CHAPTER 7

USING CLASS MODULES TO CREATE OBJECTS

Class modules are used to create objects. There are many reasons for you as a developer to create your own objects, including the following:

- To encapsulate VBA and Windows API code to make it transportable and easy to use and reuse, as shown in Chapter 12, "Understanding and Using Windows API Calls"
- To trap events
- To raise events
- To create your own objects and object models

In this chapter, we assume you are already familiar with writing VBA code to manipulate the objects in Excel and are familiar with the Excel object model that defines the relationships among those objects. We also assume you are familiar with object properties, methods, and events. If you have written code in the ThisWorkbook module, any of the modules behind worksheets or charts, or the module associated with a UserForm, you have already worked with class modules. One of the key features of these modules, like all class modules, is the ability to trap and respond to events.

The goal of this chapter is to show you how to create your own objects. We begin by explaining how to create a single custom object and then show how you can create a collection containing multiple instances of the object. We continue with a demonstration of how to trap and raise events within your classes.

Creating Objects

Say we want to develop code to analyze a single cell in a worksheet and categorize the entry in that cell as one of the following:

- Empty
- Containing a label
- Containing a constant numeric value
- Containing a formula

This can be readily accomplished by creating a new object with the appropriate properties and methods. Our new object will be a Cell object. It will have an Analyze method that determines the cell type and sets the CellType property to a numeric value that can be used in our code. We will also have a DescriptiveCellType property so we can display the cell type as text.

Listing 7-1 shows the CCell class module code. This class module is used to create a custom Cell object representing the specified cell, analyze the contents of the cell, and return the type of the cell as a user-friendly text string.

Listing 7-1 The CCell Class Module

```
Option Explicit

Public Enum anlCellType
    anlCellTypeEmpty
    anlCellTypeLabel
    anlCellTypeConstant
    anlCellTypeFormula
End Enum

Private muCellType As anlCellType
Private mrngCell As Excel.Range

Property Set Cell(ByRef rngCell As Excel.Range)
    Set mrngCell = rngCell
End Property

Property Get Cell() As Excel.Range
    Set Cell = mrngCell
End Property

Property Get CellType() As anlCellType
```

```
    CellType = muCellType
End Property

Property Get DescriptiveCellType() As String
    Select Case muCellType
        Case anlCellTypeEmpty
            DescriptiveCellType = "Empty"
        Case anlCellTypeFormula
            DescriptiveCellType = "Formula"
        Case anlCellTypeConstant
            DescriptiveCellType = "Constant"
        Case anlCellTypeLabel
            DescriptiveCellType = "Label"
    End Select
End Property

Public Sub Analyze()
    If IsEmpty(mrngCell) Then
        muCellType = anlCellTypeEmpty
    ElseIf mrngCell.HasFormula Then
        muCellType = anlCellTypeFormula
    ElseIf IsNumeric(mrngCell.Formula) Then
        muCellType = anlCellTypeConstant
    Else
        muCellType = anlCellTypeLabel
    End If
End Sub
```

The CCell class module contains a public enumeration with four members, each of which represents a cell type. By default, the enumeration members are assigned values from zero to three. The enumeration member names help make our code more readable and easier to maintain. The enumeration member values are translated into user-friendly text by the DescriptiveCellType property.

NOTE The VBA `IsNumeric` function used in Listing 7-1 considers a label entry such as 123 to be numeric. `IsNumeric` also considers a number entered into a cell formatted as Text to be a number. As both these cell types can be referenced as numeric values in formulas, this has been taken to be the correct result. If you prefer to consider these cells as label entries you can use `WorksheetFunction.IsNumber` instead of `IsNumeric`.

Listing 7-2 shows the AnalyzeActiveCell procedure. This procedure is contained in the standard module MEntryPoints.

Listing 7-2 The AnalyzeActiveCell Procedure

```
Public Sub AnalyzeActiveCell()

    Dim clsCell As CCell

    ' Create new instance of Cell object
    Set clsCell = New CCell

    ' Determine cell type and display it
    Set clsCell.Cell = Application.ActiveCell
    clsCell.Analyze
    MsgBox clsCell.DescriptiveCellType

End Sub
```

If you select a cell on a worksheet and run the AnalyzeActiveCell procedure it creates a new instance of the CCell class that it stores in the clsCell object variable. The procedure then assigns the active cell to the Cell property of this Cell object, executes its Analyze method, and displays the result of its DescriptiveCellType property. This code is contained in the Analysis1.xls workbook in the \Concepts\Ch07 – Using Class Modules to Create Objects folder on the CD that accompanies this book.

Class Module Structure

A class module can be thought of as a template for an object. It defines the methods and properties of the object. Any public subroutines or functions in the class module become methods of the object, and any public variables or property procedures become properties of the object. You can use the class module to create as many instances of the object as you require.

Property Procedures

Rather than rely on public variables to define properties it is better practice to use property procedures. These give you more control over how properties are assigned values and how they return values. Property procedures allow you to validate the data passed to the object and to perform related actions where appropriate. They also enable you to make properties read-only or write-only if you want.

The CCell class uses two private module-level variables to store its properties internally. muCellType holds the cell type in the form of an `anlCellType` enumeration member value. mrngCell holds a reference to the single-cell Range that an object created from the CCell class will represent.

Property procedures control the interface between these variables and the outside world. Property procedures come in three forms:

- **Property Let—**Used to assign a simple value to a property
- **Property Set—**Used to assign an object reference to a property
- **Property Get—**Used to return the simple value or object reference held by a property to the outside world

The property name presented to the outside world is the same as the name of the property procedure. The CCell class uses `Property Set Cell` to allow you to assign a Range reference to the Cell property of the Cell object. The property procedure stores the reference in the mrngCell variable. This procedure could have a validation check to ensure that only single-cell ranges can be specified. There is a corresponding `Property Get Cell` procedure that allows this property to be read.

The CCell class uses two `Property Get` procedures to return the cell type as an enumeration member value or as descriptive text. These properties are read-only because they have no corresponding `Property Let` procedures.

Methods

The CCell class has one method defined by the Analyze subroutine. It determines the type of data in the cell referred to by the mrngCell variable and assigns the corresponding enumeration member to the muCellType variable. Because it is a subroutine, the Analyze method doesn't return a value to the outside world. If a method is created as a function it can return a value. The Analyze method could be converted to a function that returned the text value associated with the cell type as shown in Listing 7-3.

Listing 7-3 The Analyze Method of the Cell Object

```
Public Function Analyze() As String

    If IsEmpty(mrngCell) Then
        muCellType = anlCellTypeEmpty
    ElseIf mrngCell.HasFormula Then
        muCellType = anlCellTypeFormula
    ElseIf IsNumeric(mrngCell.Formula) Then
        muCellType = anlCellTypeConstant
    Else
        muCellType = anlCellTypeLabel
    End If

    Analyze = Me.DescriptiveCellType

End Function
```

You could then analyze the cell and display the return value with the following single line of code instead of the original two lines:

```
MsgBox clsCell.Analyze()
```

Creating a Collection

Now that we have a Cell object we want to create many instances of the object so we can analyze a worksheet or ranges of cells within a worksheet. The easiest way to manage these new objects is to store them in a collection. VBA provides a Collection object that you can use to store objects and data. The Collection object has four methods:

- Add
- Count
- Item
- Remove

There is no restriction on the type of data that can be stored within a Collection object, and items with different data types can be stored in the same Collection object. In our case, we want to be consistent and store just Cell objects in our collection.

To create a new Collection, the first step is to add a new standard module to contain global variables. This module will be called MGlobals. Next, add the following variable declaration to the MGlobals module to declare a global Collection object variable to hold the collection, as follows:

```
Public gcolCells As Collection
```

Now add the CreateCellsCollection procedure shown in Listing 7-4 to the MEntryPoints module. The modified code is contained in the Analysis2.xls workbook in the *\Concepts\Ch07 – Using Class Modules to Create Objects* folder on the CD that accompanies this book.

Listing 7-4 Creating a Collection of Cell Objects

```
Public Sub CreateCellsCollection()

    Dim clsCell As CCell
    Dim rngCell As Range

    ' Create new Cells collection
    Set gcolCells = New Collection

    ' Create Cell objects for each cell in Selection
    For Each rngCell In Application.Selection
        Set clsCell = New CCell
        Set clsCell.Cell = rngCell
        clsCell.Analyze
        'Add the Cell to the collection
        gcolCells.Add Item:=clsCell, Key:=rngCell.Address
    Next rngCell

    ' Display the number of Cell objects stored
    MsgBox "Number of cells stored: " & CStr(gcolCells.Count)

End Sub
```

We declare gcolCells as a public object variable so that it persists for as long as the workbook is open and is visible to all procedures in the VBA project. The CreateCellsCollection procedure creates a new instance of the collection and loops through the currently selected cells, creating a new instance of the Cell object for each cell and adding it to the collection. The address of each cell, in A1 reference style, is used as a key to uniquely identify it and to provide a way of accessing the Cell object later.

We can loop through the objects in the collection using a `For...Each` loop or we can access individual Cell objects by their position in the collection or by using the key value. Because the `Item` method is the default method for the collection, we can use code like the following to access a specific Cell object:

```
Set clsCell = gcolCells(3)
Set clsCell = gcolCells("$A$3")
```

Creating a Collection Object

The collection we have established is easy to use, but it lacks some features we would like to have. As it stands, there is no control over the type of objects that can be added to the collection. We would also like to add a method to the collection that enables us to highlight cells of the same type and another method to remove the highlights.

We first add two new methods to the CCell class module. The Highlight method adds color to the Cell object according to the CellType. The UnHighlight method removes the color. The new code is shown in Listing 7-5.

Note that we are applying the principle of encapsulation. All the code that relates to the Cell object is contained in the CCell class module, not in any other module. Doing this ensures that the code can be easily found and maintained and means that it can be easily transported from one project to another.

Listing 7-5 New Code for the CCell Class Module

```
Public Sub Highlight()
  Cell.Interior.ColorIndex = Choose(muCellType + 1, 5, 6, 7, 8)
End Sub

Public Sub UnHighlight()
  Cell.Interior.ColorIndex = xlNone
End Sub
```

We can now create a new class module named CCells to contain the Cells collection, as shown in Listing 7-6. The complete code is contained in the Analysis3.xls workbook in the *Concepts**Ch07 – Using Class Modules to Create Objects* folder on the CD that accompanies this book.

Listing 7-6 The CCells ClassModule

```
Option Explicit

Private mcolCells As Collection

Property Get Count() As Long
    Count = mcolCells.Count
End Property

Property Get Item(ByVal vID As Variant) As CCell
   Set Item = mcolCells(vID)
End Property

Private Sub Class_Initialize()
    Set mcolCells = New Collection
End Sub

Public Sub Add(ByRef rngCell As Range)
    Dim clsCell As CCell
    Set clsCell = New CCell
    Set clsCell.Cell = rngCell
    clsCell.Analyze
    mcolCells.Add Item:=clsCell, Key:=rngCell.Address
End Sub

Public Sub Highlight(ByVal uCellType As anlCellType)
    Dim clsCell As CCell
    For Each clsCell In mcolCells
        If clsCell.CellType = uCellType Then
            clsCell.Highlight
        End If
    Next clsCell
End Sub

Public Sub UnHighlight(ByVal uCellType As anlCellType)
    Dim clsCell As CCell
    For Each clsCell In mcolCells
        If clsCell.CellType = uCellType Then
            clsCell.UnHighlight
        End If
    Next clsCell
End Sub
```

The mcolCells Collection object variable is declared as a private, module-level variable and is instantiated in the Initialize procedure of the class module. Since the Collection object is now hidden from the outside world, we need to write our own Add method for it. We also have created Item and Count property procedures to emulate the corresponding properties of the collection. The input argument for the Item property is declared as a Variant data type because it can be either a numeric index or the string key that identifies the collection member.

The Highlight method loops through each member of the collection. If the CellType property of the Cell object is the same as the type specified by the uCellType argument, we execute the Cell object's Highlight method. The UnHighlight method loops through the collection and executes the UnHighlight method of all Cell objects whose type is the same as the type specified by the uCellType argument.

We modified the public Collection variable declaration in MGlobals to refer to our new custom collection class as shown here:

```
Public gclsCells As CCells
```

We also modified the CreateCellsCollection procedure in the MEntryPoints module to instantiate and populate our custom collection, as shown in Listing 7-7.

Listing 7-7 MEntryPoints Code to Create a Cells Object Collection

```
Public Sub CreateCellsCollection()

    Dim clsCell As CCell
    Dim lIndex As Long
    Dim lCount As Long
    Dim rngCell As Range

    Set gclsCells = New CCells

    For Each rngCell In Application.ActiveSheet.UsedRange
        gclsCells.Add rngCell
    Next rngCell

    ' Count the number of formula cells in the collection.
    For lIndex = 1 To gclsCells.Count
        If gclsCells.Item(lIndex).CellType = anlCellTypeFormula Then
            lCount = lCount + 1
        End If
    Next lIndex

    MsgBox "Number of Formulas = " & CStr(lCount)

End Sub
```

We declare gclsCells as a public object variable to contain our custom Cells collection object. The CreateCellsCollection procedure instantiates gclsCells and uses a `For...Each` loop to add all the cells in the active worksheet's used range to the collection. After loading the collection, the procedure counts the number of cells that contain formulas and displays the result.

The MEntryPoints module contains a ShowFormulas procedure that can be executed to highlight and unhighlight the formula cells in the worksheet. Several additional variations are provided for other cell types.

This code illustrates two shortcomings of our custom collection class. You can't process the members of the collection in a `For...Each` loop. You must use an index and the Item property instead. Also, our collection has no default property, so you can't shortcut the Item property using the standard collection syntax gclsCells(1) to access a member of the collection. You must specify the Item property explicitly in your code. We explain how to solve these problems using Visual Basic 6 or just a text editor in the next section.

Addressing Class Collection Shortcomings

It is possible to make your custom collection class behave like a built-in collection. It requires nothing more than a text editor to make the adjustments, but first we'll explain how to do it by setting procedure attributes using Visual Basic 6 (VB6) to better illustrate the nature of the changes required.

Using Visual Basic 6

In VB6, unlike Visual Basic for Applications used in Excel, you can specify a property to be the default property of the class. If you declare the Item property to be the default property, you can omit .Item when referencing a member of the collection and use a shortcut such as gclsCells(1) instead.

If you have VB6 installed you can export the code module CCells to a file and open that file in VB6. Place your cursor anywhere within the Item property procedure and select *Tools > Procedure Attributes* from the menu to display the Procedure Attributes dialog. Next, click the *Advanced* >> button and under the Advanced options select (Default) from the *Procedure ID* combo box. This makes the Item property the default property for the class.

When you save your changes and import this file back into your Excel VBA project, the attribute will be recognized even though there is no way to set attribute options within the Excel Visual Basic Editor. VB6 also allows you to set up the special procedure shown in Listing 7-8.

Listing 7-8 Code to Allow the Collection to Be Referenced in a For...Each Loop

```
Public Function NewEnum() As IUnknown
    Set NewEnum = mcolCells.[_NewEnum]
End Function
```

This procedure must be given an attribute value of 4, which you enter directly into the *Procedure ID* combo box in the Procedure Attributes dialog. Giving the `NewEnum` procedure this attribute value enables a `For...Each` loop to process the members of the collection. Once you have made this addition to your class module in VB6 and saved your changes, you can load the module back into your Excel VBA project, and once again the changes will be recognized.

Using a Text Editor

Even without VB6 you can easily create these procedures and their attributes using a text editor such as NotePad. Export the CCells class module to a file and open it using the text editor. Modify your code to look like the example shown in Listing 7-9.

Listing 7-9 Viewing the Code in a Text Editor

```
Property Get Item(ByVal vID As Variant) As CCell
Attribute Item.VB_UserMemId = 0
    Set Item = mcolCells(vID)
End Property

Public Function NewEnum() As IUnknown
Attribute NewEnum.VB_UserMemId = -4
    Set NewEnum = mcolCells.[_NewEnum]
End Function
```

When the modified class module is imported back into your project the Attribute lines will not be visible, but the procedures will work as expected. You can now refer to a member of the collection as gclsCells(1) and use your custom collection class in a `For...Each` loop as shown in Listing 7-10.

Listing 7-10 Referencing the Cells Collection in a For...Each Loop

```
For Each clsCell In gclsCells
     If clsCell.CellType = anlCellTypeFormula Then
          lCount = lCount + 1
     End If
Next clsCell
```

Trapping Events

A powerful capability built into class modules is the ability to respond to events. We want to extend our Analysis application so that when you double-click a cell that has been analyzed it will change color to indicate the cell type. When you right-click the cell the color will be removed. We also want to ensure that cells are reanalyzed when they are changed so that our corresponding Cell objects are kept up-to-date. The code shown in this section is contained in the Analysis4.xls workbook in the *Concepts**Ch07 – Using Class Modules to Create Objects* folder on the CD that accompanies this book. To trap the events associated with an object you need to do two things:

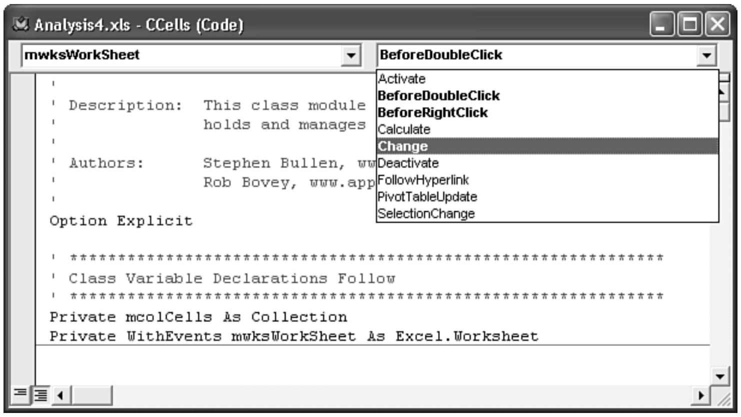
- Declare a WithEvents variable of the correct object type in a class module.
- Assign an object reference to the variable.

For the purpose of this example we confine ourselves to trapping events associated with a single Worksheet object. You could easily substitute this with a Workbook object if you wanted the code to apply to all the worksheets in a workbook. We need to create a WithEvents object variable in the CCells class module that references the worksheet containing the Cell objects. This WithEvents variable declaration is made at the module level within the CCells class and looks like the following:

```
Private WithEvents mwksWorkSheet As Excel.Worksheet
```

As soon as you add this variable declaration to the CCells class module you can select the WithEvents variable name from the drop-down menu at the top left of the module and use the drop-down menu at the top right of the module to see the events that can be trapped, as shown in Figure 7-1.

Event names listed in bold are currently being trapped within the class, as we see in a moment.



```
'
' Description:  This class module
'               holds and manages
'
' Authors:      Stephen Bullen, ww
'               Rob Bovey, www.app
'
Option Explicit

' ****************************************************************
' Class Variable Declarations Follow
' ****************************************************************
Private mcolCells As Collection
Private WithEvents mwksWorkSheet As Excel.Worksheet
```

FIGURE 7-1 The Worksheet event procedures available in CCells

Selecting an event from the drop-down creates a shell for the event procedure in the module. You need to add the procedures shown in Listing 7-11 to the CCells class module. They include a new property named Worksheet that refers to the Worksheet object containing the Cell objects held by the collection, as well as the code for the BeforeDoubleClick, BeforeRightClick, and Change events.

Listing 7-11 Additions to the CCells Class Module

```
Property Set Worksheet(wks As Excel.Worksheet)
    Set mwksWorkSheet = wks
End Property


Private Sub mwksWorkSheet_BeforeDoubleClick( _
            ByVal Target As Range, Cancel As Boolean)
    If Not Application.Intersect(Target, _
            mwksWorkSheet.UsedRange) Is Nothing Then
        Highlight mcolCells(Target.Address).CellType
        Cancel = True
    End If
End Sub
```

```
Private Sub mwksWorkSheet_BeforeRightClick( _
            ByVal Target As Range, Cancel As Boolean)
    If Not Application.Intersect(Target, _
            mwksWorkSheet.UsedRange) Is Nothing Then
        UnHighlight mcolCells(Target.Address).CellType
        Cancel = True
    End If
End Sub

Private Sub mwksWorkSheet_Change(ByVal Target As Range)
    Dim rngCell As Range
    If Not Application.Intersect(Target, _
            mwksWorkSheet.UsedRange) Is Nothing Then
        For Each rngCell In Target.Cells
            mcolCells(rngCell.Address).Analyze
        Next rngCell
    End If
End Sub
```

The CreateCellsCollection procedure in the MEntryPoints module needs to be changed as shown in Listing 7-12. The new code assigns a reference to the active worksheet to the Worksheet property of the Cells object so the worksheet's events can be trapped.

Listing 7-12 The Updated CreateCellsCollection Procedure in the MEntryPoints Module

```
Public Sub CreateCellsCollection()

    Dim clsCell As CCell
    Dim rngCell As Range

    Set gclsCells = New CCells
    Set gclsCells.Worksheet = ActiveSheet

    For Each rngCell In ActiveSheet.UsedRange
        gclsCells.Add rngCell
    Next rngCell

End Sub
```

You can now execute the CreateCellsCollection procedure in the MEntryPoints module to create a new collection with all the links in place to trap the BeforeDoubleClick and BeforeRightClick events for the cells in the worksheet. Double-clicking a cell changes the cell's background to a color that depends on the cell's type. Right-clicking a cell removes the background color.

Raising Events

Another powerful capability of class modules is the ability to raise events. You can define your own events and trigger them in your code. Other class modules can trap those events and respond to them. To illustrate this we change the way our Cells collection tells the Cell objects it contains to execute their Highlight and UnHighlight methods. The Cells collection raises an event that will be trapped by the Cell objects. The code shown in this section is contained in the Analysis5.xls workbook in the *Concepts\Ch07 – Using Class Modules to Create Objects* folder on the CD that accompanies this book. To raise an event in a class module you need two things.

- An Event declaration at the top of the class module
- A line of code that uses RaiseEvent to cause the event to take place

The code changes shown in Listing 7-13 should be made in the CCells class module.

Listing 7-13 Changes to the CCells Class Module to Raise an Event

```
Option Explicit

Public Enum anlCellType
    anlCellTypeEmpty
    anlCellTypeLabel
    anlCellTypeConstant
    anlCellTypeFormula
End Enum

Private mcolCells As Collection
Private WithEvents mwksWorkSheet As Excel.Worksheet

Event ChangeColor(uCellType As anlCellType, bColorOn As Boolean)

Public Sub Add(ByRef rngCell As Range)
    Dim clsCell As CCell
    Set clsCell = New CCell
    Set clsCell.Cell = rngCell
    Set clsCell.Parent = Me
    clsCell.Analyze
    mcolCells.Add Item:=clsCell, Key:=rngCell.Address
End Sub

Private Sub mwksWorkSheet_BeforeDoubleClick( _
            ByVal Target As Range, Cancel As Boolean)
    If Not Application.Intersect(Target, _
            mwksWorkSheet.UsedRange) Is Nothing Then
        RaiseEvent ChangeColor( _
            mcolCells(Target.Address).CellType, True)
        Cancel = True
    End If
End Sub

Private Sub mwksWorkSheet_BeforeRightClick( _
             ByVal Target As Range, Cancel As Boolean)
    If Not Application.Intersect(Target, _
             mwksWorkSheet.UsedRange) Is Nothing Then
        RaiseEvent ChangeColor( _
            mcolCells(Target.Address).CellType, False)
        Cancel = True
    End If
End Sub
```

Note that we moved the `anlCellType` Enum declaration into the parent collection class module. Now that we have created an explicit parent-child relationship between the CCells and CCell classes, any public types used by both classes must reside in the parent class module or circular dependencies between the classes that cannot be handled by VBA will be created.

In the declarations section of the CCells module, we declare an event named ChangeColor that has two arguments. The first argument defines the cell type to be changed, and the second argument is a Boolean value to indicate whether we are turning color on or off. The BeforeDoubleClick and BeforeRightClick event procedures have been changed to raise the new event and pass the cell type of the target cell and the on or off value. The Add method has been updated to set a new Parent property of the Cell object. This property holds a reference to the Cells object. The name reflects the relationship between the Cells object as the parent object and the Cell object as the child object.

Trapping the event raised by the Cells object in another class module is carried out in exactly the same way we trapped other events. We create a WithEvents object variable and set it to reference an instance of the class that defines and raises the event. The changes shown in Listing 7-14 should be made to the CCell class module.

Listing 7-14 Changes to the CCell Class Module to Trap the ChangeColor Event

```
Option Explicit

Private muCellType As anlCellType
Private mrngCell As Excel.Range
Private WithEvents mclsParent As CCells

Property Set Parent(ByRef clsCells As CCells)
    Set mclsParent = clsCells
End Property

Private Sub mclsParent_ChangeColor(uCellType As anlCellType, _
                                          bColorOn As Boolean)
    If Me.CellType = uCellType Then
        If bColorOn Then
            Highlight
        Else
            UnHighlight
        End If
    End If
End Sub
```

A new module-level object variable mclsParent is declared WithEvents as an instance of the CCells class. A reference to a Cells object is assigned to mclsParent in the Parent Property Set procedure. When the Cells object raises the ChangeColor event, all the Cell objects will trap it. The Cell objects take action in response to the event if they are of the correct cell type.

A Family Relationship Problem

Unfortunately, we introduced a problem in our application. Running the CreateCellsCollection procedure multiple times creates a memory leak. Normally when you overwrite an object in VBA, VBA cleans up the old version of the object and reclaims the memory that was used to hold it. You can also set an object equal to Nothing to reclaim the memory used by it. It is good practice to do this explicitly when you no longer need an object, rather than relying on VBA to do it.

```
Set gclsCells = Nothing
```

When you create two objects that store references to each other, the system will no longer reclaim the memory they used when they are set to new versions or when they are set to Nothing. When analyzing the worksheet in Analysis5.xls with 574 cells in the used range, there is a loss of about 250KB of RAM each time CreateCellsCollection is executed during an Excel session.

NOTE If you are running Windows NT, 2000, XP, or Vista you can check the amount of RAM currently used by Excel by pressing Ctrl+Shift+Esc to display the Processes window in Task Manager and examining the memory usage column for the row where the Image Name column is EXCEL.EXE.

One way to avoid this problem is to make sure you remove the cross-references from the linked objects before the objects are removed. You can do this by adding a method such as the Terminate method shown in Listing 7-15 to the problem classes, in our case the CCell class.

Listing 7-15 The Terminate Method in the CCell Class Module

```
Public Sub Terminate()
    Set mclsParent = Nothing
End Sub
```

The code in Listing 7-16 is added to the CCells class module. It calls the Terminate method of each Cell class contained in the collection to destroy the cross-reference between the classes.

Listing 7-16 The Terminate Method in the CCells Class Module

```
Public Sub Terminate()
    Dim clsCell As CCell
    For Each clsCell In mcolCells
```

```
        clsCell.Terminate
        Set clsCell = Nothing
    Next clsCell
    Set mcolCells = Nothing
End Sub
```

The code in Listing 7-17 is added to the CreateCellsCollection procedure in the MEntryPoints module.

Listing 7-17 The CreateCellsCollection Procedure in the MEntryPoints Module

```
Public Sub CreateCellsCollection()
    Dim clsCell As CCell
    Dim rngCell As Range

    ' Remove any existing instance of the Cells collection
    If Not gclsCells Is Nothing Then
        gclsCells.Terminate
        Set gclsCells = Nothing
    End If

    Set gclsCells = New CCells
    Set gclsCells.Worksheet = ActiveSheet

    For Each rngCell In ActiveSheet.UsedRange
        gclsCells.Add rngCell
    Next rngCell

End Sub
```

If CreateCellsCollection finds an existing instance of gclsCells it executes the object's Terminate method before setting the object to Nothing. The gclsCells Terminate method iterates through all the objects in the collection and executes their Terminate methods.

In a more complex object model with more levels you could have objects in the middle of the structure that contain both child and parent references. The Terminate method in these objects would need to run the Terminate method of each of its children and then set its own Parent property to Nothing.

Creating a Trigger Class

Instead of raising the ChangeColor event in the CCells class module we can set up a new class module to trigger this event. Creating a trigger class gives us the opportunity to introduce a more efficient way to highlight our Cell objects. We can create four instances of the trigger class, one for each cell type, and assign the appropriate instance to each Cell object. That means each Cell object is only sent a message that is meant for it, rather than hearing all messages sent to all Cell objects.

The trigger class also enables us to eliminate the Parent/Child relationship between our CCells and CCell classes, thus removing the requirement to manage cross-references. Note that it is not always possible or desirable to do this. The code shown in this section is contained in the Analysis6.xls workbook in the *Concepts**Ch07 – Using Class Modules to Create Objects* folder on the CD that accompanies this book.

Listing 7-18 shows the code in a new CTypeTrigger class module. The code declares the ChangeColor event, which now only needs one argument to specify whether color is turned on or off. The class has Highlight and UnHighlight methods to raise the event.

Listing 7-18 The CTypeTrigger Class Module

```
Option Explicit

Public Event ChangeColor(bColorOn As Boolean)

Public Sub Highlight()
    RaiseEvent ChangeColor(True)
End Sub

Public Sub UnHighlight()
    RaiseEvent ChangeColor(False)
End Sub
```

Listing 7-19 contains the changes to the CCell class module to trap the ChangeColor event raised in CTypeTrigger. Depending on the value of bColorOn, the event procedure runs the Highlight or UnHighlight methods.

Listing 7-19 Changes to the CCell Class Module to Trap the ChangeColor Event of CTypeTrigger

```
Option Explicit

Private muCellType As anlCellType
Private mrngCell As Excel.Range
Private WithEvents mclsTypeTrigger As CTypeTrigger

Property Set TypeTrigger(clsTrigger As CTypeTrigger)
    Set mclsTypeTrigger = clsTrigger
End Property

Private Sub mclsTypeTrigger_ChangeColor(bColorOn As Boolean)
    If bColorOn Then
        Highlight
    Else
        UnHighlight
    End If
End Sub
```

Listing 7-20 contains the changes to the CCells module. An array variable maclsTriggers is declared to hold the instances of CTypeTrigger. The Initialize event redimensions maclsTriggers to match the number of cell types and the `For...Each` loop assigns instances of CTypeTrigger to the array elements. The Add method assigns the correct element of maclsTriggers to each Cell object according to its cell type. The result is that each Cell object listens only for messages that apply to its own cell type.

Listing 7-20 Changes to the CCells Class Module to Assign References to CTypeTrigger to Cell Objects

```
Option Explicit

Public Enum anlCellType
    anlCellTypeEmpty
    anlCellTypeLabel
    anlCellTypeConstant
    anlCellTypeFormula
End Enum

Private mcolCells As Collection
Private WithEvents mwksWorkSheet As Excel.Worksheet
Private maclsTriggers() As CTypeTrigger

Private Sub Class_Initialize()
    Dim uCellType As anlCellType
    Set mcolCells = New Collection
    ' Initialise the array of cell type triggers,
    ' one element for each of our cell types.
    ReDim maclsTriggers(anlCellTypeEmpty To anlCellTypeFormula)
    For uCellType = anlCellTypeEmpty To anlCellTypeFormula
        Set maclsTriggers(uCellType) = New CTypeTrigger
    Next uCellType
End Sub

Public Sub Add(ByRef rngCell As Range)
    Dim clsCell As CCell
    Set clsCell = New CCell
    Set clsCell.Cell = rngCell
    clsCell.Analyze
    Set clsCell.TypeTrigger = maclsTriggers(clsCell.CellType)
    mcolCells.Add Item:=clsCell, Key:=rngCell.Address
End Sub

Public Sub Highlight(ByVal uCellType As anlCellType)
    maclsTriggers(uCellType).Highlight
End Sub

Public Sub UnHighlight(ByVal uCellType As anlCellType)
    maclsTriggers(uCellType).UnHighlight
End Sub

Private Sub mwksWorkSheet_BeforeDoubleClick( _
            ByVal Target As Range, Cancel As Boolean)
    If Not Application.Intersect(Target, _
            mwksWorkSheet.UsedRange) Is Nothing Then
        Highlight mcolCells(Target.Address).CellType
        Cancel = True
    End If
End Sub

Private Sub mwksWorkSheet_BeforeRightClick( _
            ByVal Target As Range, Cancel As Boolean)
    If Not Application.Intersect(Target, _
            mwksWorkSheet.UsedRange) Is Nothing Then
        UnHighlight mcolCells(Target.Address).CellType
        Cancel = True
    End If
End Sub

Private Sub mwksWorkSheet_Change(ByVal Target As Range)

    Dim rngCell As Range
    Dim clsCell As CCell

    If Not Application.Intersect(Target, _
            mwksWorkSheet.UsedRange) Is Nothing Then
        For Each rngCell In Target.Cells
            Set clsCell = mcolCells(rngCell.Address)
            clsCell.Analyze
            Set clsCell.TypeTrigger = _
                maclsTriggers(clsCell.CellType)
        Next rngCell
    End If

End Sub
```

Practical Example

We illustrate the use of class modules in our PETRAS example applications by providing both the Time Sheet and Reporting applications with Excel application-level event handlers.

PETRAS Time Sheet

The addition of an application-level event handling class to our PETRAS time sheet application will make two significant changes. First, it will allow us to convert the time entry workbook into an Excel template. This will simplify creation of new time entry workbooks for new purposes as well as allow multiple time entry workbooks to be open at the same time. Second, the event handler will automatically detect whether a time entry workbook is active and enable or disable our toolbar buttons accordingly. Table 7-1 summarizes the changes made to the PETRAS time sheet application for this chapter.

Table 7-1 Changes to PETRAS Time Sheet Application for Chapter 7

Module	Procedure	Change
PetrasTemplate.xlt		Changes the normal workbook into a template workbook
CAppEventHandler		Adds an application-level event handling class to the add-in
MEntryPoints	NewTimeSheet	New procedure to create time sheets from the template workbook
MopenClose	Auto_Open	Removes time sheet initialization logic and delegates it to the event handling class
MsystemCode		Moves all time entry workbook management code into the event handling class

The Template

When a template workbook is added using VBA, a new, unsaved copy of the template workbook is opened. To create a template workbook from a normal workbook, choose *File > Save As* from the Excel menu and select the Template entry from the *Save as type* drop-down. As soon as you select the Template option Excel unhelpfully modifies the directory where you are saving your workbook to the Office Templates directory, so don't forget to change this to the location where you are storing your application files.

Once we begin using a template workbook, the user has complete control over the workbook filename. We can determine whether a given workbook belongs to us by checking for the unique named constant "setIsTimeSheet" that we added to our template workbook for this purpose.

A template workbook combined with an application-level event handler allows us to support multiple instances of the time entry workbook being open simultaneously. This might be needed, for example, if there is a requirement to have a separate time sheet for each client or project.

Moving to a template user interface workbook also requires that we give the user a way to create new time sheet workbooks, since it is no longer a simple matter of opening and reusing the same fixed time sheet workbook over and over. In Figure 7-2, note the new toolbar button labeled *New Time Sheet*. This button allows the user to create new instances of our template.

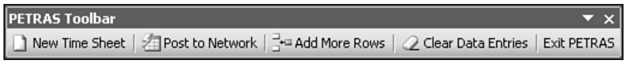


FIGURE 7-2 The PETRAS toolbar with the New Time Sheet button

As shown in Listing 7-21, the code run by this new button is simple.

Listing 7-21 The NewTimeSheet Procedure

```
Public Sub NewTimeSheet()
    Application.ScreenUpdating = False
    InitGlobals
    Application.Workbooks.Add gsAppDir & gsFILE_TIME_ENTRY
    Application.ScreenUpdating = True
End Sub
```

We turn off screen updating and call InitGlobals to ensure that our global variables are properly initialized. We then simply add a new workbook based on the template workbook and turn screen updating back on. Rather than opening PetrasTemplate.xlt, a new copy of PetrasTemplate.xlt, called PetrasTemplate1 is created. Each time the user clicks the New Time Sheet button she gets a completely new, independent copy of PetrasTemplate.xlt.

The act of creating a new copy of the template triggers the NewWorkbook event in our event handing class. This event performs all the necessary actions to initialize the template. This event procedure is shown in the next section.

The Application-Level Event Handler

Within our application-level event handling class we encapsulate many of the tasks previously accomplished by procedures in standard modules. For example, the MakeWorksheetSettings procedure and the bIsTimeEntryBookActive function that we encountered in Chapter 5, "Function, General, and Application-Specific Add-ins," are now both private procedures of the class.

We describe the layout of the class module and then explain what the pieces do, rather than showing all the code here. You can examine the code yourself in the PetrasAddin.xla workbook of the sample application for this chapter on the CD and are strongly encouraged to do so.

Module-Level Variables

Private WithEvents mxlApp As Excel.Application

Class Event Procedures

Class_Initialize
Class_Terminate
mxlApp_NewWorkbook
mxlApp_WorkbookOpen
mxlApp_WindowActivate
mxlApp_WindowDeactivate

Class Method Procedures

SetInitialStatus

Class Private Procedures

EnableDisableToolbar
MakeWorksheetSettings
bIsTimeEntryBookActive
bIsTimeEntryWorkbook

Because the variable that holds a reference to the instance of the CAppEventHandler class that we use in our application is a public variable, we use the InitGlobals procedure to manage it. The code required to do this is shown in two locations.

In the declarations section of the MGlobals module:

```
Public gclsEventHandler As CAppEventHandler
```

In the InitGlobals procedure:

```
' Instantiate the Application event handler
If gclsEventHandler Is Nothing Then
    Set gclsEventHandler = New CAppEventHandler
End If
```

The InitGlobals code checks to see whether the public gclsEventHandler variable is initialized and initializes it if it isn't.

InitGlobals is called at the beginning of every non-trivial entry point procedure in our application, so if anything causes our class variable to lose state, it will be instantiated again as soon as the next entry point procedure is called. This is a good safety mechanism.

When the public gclsEventHandler variable is initialized, it causes the Class_Initialize event procedure to execute. Inside this event procedure we initialize the event handling mechanism by setting the class module-level WithEvents variable to refer to the current instance of the Excel Application, as follows:

```
Set mxlApp = Excel.Application
```

Similarly, when our application is exiting and we destroy our gclsEventHandler variable, it causes the Class_Terminate event procedure to execute. Within this event procedure we destroy the class reference to the Excel Application object by setting the mxlApp variable to Nothing.

All the rest of the class event procedures, which are those belonging to the mxlApp WithEvents variable, serve the same purpose. They "watch" the Excel environment and enable or disable our toolbar buttons as appropriate when conditions change.

Disabling toolbar buttons when they can't be used is a much better user interface technique than displaying an error message when the user clicks one under the wrong circumstances. You don't want to punish users (that is, display an error message in response to an action) when they can't be expected to know they've done something wrong. Note that we always leave the *New Time Sheet* and *Exit PETRAS* toolbar buttons enabled. Users should always be able to create a new time sheet or exit the application.

In addition to enabling and disabling the toolbar buttons, the mxlApp_NewWorkbook and mxlApp_WorkbookOpen event procedures detect when a time entry workbook is being created or opened for the first time, respectively. At this point they run the private MakeWorksheetSettings procedure to initialize that time entry workbook. All the mxlApp event procedures are shown in Listing 7-22. As you can see, the individual procedures are simple, but the cumulative effect is powerful.

Listing 7-22 The mxlApp Event Procedures

```
Private Sub mxlApp_NewWorkbook(ByVal Wb As Workbook)
    If bIsTimeEntryWorkbook(Wb) Then
        EnableDisableToolbar True
        MakeWorksheetSettings Wb
    Else
        EnableDisableToolbar False
    End If
End Sub

Private Sub mxlApp_WorkbookOpen(ByVal Wb As Excel.Workbook)
    If bIsTimeEntryWorkbook(Wb) Then
        EnableDisableToolbar True
        MakeWorksheetSettings Wb
    Else
        EnableDisableToolbar False
    End If
End Sub

Private Sub mxlApp_WindowActivate(ByVal Wb As Workbook, _
                                    ByVal Wn As Window)
    ' When a window is activated, check to see if it belongs
    ' to one of our workbooks. Enable all our toolbar controls
    ' if it does.
    EnableDisableToolbar bIsTimeEntryBookActive()
End Sub

Private Sub mxlApp_WindowDeactivate(ByVal Wb As Workbook, _
                                    ByVal Wn As Window)
    ' When a window is deactivated, disable our toolbar
    ' controls by default. They will be re-enables by the
    ' WindowActivate event procedure if required.
    EnableDisableToolbar False
End Sub
```

The full power of having an event handling class in your application is difficult to convey on paper. We urge you to experiment with the sample application for this chapter to see for yourself how it works in a live setting. Double-click the PetrasAddin.xla file to open Excel and see how the application toolbar behaves. Create new time sheet workbooks, open non-time sheet workbooks, and switch back and forth between them. The state of the toolbar will follow your every action.

It is also educational to see exactly how much preparation the application does when you create a new instance of the time sheet workbook. Without the PetrasAddin.xla running, open the PetrasTemplate.xlt workbook and compare how it looks and behaves in its raw state with the way it looks and behaves as an instance of the time sheet within the running application.

PETRAS Reporting

By adding a class module to handle application-level events to the PETRAS Reporting application, we can allow the user to have multiple consolidation workbooks open at the same time and switch between them using the new Window menu, as shown in Figure 7-3.

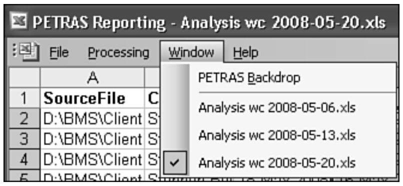

PETRAS Reporting - Analysis wc 2008-05-20.xls
File
Processing
Window
Help
A
C
1 SourceFile
2 D:\BMS\Client
3 D:\BMS\Client
4 D:\BMS\Client
PETRAS Backdrop
Analysis wc 2008-05-06.xls
Analysis wc 2008-05-13.xls
✓ Analysis wc 2008-05-20.xls

FIGURE 7-3 The PETRAS Reporting menu bar with the new Window menu

Table 7-2 summarizes the changes made to the PETRAS time sheet application for this chapter. Rather than repeat much of the previous few pages, we suggest you review the PetrasReporting.xla workbook to see exactly how the multiple-document interface has been implemented.

Table 7-2 Changes to PETRAS Reporting Application for Chapter 7

Module	Procedure	Change
CAppEventHandler		Adds an application-level event handling class to the application to manage multiple consolidation workbooks.
MCommandBars	SetUpMenus	Adds code to create the Window menu.
MSystemCode		Adds procedures to add, remove, and place a tick mark against an item in the Window menu.
MEntryPoints	MenuWindowSelect	New procedure to handle selecting an item within the Window menu. All Window menu items call this routine.

Summary

You use class modules to create objects and their associated methods, properties, and events. You can collect child objects in a parent object so that you can create a hierarchy of objects to form an object model. You can use class modules to trap the events raised by other objects including the Excel application. You can also define and raise your own events in a class module.

When you set up cross-references between parent and child objects so that each is aware of the other you create a structure that is not simple to remove from memory when it is no longer useful. You need to add extra code to remove these cross-references.

Class modules are a powerful addition to a developer's toolkit. The objects created lead to code that is easier to write, develop, maintain, and share than traditional code. Objects are easy to use because they encapsulate complex code in a form that is accessible. All you need to know to use an object are its methods, properties, and events. Objects can be shared because the class modules that define them are encapsulated (self-contained) and therefore transportable from one project to another. All you need to do is copy the class module to make the object available in another project.

As a developer you can easily add new methods, properties, and events to an object without changing the existing interface. Your objects can evolve without harming older systems that use them. Most developers find class modules addictive. The more you use them, the more you like them and the more uses you find for them. They are used extensively throughout the rest of this book.