chart or look at the Network view. Both these options can be challenging at times.

It would certainly be useful to be able to filter the display so that only the related tasks are shown. With the advent of overhead projectors and electronic collaboration allowing schedule review and analysis in real-time, it is also important to be able to do this quickly. The `TraceDependencies` macro is meant to solve the problem.

First, let's clearly define the problem. You want to find all the tasks related to a selected task. You would like to be able to look at all the predecessors and successors. In some cases, only one or another of these is necessary, so you want to give the user a choice. On occasion you might want to narrow your view even further and just see the tasks that are on the critical path.

Earlier in this chapter you learned that sometimes the summary task provides necessary context for figuring out what a certain task is if there are many similarly named tasks. Other times, they just clutter things up. You would like the user to be able to choose. Because the display will shift, the user should be able to find the task that he or she had originally selected, even if it has moved on the screen.

Here is the code for the `TraceDependencies` macro:

```
'   This macro filters the project to show the
'   predecessors and/or successors of a
'   selected task depending on the user input
'   This macro works best if assigned to a button on the toolbar
'   It uses Flag5 to store information -
'   Please be sure that this field is NOT being used for other purposes
'   Note:   It does not trace across external (Cross project) links.

Option Explicit
Dim strFanType As String
Dim boolSum As Boolean
Dim gboolboolCrit As Boolean
Dim T, TT, Tselect As Task

'This is the master macro
Sub Trace()

'If selection is more than one task or a blank line warn and then quit the macro
If ActiveSelection.Tasks.Count <> 1 Or ActiveSelection.Tasks(1) Is Nothing Then
MsgBox "You must have just one task selected for this macro to work"
    Exit Sub
End If
```

```
'Assign the variable for the selected task
Set Tselect = ActiveSelection.Tasks(1)
'If the selected task is a summary task, warn and then quit macro
If Tselect.Summary = True Then
    MsgBox ("You have selected a summary task. " _
    & "Select a task or milestone and try again")
    Exit Sub
End If

'This sets flag used later for tracing paths.
strFanType = InputBox(("Please Enter Fan Type" & vbCr _
            & vbCr _
            & "P   (Predecessors)" & vbCr _
            & "S   (Successors)" & vbCr _
            & "A   (All)") & vbCr _
            & "Leave Blank to quit here", "Fan-out Dependencies")

'Convert the input into correct case if necessary
strFanType = UCase(Left(strFanType, 1))

'Quit if no information is typed
If strFanType = "" Then
    Exit Sub
End If

'Clear the flag used to show tasks
ClearFlags

'Set the flag which determines if only critical tasks are shown
gboolboolCrit = False
If Tselect.boolCritical = True Then
    If MsgBox("Do you want to display only boolCritical Tasks?", _
    260, "Display boolCritical Tasks Only?") = vbYes Then
        gboolboolCrit = True
    End If
End If

'Use the input about what the user wants to trace
'to determine what action to take
Select Case strFanType
    Case "P"
        'Traces Only Predecessor Tasks
        FanBackward Tselect, gboolboolCrit
    Case "S"
        ' Traces Only Successor Tasks
        FanForward Tselect, gboolboolCrit
    Case Else
        ' Traces All Tasks - one pass for successors, then one for predecessors
        FanForward Tselect, gboolboolCrit
        FanBackward Tselect, gboolboolCrit
End Select

'Run a subroutine to filter the activities
FilterMe

'Make sure that the original task is still selected
Find Field:="ID", Test:="equals", Value:=Tselect.ID ', 'Next:=True
```

```
End Sub

'Set all tasks Flag5 to false
Private Sub ClearFlags()
    For Each T In ActiveProject.Tasks
        If Not (T Is Nothing) Then
            T.Flag5 = False
        End If
    Next T
End Sub

' Walks through all successors to a task and marks their flag5 as true
Sub FanForward(T As Task, boolCrit As Boolean)
Dim TT As Task
T.Flag5 = True
    For Each TT In T.SuccessorTasks
        If TT.Flag5 <> True Then
            If Not boolCrit Then
                FanForward TT, boolCrit
            End If
            If boolCrit And TT.boolCritical Then
                FanForward TT, boolCrit
            End If
        End If
    Next TT
End Sub
' Walks through all predecessors to a task and marks their flag5 as true
Sub FanBackward(T As Task, boolCrit As Boolean)
Dim TT As Task
T.Flag5 = True
    For Each TT In T.PredecessorTasks
        If TT.Flag5 <> True Then
            If Not boolCrit Then
                FanBackward TT, boolCrit
            End If
            If boolCrit And TT.boolCritical Then
                FanBackward TT, boolCrit
            End If
        End If
    Next TT
End Sub

' Subroutine which will Filter with or without summary tasks
Private Sub FilterMe()
Dim V As View
Dim Vis As Boolean
Vis = False

'Ask the user if they want to show Summary tasks as well
If MsgBox("Do you want to display Summary Tasks?", _
    vbYesNo, "Display Summary Tasks?") = vbYes Then
    boolSum = True
Else: boolSum = False
End If

'Construct the filter
FilterEdit Name:="_Trace", TaskFilter:=True, _
    Create:=True, _
    OverwriteExisting:=True, _
    FieldName:="Flag5", _
    Test:="Equals", _
    Value:="Yes", _
    ShowInMenu:=False, _
    ShowSummaryTasks:=boolSum


'Check to see if view exists
For Each V In ActiveProject.Views
If V.Name = "Trace" Then
                    Vis = True
End If
Next V

'If it doesn't then create it
If Not Vis Then
ViewEditSingle Name:="Gantt Chart", _
    Create:=True, _
    NewName:="Trace", _
    Screen:=1, _
    ShowInMenu:=True, _
    HighlightFilter:=False, _
    Table:="Entry", _
    Filter:="_Trace", _
    Group:="No Group"
End If
ViewApply Name:="Trace"
OutlineShowAllTasks

End Sub
```

TIP

This code starts with a set of comments that describe what it does, which fields it uses, and some limitations. These code comments are often the only documentation available to you and to users, so spend some time to use comments to record any particulars or instructions if they aren't obvious.

Notice that several of the control structures in this macro are the same as those from the previous examples. Also notice that there are several subroutines here, for two main reasons:

- **To help you write the code**—Just as eating an elephant is best done one bite at a time, writing and debugging code works best if you break the task into smaller parts. After you get small parts of the code written and working, you can add additional parts.
- **So blocks of code can be reused either within the same module or in some other place**—If there is a function you use twice or more in the same macro, it is worth making it a separate subroutine and then calling on that subroutine whenever you need it. That way you don't have to type the same code twice, and if you have to fix it, you need to fix it in only one place instead of searching for it in the various places you have used it and fixing it in each place.

The following sections walk through the code of the Trace Dependencies macro, looking at the new structures and concepts.

Public Versus Private Variables

Several variables are defined before any of the subroutines begin. This happens because some of these variables need to be available for more than one of the subroutines. If a variable is defined within a subroutine, it is not visible to other subroutines or procedures. If you define these variables at the module level, they are available within the module. If you preface them with the keyword `Public`, they are also available to other modules. By default they are private to the module. You can be explicit about this and declare them as private by using the `Private` keyword. At the module level, definition of any variables must occur before any subroutines in the module. Again, grouping variables in the same location so they can easily be found will make things easier for you.

Subroutines within a module are by default public and are accessible to other modules, unless they are defined as private. Generally, it is good practice to keep variables as private as possible, to avoid any problems because similarly named variables appear in other modules.

Note that several task variables are declared with a single `dim` statement; their names are separates with commas:

```
Dim T, TT, Tselect As Task
```

Some Boolean (true/false) and string variables are declared as well. Variable types, including definitions, can be found in the Microsoft Project Visual Basic help.

The Main Subroutine

The *main subroutine* is the one that you would call by name if you wanted to run the macro or assign it to a toolbar button.

The first thing it does is make sure that you have a task, and only one task, selected. This macro could be modified to handle more than one selected task, but at this time it does not support that.

The following line combines two conditions as a first test:

```
If ActiveSelection.Tasks.Count <> 1 Or ActiveSelection.Tasks(1) Is Nothing Then
```

Because of the `OR`, only one of the conditions needs to pass in order for the macro to move to the next operation. Once again, you must check whether the task is `Nothing`, or the macro will fail.

`ActiveSelection` is a collection of tasks. If you want to refer to an individual task within that collection, you need to refer to it by name or by its index number. Because you don't know the name, you refer to the first task in the collection by using the index 1. Because the `Is Nothing` test works on only the first task in the selection, it would not be sufficient by itself, but because you are also testing whether there are more or fewer than one tasks, the only time it needs to work is when one task is selected; therefore, the use of 1 for an index will always be safe.

If either of these tests is positive, the rest of the code in the `If` statement is executed, and a message box stating the problem is displayed. The next statement quits the macro:

```
Exit Sub
```

Because continuing with selection that the rest of the macro can't handle would cause an error, exiting the subroutine is a sound error-avoidance practice. If this test is positive, the next line:

```
Set Tselect = ActiveSelection.Tasks(1)
```

sets the variable `Tselect` to the task that is selected in the `ActiveProject` file.

REQUESTING USER INPUT BY USING THE INPUT BOX

When a valid task is selected, you need to ask the user what he or she wants to do. You could have asked this earlier, but it saves the user trouble if you first check the input to make sure it is okay before the user takes the time to enter choices.

The following code solicits input from the user:

```
strFanType = InputBox(("Please Enter Fan Type" & Chr(13) _
            & Chr(13) _
            & "P   (Predecessors)" & Chr(13) _
            & "S   (Successors)" & Chr(13) _
            & "A   (All)") & Chr(13) _
            & "Leave Blank to quit here", "Fan-out Dependencies")
```

To get input from the user, you can use an input box. An input box is similar to a message box, except that it has a space for the user to enter text. You use the string variable `strFanType` to hold the user's response. Notice that the text to be displayed as a user prompt is fairly long. The line continuation character (_) is used to continue a statement from one line to another. This allows you to view the code within the VBE code window without having to scroll. When you put a line continuation character after each carriage return, the prompt text is shown similar to the way it would be in the input box itself.

You have set `strFanType` to be whatever text is typed by the user in the box. Of course, the user might type gibberish or have Caps Lock on, so the response might be different from what you expect. You can handle capitalization problems by converting the case of whatever the user types to uppercase. If the user types nothing, the macro once again exits the subroutine.

REQUESTING USER INPUT BY USING THE MESSAGE BOX

An alternative to using an input box is to use a message box. Earlier in the chapter you used a message box to display a message, but it is also useful for asking users to make a choice about something. A message box displays a message, waits for the user to click a button, and returns an integer indicating which button the user clicked.

The following code creates a message box that has Yes and No buttons and that asks if the user wants to display only the critical tasks:

```
If Tselect.Critical = True Then
    If MsgBox("Do you want to display only Critical Tasks?", _
        260, _
        "Display Critical Tasks Only?") = vbYes Then
        gboolCrit = True
    End If
End If
```

The number `260` indicates the type of buttons and which is the default or highlighted button. You can find details about the values that can be used in place of `260` in the Microsoft Project Visual Basic help. Note that this box is displayed only if the task the user has selected is a critical task. If the task is not critical, then it should have few, if any, critical tasks that are dependent on it, so the code avoids an unnecessary choice by asking this question only when it is relevant.

CALLING A SUBROUTINE WITHOUT PARAMETERS

After you have checked the input and verified that the user wants to actually do something, you can start doing some work. You need to clear the flag that you will be using to identify the tasks that are linked to the selected task. You could clear them all at the beginning, but because that will take some computation and might take some time, it is better to do it after you have checked for valid input and user intention.

Calling a subroutine is very easy. You simply enter the name of the subroutine. The one in the Trace Dependencies macro is simple because you are not passing any parameters so the statement is as follows:

```
ClearFlags
```

If you look at the code for `ClearFlags`, you will see that it uses a `For Next` structure, just like in the `TaskSummary` macro. It is declared as private because you want to keep it local to the module. Because clearing fields is a common thing to do, you might have other `ClearFlags` subroutines elsewhere to clear other fields.

After this subroutine runs, control returns to the main subroutine. An `Exit Sub` inside this subroutine also returns control to the main subroutine because it is nested inside.

USING A `Case` STATEMENT

One of the easiest ways to allow code to branch if there are more than a couple possible choices is to use a `Case` statement. A `Case` statement evaluates or reads a variable and executes different statements for each of the cases you have defined.

In the following example, you store the user's response in a variable called `strFanType`:

```
Select Case strFanType
    Case "P"
        'Traces Only Predecessor Tasks
        FanBackward Tselect, gboolCrit
    Case "S"
        ' Traces Only Successor Tasks
        FanForward Tselect, gboolCrit
    Case Else
        ' Traces All Tasks - one pass for successors, then one for predecessors
        FanForward Tselect, gboolCrit
        FanBackward Tselect, gboolCrit
End Select
```

The Select statement reads the value of strFanType, and if the value matches any of the cases, the code for that case is executed. P traces predecessors and S traces successors, and if the input is anything other than S or P, the code for the case Else, which traces both predecessors and successors, is executed.

The keyword Else is used to catch anything other than the two cases that were defined before it. You could be stricter in checking the input and then you could construct a Select statement with only the three choices allowed, but because you are less strict, a user's typing error does not cause the macro to fail; it still runs and shows all the dependent tasks.

You can see that even with only three cases, this structure is easier than writing three separate If…Then statements and because all the choices are in the same place, it is easier to read, debug, and maintain. One thing to watch out for in a Select statement is that the cases are in the correct order. A Select statement runs the first case that is true, so you need to use care if you are selecting from ranges.

CALLING A SUBROUTINE WITH ARGUMENTS

After you have captured the information you need to begin the work of tracing dependencies, it is time to do the tracing. You actually have two subroutines that trace the tasks. One traces successors and one traces predecessors, but other than that, they are very similar. Unlike the ClearFlags subroutine, this subroutine has two additional variables, called *arguments*.

The following code calls the function FanBackward; the variables following FanBackward are the arguments:

```
Select Case strFanType
    Case "P"
        'Traces Only Predecessor Tasks
        FanBackward Tselect, gboolCrit
```

Arguments are values that are passed along to a subroutine in order for it to do its work. In the following subroutine you pass the task you want to use as a starting point and a value that tells the subroutine whether you are only tracing critical tasks:

```
' Walks through all predecessors to a task and marks their flag5 as true
Sub FanBackward(T As Task, boolCrit As Boolean)
Dim TT As Task
T.Flag5 = True
    For Each TT In T.PredecessorTasks
        If TT.Flag5 <> True Then
            If Not boolCrit Then
                FanBackward TT, boolCrit
            End If
            If boolCrit And TT.Critical Then
                FanBackward TT, boolCrit
            End If
        End If
    Next TT
End Sub
```

On the first line of the subroutine, instead of an empty pair of parentheses, you have two items within the parentheses. These are the arguments that the subroutine uses as input. The first is defined as a task, so you can pass any variable that is a task variable. The second, `boolCrit`, is defined as Boolean. You can pass any variable that is a Boolean type to this subroutine. The ability to write subroutines with arguments makes your code more flexible and reusable. In this example, it is essential that you use arguments because you are passing the predecessors of the selected task to the macro, and you don't know in advance what the tasks in the chain are going to be.

The other advantage of using subroutines that take arguments is that they can be used from other subroutines without requiring you to rename the variables you are using. As long as the variables are of the same type as the arguments, you can pass those objects or values along. In the Trace Dependencies macro, you pass `Tselect` and `gboolCrit` to the subroutine. Within the subroutine, they are initially referred to as `T` and `Crit`.

RECURSION

You should recognize most of the control structures in the `FanBackward` subroutine, but there is one new element. This subroutine calls itself while it is still running by using this line:

```
FanBackward TT, boolCrit
```

This process is called *recursion*. A *recursive* procedure is one that calls itself, and it is very useful if you are trying to trace a hierarchy. In this case we are tracing predecessors. We select a predecessor and then call the function to select each of its predecessors. You use recursion here because you want to do the same thing to each of the predecessors that you are doing to the initial task. You also want to do the same thing to each of the predecessor's predecessors and so on.

TIP

You need to be careful with recursion because it is possible to create a recursive procedure that does not have an end. Each time the procedure is run, a certain amount of memory is reserved for it. If the process continues to run thousands or millions of times, it will eventually run out of memory and cause the application to fail or crash. To prevent this you use a "base case," where the procedure stops calling itself. Because the procedure in the `FanBackward` subroutine executes once for each predecessor to a task and you know that the predecessors are finite in number, you can be certain that the procedure will stop.

Recursion can be a bit confusing, so let's look at what happens, step-by-step:

1. First, `Flag5` for the task we are working on is set to true:

   ```
   T.Flag5 = True
   ```

   This is the field you will be using to filter the project on later.

2. You step through each of the selected tasks predecessors. You use the variable TT to hold the predecessor task that you are working on:

   ```
   For Each TT In T.PredecessorTasks
   ```

3. You test to see if Flag5 for that task has already been set to true:

   ```
   If TT.Flag5 <> True Then
   ```

   If it is true, you have already traced that path, perhaps while tracing the predecessors to another activity. To be efficient, you can skip the branches that have already been traced.

4. After testing to see if you are looking for only critical tasks, you call FanBackward again, passing along the task TT this time:

   ```
   If Not Crit Then
       FanBackward TT, Crit
   End If
   ```

   The subroutine begins again and follows all the predecessors of that task, and as it goes through them one by one, it traces all their predecessors, until it runs out of predecessors to trace. Then because of the For Each statement, it goes to the next predecessor in the collection and does the same thing.

As you can see, recursion can be a powerful tool when you're tracing dependencies, objects with a parent/child relationship, or any other sort of hierarchy. For example, the following code fills in the Text5 field with a string that shows all the tasks' parents and the parents' parents:

```
Sub Inherit(T as Task)
T.Text5 = T.Name
Genes T
End Sub

Sub Genes(T As Task)
Dim TT As Task
For Each TT In T.OutlineChildren
   TT.Text5 = T.Text5 & "_" & TT.Name
   Genes TT
Next TT
End Sub
```

CONTROLLING FILTERING AND VIEWS

After you have correctly set Flag5 for all the tasks in the hierarchy, you are almost ready to set the display to show only those tasks. To do that, you use the FilterMe subroutine, which gathers input from the user and then constructs and applies a filter. The following is the FilterMe subroutine:

```
Private Sub FilterMe()
If MsgBox("Do you want to display Summary Tasks?", _
        vbYesNo, _
        "Display Summary Tasks?") = vbYes Then
    boolSum = True
Else: boolSum = False
```

```
End If

'Construct the filter
FilterEdit Name:="_Trace", _
        TaskFilter:=True, _
        Create:=True, _
        OverwriteExisting:=True, _
        FieldName:="Flag5", _
        Test:="Equals", _
        Value:="Yes", _
        ShowInMenu:=False, _
        ShowSummaryTasks:=boolSum

'Check to see if view exists
For Each V In ActiveProject.Views
    If V.Name = "Trace" Then
        Vis = True
    End If
Next V

'If it doesn't then create it
If Not Vis Then
    ViewEditSingle Name:="Gantt Chart", Create:=True, NewName:="Trace", _
    Screen:=1, ShowInMenu:=True, _HighlightFilter:=False, _
    Table:="Entry", Filter:="_Trace", Group:="No Group"
End If
ViewApply Name:="Trace"
OutlineShowAllTasks
```

The first part of this subroutine should be familiar to you because it is similar to the message box you worked with earlier in the chapter. Next, in the subroutine, you expand the view to show all tasks. This is an important step because Project filters based on the visible tasks. If some of the task summaries are collapsed, the tasks within those summaries will not be shown.

Next, the subroutine creates a filter. The syntax for creating a filter can be a bit complex, so this filter was created by recording a macro to get the basic structure and then putting the variable `boolSum` in to pass the value that you get from the user. The subroutine creates the filter each time you use it and overwrites the previous version because it must include the new value for `boolSum`, which controls the display of summary tasks.

One of the new objects in Project 2002 is the `View` object. You want to have a separate view to display the filtered tasks so that the user's original view is preserved and he or she can easily flip back and forth between the traced tasks and the complete project. The following code checks to see if the view exists, and if it does not exist, it creates a view named `Trace`:

```
'Check to see if view exists
For Each V In ActiveProject.Views
    If V.Name = "Trace" Then
        Vis = True
    End If
Next V
```

```
'If it doesn't then create it
If Not Vis Then
    ViewEditSingle Name:="Gantt Chart", Create:=True, NewName:="Trace", _
    Screen:=1, ShowInMenu:=True, _HighlightFilter:=False, _
    Table:="Entry", Filter:="_Trace", Group:="No Group"
End If
ViewApply Name:="Trace"
OutlineShowAllTasks
End sub
```

The check for existing views is important because you might run this macros many times, and you don't want to create a new view each time. (Note that the view is defined with the `_Trace` filter, which you created earlier as one of the arguments.) When you have finished this, you apply the view. Then you expand the outline to show all outline levels to make sure no tasks are hidden. The subroutine then exits and control returns to the main Trace Dependencies macro.

The final statement in the macro is used to once again select the original task:

```
Find Field:="ID", Test:="equals", Value:=Tselect.ID
```

You can use the `Find` method to find tasks in different fields and to select or highlight the task in the display.

Working with Other Applications

One of the valuable features of VBA is that you can use it in Project to control other applications that have VBA. Often when you're working in Project, you want to be able to export to Excel with a bit more intelligence than is provided by the simple export maps.

For example, you might want to be able to export the tasks and retain the hierarchy that exists in the project file. The code to do this does a fairly simple export of just the task names, but you can easily extend it to export other columns, containing duration, start dates, task percentage complete, or any of the fields available in Project.

All the objects, methods, and properties in Excel can be accessed and controlled by Project. To do this, Project needs to reference the Excel object model, which is contained in the Microsoft Excel Object Library. This library is present on any computer that has Excel installed; however, it must be referenced within the VBE before you can begin working.

To set a reference to the Excel Object Library, select Tools, References. You should see a list of all the libraries that are available on the computer you are working on. Scroll down until you find the Excel library, and place a checkmark in the box next to it.

If you were giving instructions to a person, you would ask that he or she follow these steps:

1. Open a copy of Excel.
2. Create a sheet and give it the same name as the project.
3. Put some identifying information and column headers in place.
4. Export the tasks one by one and indent them.
5. When the work is done, return a message to the user, stating that the work is done.

Here is the translation of those steps into Visual Basic statements:

```
Option Explicit

Sub TaskHeirarchy()
Dim xlApp As Excel.Application
Dim xlBook As Excel.Workbook
Dim xlSheet As Excel.Worksheet
Dim xlRow As Excel.Range
Dim xlCol As Excel.Range
Dim Proj As Project
Dim T As Task
Dim ColumnCount, Columns, Tcount As Integer

Tcount = 0
ColumnCount = 0

Set xlApp = New Excel.Application
xlApp.Visible = True
AppActivate "Microsoft Excel"
xlApp.Cursor = xlWait
Set xlBook = xlApp.Workbooks.Add
Set xlSheet = xlBook.Worksheets.Add
xlSheet.Name = ActiveProject.Name

'Set Range to write to first cell
Set xlRow = xlApp.ActiveCell
xlRow = "Filename: " & ActiveProject.Name
Set xlRow = xlRow.Offset(1, 0)
xlRow = Date()
Set xlRow = xlRow.Offset(2, 0)

'Write each task and indent to match outline level
For Each T In ActiveProject.Tasks
    If Not T Is Nothing Then
        Set xlRow = xlRow.Offset(1, 0)
        Set xlCol = xlRow.Offset(0, T.OutlineLevel - 1)
        xlCol = T.Name
        If T.Summary Then
            xlCol.Font.Bold = True
        End If
        Tcount = Tcount + 1
    End If
Next T

'Switch back to Project and display completion message
xlApp.Cursor = xlDefault

AppActivate "Microsoft Project"
MsgBox ("Macro Complete with " & Tcount & " Tasks Written")
AppActivate "Microsoft Excel"
End Sub
```

This code begins with a definition of the variables and sets initial values for some of the variables that you will be using. Because the code is working with Excel as well as Project, the variables include some objects from Excel:

```
Dim xlApp As Excel.Application
Dim xlBook As Excel.Workbook
Dim xlSheet As Excel.Worksheet
Dim xlRow As Excel.Range
Dim xlCol As Excel.Range
```

You already have Project open if you are running this macro, but it is possible that Excel is not started. The procedure must start Excel before it can export data to it.

To start Excel, follow these steps:

1. Create an instance of Excel by setting the variable `xlApp` to a new instance of Excel:
   ```
   Set xlApp = New Excel.Application
   ```
2. Set the application to be visible so that you can see it and activate it, which brings it to the front of your desktop:
   ```
   xlApp.Visible = True
   AppActivate "Microsoft Excel"
   ```
3. While you are processing the macro, you need to also set the cursor to an hourglass so users will know that something is going on and they should wait:
   ```
   xlApp.Cursor = xlWait
   ```
4. Create a new workbook, similarly to the way you created an instance of Excel, and then add a worksheet to hold the data you are exporting. To make sure that the user can find this sheet easily, you should give it the same name as the project:
   ```
   Set xlBook = xlApp.Workbooks.Add
   Set xlSheet = xlBook.Worksheets.Add
   xlSheet.Name = ActiveProject.Name
   ```

With Excel open and active, the procedure can start writing data to Excel. The steps the procedure performs to write are as follows:

1. Select which cell to write to. By default, the active cell in a new worksheet is the first cell (A1). The procedure takes the predefined range and sets it to that cell:
   ```
   Set xlRow = xlApp.ActiveCell
   ```
2. Set the value of that range to be the value you would like. As in this example, it is set to project name:
   ```
   xlRow = "Filename: " & ActiveProject.Name
   ```
3. After the first cell is written in, move to another cell for the next piece of data. This uses the offset method. The values in the parentheses are the offset in number of rows and columns, respectively. In the new cell write `OutlineLevel`:
   ```
   Set xlRow = xlRow.Offset(1, 0)
   xlRow = "OutlineLevel"
   ```
4. Continue to move through the spreadsheet in a similar manner. Because the procedure will be moving down the spreadsheet and across the spreadsheet, two different ranges (or pointers) have been defined. The first, `xlRow`, moves down the sheet, offsetting row by row. The second, `xlCol`, moves across the spreadsheet. By using offsets from `xlRow`, `xlCol` can be held stationary at any column and still keep `xlRow` moving down row by

row in the first column. Using this method makes it easy to keep track of the position of the cell that is being writing to.

5. The procedure loops through all the tasks and offset the range an amount equal to the outline level of the task:

```
Set xlCol = xlRow.Offset(0, T.OutlineLevel - 1)
        xlCol = T.Name
```

   Each time the range moves down a line, the task counter is incremented by 1 and the cells are set to bold if the task is a summary task:

```
If T.Summary Then
            xlCol.Font.Bold = True
```

6. The procedure resets the cursor to the default, so after it finishes writing, the user is not stuck with an hourglass when he or she goes to the spreadsheet.
7. Switch back to Project to display a completion message.
8. Switch back to display the finished spreadsheet.

There are many things you could add to this macro. As in Project, if you can do something manually in Excel, there is almost always a way to do it automatically by using Visual Basic, so if you need special formatting or graphing of the data, it is possible to include that in your macro.

EXPORTING TO A TEXT FILE

Although Project interacts quite well with Excel and other Microsoft applications, sometimes you might want the data in a different format. The following example reads data from a Project file, modifies it slightly, and writes the results to a text file:

```
Option Explicit

Sub NoteFile()
Dim MyString As String
Dim MyFile As String
Dim fnum As Integer
Dim myTask As Task

'set location and name of file to be written
MyFile = "c:\" & ActiveProject.Name & "_Project_Notes" & ".txt"

'set and open file for output
fnum = FreeFile()
Open MyFile For Output As fnum

'Build string with project info
MyString = ActiveProject.Name _
        & "  " _
        & ActiveProject.LastSaveDate _
        & "  " _
        & Application.UserName
```

```
'write project info and then a blank line
Write #fnum, MyString
Write #fnum,

'step through tasks and write notes for each then a blank line
For Each myTask In ActiveProject.Tasks
    If myTask.Notes <> "" Then
        'edit the following line to include the fields you want
        'use " " to include any text or spaces.
        MyString = myTask.UniqueID & ": " & myTask.Name & ": " & myTask.Notes
        'Some other Examples: MyString = MyTask.Text1 & ": " & MyTask.Start
        Write #fnum, MyString
        Write #fnum,
    End If
Next myTask
Close #fnum
End Sub
```

Opening a File for Writing

To write to a file, you use the `Open` statement, which either opens a file or creates one. The `Open` statement requires a number of arguments, including the pathname to the file, the mode (which controls the behavior), and a file number (which it obtains by using the `FreeFile()` function). The key things to understand here are how to create and write to the file.

To use the `Open` statement, follow these steps:

1. Get the arguments ready to execute the `Open` statement. You can use the variable `MyFile` to hold the name of the file, which we create by concatenating the project name and some other text:

   ```
   MyFile = "c:\" & ActiveProject.Name & "_Project_Notes" & ".txt"
   ```

2. Use the `FreeFile()` function to find the next available file number, and use `it` to create a file object that you can use within the macro:

   ```
   fnum = FreeFile()
   ```

3. Open the file for writing:

   ```
   Open MyFile For Output As fnum
   ```

4. Control what is written, by using a `write #` statement, which takes a file number and an expression to write. For example, you can write a string that we created earlier and then execute a `write` statement with no expression you write a blank line:

   ```
   Write #fnum, MyString
   Write #fnum,
   ```

The rest of the macro simply steps through all the tasks and writes the task unique ID, the task name, and the task notes. When you are done, close the file. This macro outputs text by using the input mode and `Read #` statements, but you could bring in the contents of a text file or a `.csv` file that might be generated by some other application or that contains configuration information.

Working with Events

Another powerful feature in VBA is the ability to use project-level and application-level events to trigger the execution of some code that you have written. Project events occur when the project changes, and Application events occur when a project is created. There are fewer Project events, and they are simpler to use, so they are covered here first.

An example of a project-level event is the `Project_Open` event. You can write some code, and whenever that project opens, the code can automatically be executed. A typical use of the `Project_Open` event is to trigger setting up the environment by customizing the toolbars and menus. Another use would be to open and write to a file use log. This section uses the event to pop up a simple window.

The `Project_Open` event is specific to the project it exists in. You need to follow these steps to create code that will execute on the `Project_Open` event in your file:

1. Create a new project by selecting File, New.
2. When the file is open, go to the VBE by pressing Alt+F11.
3. Go to the Project Explorer. If it is not visible, you can display it by pressing Ctrl+R or selecting it from the View menu.
4. Navigate to the `ThisProject` object and double-click it. This opens a code window for the project. This code window is similar to that of the modules you created earlier. At the top of the window are two drop-down lists. On the left is the Object list. Use this list to select the `Project` object. This creates a shell for you, and you can write your code within this shell. Notice that the drop-down list on the right now shows `Open`. You can select other events from the list now.
5. The `pj as Project` part of the code specifies a Project object, just as you specified the arguments in subroutines earlier in this chapter. This gives your code an object to work with. You don't have to worry much about this because Project puts this in for you automatically.
6. Inside the subroutine, you want to pop up a message box indicating the project name and the last time it was saved. You do this with the following code:
   ```
   MsgBox (pj.Name & vbCr & "Last Saved: " & pj.LastSaveDate)
   ```
7. Save the project, and the next time you open it, the code will run.

The following Project events are also available:

```
Activate
BeforeClose
BeforePrint
BeforeSave
Calculate
Change
```

`Deactivate`

`Open`

WORKING WITH APPLICATION-LEVEL EVENTS

The difference between Application events and Project events is confusing. Microsoft says, "Project events occur when the project changes. Application events occur when the project is created." Perhaps a better way to think of this is that the Project events are triggered only when the `Project` object changes or is operated on. Application events occur when objects within a project change or are operated on.

With Application events, you can respond to most of the important actions a user might take in Project, such as changing or deleting tasks, resources, or assignments. There are also a number of events provided so that you can monitor changes that the user makes in the environment. Those events can be used when elements of Project are used in a digital dashboard or another Web application.

The entire list of Application events is too long to reproduce here. They are well documented in the Project VBA help, as are a number of examples of how you might use them. Unfortunately, Application events require you to do a larger amount of coding than Project events do. This section provides details on how to set up a project to use Application events. It uses the `ProjectBeforeTaskDelete` event as an example of how these events can be used.

GETTING HELP

Project has well-developed help documentation for VBA, but unfortunately it can be difficult to find. This is due to two things. First, the Project specific help is not installed in the typical Project installation. When you are installing Project, be sure to include help as part of a custom installation.

The second problem is that the VBE interface does not easily handle both the general Visual Basic help and the Project-specific Visual Basic help. When you select Help from the Help menu, the general Visual Basic help displays and you do not see the information you need about Project.

The easiest way to get to the Project-specific help is to open the Object Browser window, select an object from the project library, right-click, and select Help. The Help window opens, with the Project-specific help in it. Make sure that the object you select is in the Project library and not in just the standard Visual Basic library. Anything with the name `Project` or `Task` is a good thing to choose.

TIP

You can open Project's help system directly from outside Project. It is called `VBAPJ10.CHM`, and it opens in a window without the functionality of the Answer Wizard or search functions you get when you use it within Project, but it is an easy way to read up on the object model and get a good understanding of Visual Basic programming within Project.

Project's help system discusses the object model in great detail and also provides many useful examples that you can use when you are trying to solve problems. Because the object model is so thoroughly covered in help, the remainder of this chapter focuses on the most commonly used programming structures and objects.