Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data:
Professional Excel development : the definitive guide to developing applications using Microsoft Excel, VBA, and .NET / Rob Bovey ... [et al.]. — 2nd ed.

p. cm.


HF5548.4.M523B85 2009
005.54—dc22

Copyright © 2009 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax (617) 671-3447


Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.

First printing May 2009
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 portable 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

```vba
Option Explicit

Public Enum anlCellType
    anlCellTypeEmpty
    anlCellTypeLabel
    anlCellTypeConstant
    anlCellTypeFormula
End Enum

Private muCellType As anlCellType
Private mrngCell As Excel.Range

Property Set Cell(ByVal 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
End Property
```
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`. 


Chapter 7  Using Class Modules to Create Objects

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.xlxs 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. mrgCell 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 mrgCell 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 mrgCell 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.
Creating a 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 $A$1 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:

```vba
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.

Listing 7-5  New Code for the CCell Class Module

```vba
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.
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(ByVal 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:

```vbnet
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

```vbnet
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
```
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**

```vbnet
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**

```vbnet
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.
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.

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

```vba
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
```

**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

```vba
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

```vba
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 EXCELEXE.

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**

```vba
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**

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

```vba
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
    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.
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.

<table>
<thead>
<tr>
<th>Module</th>
<th>Procedure</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>PetrasTemplate.xlt</td>
<td></td>
<td>Changes the normal workbook into a template workbook</td>
</tr>
<tr>
<td>CAppEventHandler</td>
<td></td>
<td>Adds an application-level event handling class to the add-in</td>
</tr>
<tr>
<td>MEntryPoint</td>
<td>NewTimeSheet</td>
<td>New procedure to create time sheets from the template workbook</td>
</tr>
<tr>
<td>MopenClose</td>
<td>Auto_Open</td>
<td>Removes time sheet initialization logic and delegates it to the event handling class</td>
</tr>
<tr>
<td>MsystemCode</td>
<td></td>
<td>Moves all time entry workbook management code into the event handling class</td>
</tr>
</tbody>
</table>
As shown in Listing 7-21, the code run by this new button is simple.

Listing 7-21 The NewTimeSheet Procedure

```vba
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:

```vba
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**

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

Chapter 7 Using Class Modules to Create Objects
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 worksheet 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 menu bar with the new Window menu](image)

**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**

<table>
<thead>
<tr>
<th>Module</th>
<th>Procedure</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAppEventHandler</td>
<td></td>
<td>Adds an application-level event handling class to the application to manage multiple consolidation workbooks.</td>
</tr>
<tr>
<td>MCommandBars</td>
<td>SetUpMenus</td>
<td>Adds code to create the Window menu.</td>
</tr>
<tr>
<td>MSystemCode</td>
<td></td>
<td>Adds procedures to add, remove, and place a tick mark against an item in the Window menu.</td>
</tr>
<tr>
<td>MEntryPoints</td>
<td>MenuWindowSelect</td>
<td>New procedure to handle selecting an item within the Window menu. All Window menu items call this routine.</td>
</tr>
</tbody>
</table>
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.
In 2002, Microsoft released the first version of its development suite Visual Studio.NET (VS.NET) together with the .NET Framework. Since then, Microsoft has released new versions of the Framework and development suite in quick succession. Microsoft has strongly indicated that .NET is the flagship development platform now and for the foreseeable future.

Visual Basic.NET (VB.NET) is part of VS.NET, and despite its similarity in the name with Classic VB (VB6), the two have little in common. VB.NET is the successor to Classic VB and as such it provides the ability to create more technically modern solutions, a large group of new and updated controls, and a new advanced IDE. Moving from Classic VB to VB.NET is a non-trivial process, primarily because VB.NET is based on a new and completely different technology platform.

Excel developers also face the situation where applications created with the new .NET technology need to communicate with applications based on the older COM technology, for example, VB.NET applications communicating with Excel. Because Excel is a COM-based application it cannot communicate directly with code written in .NET. All .NET code that communicates with Excel must cross the .NET ➔ COM boundary. This is important to keep in mind because it is a challenge to manage and can have significant performance implications.

In the first part of this chapter, VB.NET is introduced along with the .NET Framework. The second part of this chapter focuses on how we can automate Excel with VB.NET. Finally, we cover ADO.NET, which is used to connect to and retrieve data from various data sources. ADO.NET is the successor to classic ADO on the .NET platform.

To provide a better understanding of VB.NET, we develop a practical solution, the PETRAS Report Tool.NET. This solution is a fully functional Windows Forms based reporting tool. It retrieves data from the PETRAS SQL Server database and uses Excel templates to present the reports.
VB.NET, ADO.NET, and the .NET Framework are book-length topics in their own right; what we examine here and in the two following chapters merely scratches the surface. At the end of this chapter you find some recommended books and online resources that provide additional detail on these subjects.

.NET Framework Fundamentals

The .NET Framework is the core of .NET. Before we can develop or run any .NET-based solutions, the Framework must be installed and available. The Framework provides the foundation for all .NET software development. The .NET Framework is also responsible for interoperability between .NET solutions and COM servers and components. This topic is covered later in the chapter. For the purposes of our discussion, we can think of the .NET Framework architecture as consisting of two major parts:

- **A huge collection of base class libraries and interfaces**—This collection contains all the class libraries and interfaces required for .NET solutions. **Namespaces** are used to organize these class libraries and interfaces into a hierarchical structure. The namespaces are usually organized by function, and each namespace usually has several child namespaces. Namespaces make it easy to access and use different classes and simplify object references. We discuss namespaces in more detail when presenting VB.NET later in this chapter.

- **Common Language Runtime (CLR)**—This is the engine of the .NET Framework, and it is responsible for all .NET base services. It controls and monitors all activities of .NET applications, including memory management, thread management, **structured exception handling (SEH)**, **garbage collection**, and security. It also provides a **common data type system (CTS)** that defines all .NET data types.

The rapid evolution of the .NET Framework is reflected in the large number of versions available. Different Framework versions can coexist on one computer, and multiple versions of the Framework can be run side-by-side simultaneously on the same computer. However, an application can only use one version of the .NET Framework at any one time. The Framework version that becomes active is determined by which version is required by the .NET-based program that is loaded first. A general recommendation is to only have one version of the Framework installed on a target computer.
Because there are several different Framework versions in common use and we may not be able to control the version available on the computers we target, we need to apply the same strategy to the .NET Framework as we do when targeting multiple Excel versions: Develop against the lowest Framework version we plan to target. Of course there will also be situations that dictate the Framework version we need to target, such as corporate clients who have standardized on a specific version.

As of this writing, the two most common Framework versions are 2.0 and 3.0. Both versions can be used on Windows XP, and version 3.0 is included with Windows Vista and Windows Server 2008. Visual Studio 2008 (VS 2008) includes both of these Framework versions as well as version 3.5. By providing all current Framework versions, VS 2008 makes it easy to select the most appropriate version to use when building our solutions. Versions 3.0 and 3.5 of the .NET Framework are backward compatible in a similar manner as the latest versions of the Excel object libraries.

The .NET Framework can run on all versions of Windows from Windows 98 forward, but to develop .NET-based solutions we need to have Windows 2000 or later. If we plan to target Windows XP or earlier we need to make sure the desired version of the .NET Framework is installed on the target computer, because these Windows versions do not include the Framework preinstalled. All versions of the Framework are available for download from the Microsoft Web site and can be redistributed easily. To avoid confusion, we only use version 2.0 of the .NET Framework in this chapter and the next.

NOTE The standard version 3.5 .NET Framework distribution is around 197MB in size. Microsoft provided a lighter edition of about 25MB in size that can be installed on the target computers instead. To find out more about this edition, search for the phrase "NET Framework Client Profile Deployment Guide" at www.microsoft.com.

**Visual Basic.NET**

With VS.NET we can create Web applications, server applications, database applications, console applications, Windows desktop applications, setup and deployment projects, and much more. VS.NET ships with the following programming languages: Visual C#, VB.NET, and Visual C++.

VB.NET is distributed in all VS.NET packages as well as in a standalone version. However, not all capabilities are present in all distributions.
You need to select the version of VB.NET that fits your requirements best. Table 24-1 shows the capabilities related to Excel development and the distributions in which they are available.

### Table 24-1  Available Tools in Different Versions of VS.NET

<table>
<thead>
<tr>
<th>Feature</th>
<th>VB.NET Express</th>
<th>VS.NET Standard</th>
<th>VS.NET Professional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automate Excel</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Shared Add-in Template (To create managed COM add-ins with.)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Office templates</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Visual Studio Tools for Office System (VSTO)</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

For a full comparison among the versions, see http://msdn.microsoft.com/en-us/vs2008/products/cc149003.aspx. If you just want to try out VB.NET you can download the free Express Edition from Microsoft’s Web site. VS.NET Professional is required if you plan to develop managed COM add-ins and VSTO solutions. It is also required to follow the discussions here and in the next two chapters.

VB.NET was the first version of VB that broke backward compatibility badly enough that you could not even open a project created in an earlier version of VB. If you have non-trivial Classic VB projects that you would like to transfer to VB.NET, the best choice is to create them from scratch in VB.NET. Microsoft has some tools to ease the transition, but for larger VB projects they cannot do all the work. On the other hand, you may also consider keeping your Classic VB solutions for as long as it is still possible to run them on the Windows versions your solution targets. VB.NET is the first BASIC language version that fully supports object oriented programming (OOP). It means that with VB.NET we can fully utilize encapsulation, inheritance, and polymorphism.

Code that targets the .NET runtime is described as **managed code** while code that cannot be hosted by the .NET runtime is described as **unmanaged code**. **Assemblies** are the binary units (*.DLL or *.EXE) that contain the managed code. Since it is common that one .NET assembly contains only one binary unit, it is safe to refer to .NET-based DLL files as assemblies.
The Visual Studio IDE

The Visual Studio IDE (VS IDE) is shared by all .NET programming languages. The VS IDE is a complex development environment, even for developers who are very familiar with the Classic VB IDE. Figure 24-1 shows the VS IDE with a simple VB.NET Windows Forms project open.

When you first run VS.NET, you are prompted to select a development category for VS.NET to use in customizing the environment. If your previous experience is with Classic VB or VBA, you will probably want to allow VB.NET to be your first choice of programming language. In this case, choose the Visual Basic Development Settings. The VS IDE is also highly customizable by the user, but before beginning to customize it you should learn the basics using the default configuration.
General Configuration of the VS IDE

After launching the VS IDE, you should change some general configuration settings for the development environment. Start by selecting Tools > Options... from the menu. This displays the Options dialog shown in Figure 24-2.

The Options dialog organizes its settings in a tree view on the left side. The VB Defaults section under Projects and Solutions contains four of the more important settings for VB.NET development. We recommend that you set them exactly as shown in Figure 24-2. A detailed description of each setting follows:

- **Option Explicit**—Determines whether VB.NET requires us to declare all variables before using them.
- **Option Strict**—Turning on this setting disallows late binding (to improve performance), implicit data type conversion, and provides strong typing (strict use of type rules with no exceptions).
- **Option Compare**—Specifies the default method used for string comparisons. It can either be Binary (case-sensitive) or Text (case-insensitive). The default value is Binary, which provides the same text comparison behavior as Classic VB. See Chapter 3, “Excel and VBA Development Best Practices,” for more information.
- **Option Infer**—When this setting is turned on it allows us to omit the data type when declaring a variable and instead let VB.NET
identify (“infer”) the data type. Listing 24-1 shows a simple example. The right-hand value tells the compiler the data type is an Integer. Declaring a variable and giving it a value at the same time in this manner is fully supported in VB.NET.

**Listing 24-1  Omitting the Data Type When Declaring a Variable**

```vbnet
Dim iCountRows = 225
```

When working with **VB.NET solutions** (a solution can contain one or more projects), these settings can be overridden at the code module level. This means, for example, that if we really need to use late binding in one code module we can modify the Option Strict setting at the top of that code module. Listing 24-2 shows how to turn off the Option Strict setting and also change comparisons to Text.

**Listing 24-2  Changing Settings at the Code Module Level**

```vbnet
Option Compare Text
Option Strict Off
```

Adding line numbers to your code can make many development tasks easier, the debugging process in particular. To activate this option, expand the **Text Editor** section in the Options dialog and select the **Basic** section below it. Check the option **Line numbers** and then close the dialog.

Next we make screentips and keyboard shortcuts available in the IDE. Choose **Tools > Customize...** from the menu. This displays the Customize dialog shown in Figure 24-3. Check the two options **Show ScreenTips on toolbars** and **Show shortcut keys in ScreenTips** and then close the dialog.

The final setting is to make various docked windows in the IDE hide themselves when they are not being used. This provides us with a workspace that is not cluttered with open windows not relevant to the current context.

1. Click on the window you want to hide so it gets the focus.
2. On the **Window** menu click on the option **Auto Hide** or click on the pushpin icon on the title bar for the window.
3. Repeat these steps for every window that you want to auto hide.

When an auto-hidden window loses focus, it automatically slides back to its tab on the edge of the IDE.
Creating a VB.NET Solution

We create a new VB.NET project by selecting the File > New Project... from the menu. This displays the New Project dialog shown in Figure 24-4.
Since we are creating a Windows based-solution, select Windows in the Project types section and then select the Windows Forms Application template. We also select the version of .NET Framework to target using the combo box in the upper-right corner. Next, enter the name “First Application” in the Name box. By default, the solution name is the same as the name entered in the Name box, as shown in Figure 24-4. The solution name is also used to name the main folder of the project. Finally, click the OK button to create the solution.

The Solution Explorer window provides the workspace for working with files and projects inside VB.NET solutions. Figure 24-5 shows the workspace for our solution. A single Windows Form has been added to the solution and we have right-clicked on the form to display the shortcut menu containing the various actions available to perform on that object.

Windows Forms are the basic building block of many solutions. They provide us with a graphical user interface to which we can add controls. Windows Forms and all other Windows controls are contained in the System.Windows.Forms namespace. Windows Forms are in many ways identical to their counterpart Forms in Classic VB but are more modern and offer more properties to work with. VB.NET provides a large number of Windows controls for various purposes. However, use the new controls with good judgment. They exist to create a friendly user interface, not confuse the user.
Although VB.NET is designed to use Windows Forms controls, we can still use ActiveX controls. Therefore, if we own expensive third-party ActiveX controls, we can still use them in VB.NET. To add a control to a Windows Form, click the control’s icon in the Toolbox and then drag and drop over the area on the form where you want the control to be placed. For our solution we add a label control, combo box, and two buttons to the Windows Form and resize the form itself. Figure 24-6 shows how the final Windows Form looks.

Before we add code to the Windows Form, we set the tab order for the controls. Select View > Tab Order from the menu. The tab order for each control is now displayed visually on the form. Clicking on a control’s tab number increases the number. Change the tab order so that it matches the order shown in Figure 24-7. To exit the tab order view, select View > Tab Order from the menu again.

As a final step, we add code to the solution. Select View > Code from the menu. This opens the class module for the Windows Form. The first event we use is the Load event of the Windows Form. This is created by first selecting (Form1 Events) from the combo box in the upper-left corner of the module and then selecting Load from the combo box in the upper-right corner of the module. Listing 24-3 shows the code in the Load event.
Listing 24-3  The Code for the Load Event of the Windows Form

Private Sub Form1_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load

'Create and populate the array with names.
Dim sArrNames() As String = {"Rob Bovey", "Stephen Bullen", "John Green", "Dennis Wallentin")

With Me
    'The caption of the Form.
    .Text = "First Application"

    'The captions of the label and button controls.
    .Label1.Text = "Select the name:"
    .Button1.Text = "&Show value"
    .Button2.Text = "&Close"

    'Populate the combobox control with the list
    'of names.
    With .ComboBox1
        .DataSource = sArrNames
        .SelectedIndex = -1
    End With
End With

End Sub

In this code, we create a string array, set values for various control properties, and then add the array as a data source for the combo box control. We use a single dimension array to populate the combo box with the list of names. It is a perfectly accepted practice to declare and initialize an array at the same time in VB.NET, as shown in Listing 24-3. When using this approach we do not need to specify the size of the array because this is inferred from the number of items within the scope of the curly brackets.

The next step is to get the selected value from the combo box and display it in a message box. Before doing that we need to import the namespace System.Windows.Forms into the code module, which gives us a shortcut to the .NET MessageBox class. Importing namespaces saves keystrokes each time we refer to objects that are part of the imported namespaces. It also makes our code easier to read and maintain by making it less verbose.
The **Imports** statements tell the compiler which namespaces the code uses. Usually we first set a reference to a namespace and then we import it to one or more code modules. Here we only do the latter because the **System** namespace is referenced by default in all new VB.NET solutions. This is because Visual Studio automatically adds a reference to the **System** namespace when a new VB.NET project is created. At the top of the Form’s class module we add the **Imports** statement shown in Listing 24-4.

**Listing 24-4**  The **Imports** Statement

```vbnet
' To use the messagebox object.
Imports System.Windows.Forms
```

The namespace **Microsoft.VisualBasic** also belongs to the namespaces that are referenced by default in all new VB.NET solutions. This namespace is also globally imported, which means we do not need to import it into individual code modules to use it. From a practical standpoint this means we can use the well-known **MsgBox** function instead of its .NET variant. However, in Listing 24-5 we use .NET **MessageBox** class in the Click event for Button1, which displays the selected name in a message box.

**Listing 24-5**  Show Selected Name

```vbnet
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
    ' Make sure that a name has been selected.
    If Me.ComboBox1.SelectedIndex <> -1 Then
        ' Show the selected value.
        MessageBox.Show( ,
            text:=Me.ComboBox1.SelectedValue.ToString(),
            caption:="First Application")
    End If
End Sub
```

The final piece of the puzzle is to add a command to close (unload) the Windows Form in the Button2 Click event. Listing 24-6 shows the required code.
Listing 24-6   Unload the Windows Form

Private Sub Button2_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles Button2.Click
        Me.Close()
    End Sub

To begin testing the application, we just have to press the F5 key. Figure 24-8 shows the First Application in action after we select a value in the combo box and then click the Show value button.

Whenever we execute the application in the debugger, the VS IDE creates a number of new files, including an executable file for our application. These files are located in the ..\First Application\bin\Debug folder. A working example of this solution can be found on the companion CD in the \Concepts\Ch24 - Excel & VB.NET\First Application folder. If you just want to run the application without opening it in Visual Studio, the First Application executable file can be found in the \Concepts\Ch24 - Excel & VB.NET\First Application\First Application\bin\Debug folder on the CD.

Structured Exception Handling

When an unexpected condition occurs in managed code, the CLR creates a special object called an exception. The exception object contains properties and methods that give detailed information about the unexpected condition. Because we deal with exceptions rather than errors in .NET development, we use the expression exception handling rather than error handling.
Exception handling covers the techniques used to detect exceptions and take appropriate actions after they are detected. **Structured exception handling (SEH)**, is the term used to describe how we implement exception handling in managed code. Although it is possible to use the Classic VB error handling approach in VB.NET, we strongly encourage the use of SEH because it gives us much better options for dealing with exceptions. SEH consists of the following building blocks:

- **Try**—We place the code we want to execute in this block. This code may create one or more exceptions.
- **Catch**—In this block we place the code that handles the exceptions. It is possible to place several Catch blocks within the same structure to handle different types of exceptions. Catch blocks are optional.
- **Finally**—Code placed in this block always is executed, which makes this block a perfect place for code to clean up and release references to objects like COM objects and ADO.NET objects. This block is also optional.
- **End Try**—Ends the SEH structure.

Listing 24-7 shows the skeleton structure of SEH in code. When we enter a Try statement in a code module, the VS IDE automatically adds the Catch block and End Try statement. The Finally block must be typed manually.

Listing 24-7  The Building Blocks of SEH

```vbnet
Private Function iDiscount(ByVal iPrice As Integer) As Integer

    Try
        'Do the calculation here.

    Catch ex As Exception
        'In case of any unexpected scenarios take 'some action here, like a message to the user.

    End Try

End Function
```

Most of the namespaces in the .NET Framework class library include their own specific exception classes, which make it possible to catch
them in separate Catch blocks. All built-in exception classes extend the built-in System.Exception class. Catch blocks are executed (or tested for execution) in the order in which they are coded. .NET works its way through the Catch blocks trying to find a matching exception type. Therefore the preferred approach is to implement the Catch blocks with more specific exception types first, followed by the Catch blocks with the more generic exception types. Listing 24-8 shows an example using two Catch blocks.

**Listing 24-8  Using Several Catch Blocks and the Finally Block**

```vbnet
Try
    frmSaveFile = New SaveFileDialog
    With frmSaveFile
        .Filter = "XML File|*.xml"
        .Title = "Save report to XML file"
        .FileName = sFileName
    End With
    dtTable.WriteXml(fileName:=sFileName)
    dtTable.WriteXmlSchema(FileName:=Strings.Left(sFileName, Len(sFileName) - 4) & ".xsd"
Catch XMLexc As Xml.XmlException
    MessageBox.Show(text:=sMESSAGENOTSAVEDXML, _
                    caption:=swsCaption, _
                    buttons:=MessageBoxButtons.OK, _
                    icon:=MessageBoxIcon.Stop)
Catch COMExc As COMException
    MessageBox.Show(text:=_sERROR_MESSAGE & _
            sERROR_MESSAGE_EXCEL, _
            caption:=swsCaption, _
            buttons:=MessageBoxButtons.OK, _
            icon:=MessageBoxIcon.Stop)
Catch Generalex as Exception
```
The first `Catch` block handles any `XmlException` exceptions. The second block catches COM exceptions that might occur when working with COM servers like Excel. The final `Catch` block is generic and handles all other exceptions. The example also shows how we can use the `Finally` block to release an object. Listing 24-8 also shows how to use custom error messages to respond to each exception type.

During development we need to see the underlying technical details for all exceptions. In Listing 24-9 the previously customized end user messages have been replaced with the exception object and its method `ToString` in each `Catch` block. The `ToString` method gives a textual summary of the exception. You can also use the `GetBaseException` method to return the first exception in the chain.

Listing 24-9  Displaying Exception Descriptions

```vbnet
Catch XMLexc As Xml.XmlException
    MessageBox.Show(XMLexc.ToString())
End Try

Catch COMExc As COMException
    MessageBox.Show(COMExc.ToString())
    MessageBox.Show(COMExc.ErrorCode.ToString())
End Try

Catch Generalexc As Exception
    MessageBox.Show(Generalexc.ToString())
End Try
```

When VB.NET receives an exception from a COM server like Excel, it checks the **COM exception** code and tries to map that code to one of the
.NET exceptions classes. If this fails, which is the most common outcome, VB.NET throws a large and mostly unhelpful HRESULT message like the one shown in Figure 24-9.

The line of code that generates this message is the first MessageBox.Show line under the COM exception block in Listing 24-9. COM exceptions are wrapped into generic COMException objects when .NET does not have a matching exception class for the HRESULT error generated by a COM component.

In SEH, it is possible to exit a Try block with the Exit Try statement. This statement can be placed either in the Try block or in any Catch block. Any code in a Finally block is still executed after the Exit Try statement.

Another option is to use nested Try structures. A nested SEH can be added either to the Try block or to a Catch block. When using nested exception handlers the InnerException property of the exception object becomes very important. It helps us determine the cause of the nested exception and allows us to obtain the chain of exceptions that led to that exception.

We can use the Throw statement to communicate exceptions to the calling code. Throw is usually used within a Catch block only if the exception is to be bubbled up the call stack. A Throw statement causes code execution to be intentionally interrupted. The Throw statement also allows us to create our own exceptions, but this topic is beyond the scope of this chapter.

**Modules and Methods, Scope and Visibility**

When we make a declaration at the module level (module here stands for module, class, or structure), the access level we choose determines the scope of the thing being declared. In VB.NET we can use the keywords Public and Private, which have the same scopes as in Classic VB, but VB.NET also provides the following additional keywords to specify module scope and visibility:
- **Friend**—A data member or method (function or subroutine) declared with the Friend modifier can be accessed from any part of the program containing the declaration. This is not a new keyword, as it is also available in Classic VB. However, if we do not explicitly include a scope in our declaration, then the default scope is Friend in VB.NET, while in Classic VB the default scope is Public.

- **Protected**—Data members or methods declared with Protected scope are only accessible from the module itself or from derived classes.

- **Protected Friend**—This scope is equivalent to the union of Protected and Friend access. A data member or method declared as Protected Friend is accessible from anywhere in the program in which the declaration occurs, or from any derived class containing the declaration.

### Declare Variables and Assign Values

In VB.NET, we declare local variables using the keyword `Dim`, module-level variables using the keyword `Private`, solution-level variables using the keyword `Friend`, and public variables using the keyword `Public`. All .NET programming languages provide the option to declare variables and assign values to them at the same time.

The first two lines in Listing 24-10 show how we can declare variables and initialize them with values using one line of code. The third line creates three String variables without assigning any values to them. Since they don’t have assigned values, these String objects are marked as unused local variables by the VS IDE. This is a result of the `Option Strict` setting being on. Good coding practice in .NET says that we should always assign known values to variables, even if they initially will not have any “real” values. Lines 4 through 6 show how we can achieve this in practice.

#### Listing 24-10  Declare Variables and Assign Values to Them

1. Dim sTitle As String = "PETRAS Report Tool"
2. Dim iPrice As Integer = 100
3. Dim sAddress, sCity, sCountry As String
4. Dim sName = String.Empty
5. Dim bReportStatus = Nothing
6. Dim iNumberOfRecords As Integer = Nothing
7. Dim iNumberOfColumns As Integer = dtTable.Columns.Count - 5
8. Dim iNumberOfRows As Integer = dtTable.Rows.Count - 1
9. Dim obDataArr(iNumberOfRows, iNumberOfColumns) As Object
Lines 7 and 8 in Listing 24-10 contain two variables that hold the number of columns and rows of a **DataTable** (an ADO.NET object covered later in this chapter). These two variables are then used as parameters to dimension the array of data type Object in line 9. The data type Object is the VB.NET counterpart to the data type Variant in Classic VB. An Object array behaves in roughly the same manner as a Variant array.

VB.NET also offers the ability to declare variables anywhere in the code. Listing 24-11 shows an example where we have declared a variable within a **Try** block in conjunction with a **For...Next** loop.

**Listing 24-11  Block Scope Variable Declaration**

```vbnet
Try
  For iCountRows As Integer = 0 To 9
    'Do the iteration.
    Next iCountRows
  Catch ex As Exception
    MessageBox.Show(ex.ToString())
End Try
```

Block scope can also be achieved by declaring variables within **With...End With** blocks, **For...Next** blocks, and **Do...Loop** blocks. In Listing 24-12 we show a variable that is declared in a **Do...Loop**.

**Listing 24-12  Block Scope within a Do...Loop**

```vbnet
'Declaration of a variable with
'a block scope of Do...Loop.
Do
  Dim iMonth As Integer = 1
  'Other code goes here...
Loop
```

However, declaring variables using this method may cause unexpected problems. This is because the scope of variables declared in this manner is limited to the block in which they are declared. This means we cannot
access these variables or use them outside that block. Code that uses this method can also be more difficult to debug and maintain. In general we should avoid this approach. Good coding practice suggests that all variables used within a method should be declared at the beginning of that method.

**Creating New Instances of Objects**

We can create new instances of objects in VB.NET using the same techniques as in Classic VB. The only difference is that we do not use the `Set` keyword in VB.NET. Listing 24-13 shows two methods of creating objects in VB.NET. The `Nothing` keyword is a way of telling the system that the variable does not currently have any value but still may use memory.

Listing 24-13  Declare and Instantiate Objects

```vbcn```
' The classic approach.
Dim frmSaveDialog As SaveFileDialog = Nothing
frmSaveDialog = New SaveFileDialog

'.NET approach.
Dim frmSaveDialog As New SaveFileDialog
```

The .NET approach is singled out in the second example in Listing 24-13, which shows that we declare and set the variable to a new instance of the SaveFileDialog class with one line of code. Although the .NET approach may look attractive, we still recommend using the classic approach. This is also outlined as the best practice in Chapter 3.

Using the .NET approach can cause unwanted exceptions because of the block scoping of variables. For example, if we create a new instance of the SaveFileDialog component and we want to trap any exceptions that may occur (or we want to throw an exception), block scoping of the variable itself causes an exception. This is demonstrated in Listing 24-14, where we have declared and instantiated the `frmSaveDialog` object variable in the `Try` block. However, because the scope of this variable is limited to the `Try` block, the VS.IDE displays a compile error for the two lines of code inside the `Finally` block.

Listing 24-14  Using the .NET Approach

```vbcn```
Sub Show_Save_Dialog()

Try
To correct this problem, we modify the code to use the classic approach as shown in Listing 24-15. The `frmSaveDialog` variable can now be seen throughout the `Try` block, and it traps any exceptions that may occur.

Listing 24-15  Using the Classic Approach

```vbnet
Sub Show_Save_Dialog()
    Dim frmSaveDialog As New SaveFileDialog
    frmSaveDialog.ShowDialog()

    Catch ex As Exception
        MessageBox.Show(ex.ToString())
    Finally
        frmSaveDialog.Dispose()
        frmSaveDialog = Nothing
    End Try
End Sub
```
Using ByVal or ByRef

Unlike Classic VB, procedure arguments in VB.NET are passed ByVal by default, not ByRef. If we do not explicitly specify procedure arguments as either ByVal or ByRef, the VB.NET default is ByVal. However, good practice states that we should always explicitly specify the keyword we want to use.

Using Wizards in VB.NET

Compared to the wizards in Classic VB, the wizards in VB.NET have been significantly improved. New wizards have also been added to the VS IDE. The advantage of using a wizard is that we get the desired result in a fast and reliable way without needing to have a deep understanding of the process. The wizard takes care of the details. The disadvantage of using a wizard is that the wizard works in “black box” mode, which means we do not have much control over the process. Developing real-world applications requires you to be in control and to understand your solutions inside and out. You can explore the wizards in the VS IDE, but for any non-trivial solution you should avoid them.

Data Types in VB.NET

Compared with Classic VB, some data types are new in VB.NET. Table 24-2 shows most of the VB.NET data types but not all of them.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Size</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>2 bytes</td>
<td>True or False.</td>
</tr>
<tr>
<td>Short</td>
<td>2 bytes</td>
<td>-32,768 to 32,768.</td>
</tr>
<tr>
<td>Integer</td>
<td>4 bytes</td>
<td>-2,147,483,648 to 2,147,483,648.</td>
</tr>
<tr>
<td>Decimal</td>
<td>16 bytes</td>
<td>It provides the greatest number of significant digits for a number.</td>
</tr>
<tr>
<td>Double</td>
<td>8 bytes</td>
<td>It provides the largest and smallest possible magnitudes for a number.</td>
</tr>
</tbody>
</table>
As we can see in Table 24-2, the data type **Short** includes the interval -32,768 to 32,768, and the **Integer** data type now covers a much greater interval than it does in Classic VB. The Currency data type is no longer available in VB.NET. It has been replaced by the new **Decimal** data type, which can handle more digits on both sides of the decimal point. The Byte data type from Classic VB has no counterpart in VB.NET. The data type **Object** is the universal data type in VB.NET, taking the place of the Variant data type in Classic VB.

### String Manipulation

As previously mentioned, whenever a new .NET solution is created the namespace `Microsoft.VisualBasic` is included by default. This provides access to the .NET versions of the well-known string functions in Classic VB. The .NET Framework also provides us with a `System.String` class to manipulate strings. However, using the old functions has no negative impact on solution performance, so using the old familiar functions is completely acceptable.

### Using Arrays in VB.NET

The .NET Framework provides us with powerful new options for creating and using arrays and collections in VB.NET. There are two basic kinds of VB.NET arrays. Arrays that we declare as array variables of a specific data type by using parentheses after the variable name are normal arrays. We can also use the Array class, which provides us with a new array data type that offers methods for managing items in arrays as well manipulating arrays. Arrays in VB.NET inherit from the Array class in the System namespace, so methods of the Array class can also be used with normal arrays.

In this section, we discuss normal arrays along with some methods of the Array class. In VB.NET, all arrays are zero-based. This is important to keep in mind, especially when working with Excel objects or Classic VB.
code that may have 1-based arrays. We already showed one way to use an array in Listing 24-3, where we used an array to populate a combo box control. In Listing 24-16 we use the same approach to populate a list box control and then add the selected items to an array.

**Listing 24-16**  Populate an Array with Selected Items from a List Box

```vbnet
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
    'Make sure that at least one item is selected.
    If Me.ListBox1.SelectedIndex <> -1 Then
        'Grab the number of selected items.
        Dim iCountSelectedItems As Integer = Me.ListBox1.SelectedItems.Count - 1

        'Declare and dimension the one-dimensional array.
        Dim sArrSelectedItems(iCountSelectedItems) As String

        'Populate the array.
        For iCountSelectedItems = 0 To iCountSelectedItems
            sArrSelectedItems(iCountSelectedItems) = Me.ListBox1.SelectedItems(iCountSelectedItems).ToString()
        Next iCountSelectedItems

        'Show the number of items in the array.
        MessageBox.Show(CStr(sArrSelectedItems.GetLength(0)))

        'Show the lower bound of the array.
        MessageBox.Show(CStr(sArrSelectedItems.GetLowerBound(0)))

        'Show the upper bound of the array.
        MessageBox.Show(CStr(sArrSelectedItems.GetUpperBound(0)))

        'Iterate through the array and display each value.
        For iCountSelectedItems = sArrSelectedItems.GetLowerBound(0) To sArrSelectedItems.GetUpperBound(0)
            MessageBox.Show(text:= sArrSelectedItems(iCountSelectedItems).ToString())
        Next iCountSelectedItems
    End If
End Sub
```
When working with arrays we should always specify which dimension we are targeting. Since we are working with a one-dimensional array in this example, the dimension we are targeting is zero.

One of the more resource-intensive processes in VB development is redimensioning arrays, so we should always look for ways to reduce or eliminate this process. Listing 24-16 shows how VB.NET allows us to do this easily. We first retrieve the number of selected list items and then declare and dimension the array all at once. Note that in Listing 24-16 we use the `GetLowerBound` and `GetUpperBound` methods to return the lower bound and upper bound index values for the array. Both these methods are part of the Array class. In some scenarios we may not know the bounds for an array initially, but we can get the necessary information later. Listing 24-17 shows how we can initialize an array after declaring it.

**Listing 24-17  Declare an Array and Initialize It Later**

```vbnet
Dim iNumberOfHouses() As Integer
...
...
iNumberOfHouses = New Integer() {10, 15, 20}
```

Listing 24-16 shows one way to iterate an array, but we could actually enumerate it as shown in Listing 24-18.

**Listing 24-18  Enumerating an Array**

```vbnet
Dim iNumberOfHouses() As Integer = {10, 15, 20}
Dim iItem As Integer

For Each iItem In iNumberOfHouses
    Debug.WriteLine(iItem)
Next iItem
```

The Array class also provides methods that allow us to manipulate the items in different ways. Among the more common actions we might want to perform on an array are reversing the order of items in the array, sorting the array, removing items from the array, returning specific array items, and copying items from one array to another. Listing 24-19 shows how to perform these operations using methods of the Array class.
Listing 24-19  Methods of the Array Object

Dim sArrProjects() As String = _
    {"Upgrade","Investment", "Maintenance"}

Array.Reverse(sArrProjects)

Array.Sort(sArrProjects)

Array.Clear(sArrProjects, 0, 1)

Dim sItem As String = sArrProjects.GetValue(1).ToString()

Dim sArrProjectsCopy(sArrProjects.GetLength(0)) As String

Array.Copy(sArrProjects, sArrProjectsCopy, _
    sArrProjects.GetLength(0))

The first example shows how to reverse the order of the items in an array. The second example sorts the array in ascending order. The third example shows how to delete the first item from an array. Note that deleting an item from an array in this manner does not resize the array or move any of the other items into new positions in the array.

To get a specific item value from an array you use the GetValue method, as shown in the fourth example. And as the final example shows, we can even copy one array to another using the Copy method. The last argument of this method allows us to specify the number of items to be copied. This can be a good alternative to the redimension approach when resizing an array. In this example we copy all items from the first array into the second array.

Next we demonstrate how to search for a value in an array using the BinarySearch method. This method is useful when you want to determine whether a specific value exists in an array. To use this method the items in the array must be sorted. The result of executing the BinarySearch method is an integer that represents the index number of the value you are searching for within the array. If the result is -1 the value you are searching for does not exist. If the value you are searching for exists more than once within the array, the index number of the last occurrence is returned.

Listing 24-20 shows how to use the BinarySearch method to locate the index number of an item in an array. There are also several other methods of the array object that allow us to find specific items and work with them in various ways.
Listing 24-20  The BinarySearch Method

Dim sArrProjects() As String = _
    {"Upgrade", "Investment", "Maintenance"}

Dim sSearchedValue As String = "Investment"

Array.Sort(sArrProjects)

Dim iSearchedIndex As Integer = _
    Array.BinarySearch(sArrProjects, sSearchedValue)

MessageBox.Show(CStr(iSearchedIndex))

A good alternative to the normal array is the ArrayList class, which is part of the System.Collection namespace. By using this class we can dynamically increase a list, hold several different data types in one list, manipulate the elements in a list, and manipulate ranges of elements in one operation. The ArrayList is something of a hybrid between the Array and Collection objects. In Listing 24-21 we demonstrate the use of an ArrayList object.

Listing 24-21  Working with the ArrayList Object

Dim Arrlst As New ArrayList(7)
Dim oArrlstObject As Object = Nothing

Debug.Print(Arrlst.Capacity.ToString())

With Arrlst
    .Add("Dennis")
    .Add(True)
    .Add(12)
End With

Debug.Print(Arrlst(1).GetType.ToString())

Dim sNames() As String = {
    "Rob Bovey", _
    "Stephen Bullen", _
    "John Green", _
    "Dennis Wallentin"}

Arrlst.AddRange(sNames)
We first create a new ArrayList object and dimension it to hold seven items. Expanding an ArrayList is a resource-intensive process, so we want to try and create it with the capacity to hold as many items as we will need. The first debug print command gives us the current capacity of the ArrayList. We then populate the ArrayList object with items that represent different data types, in this case a string value, a boolean value, and an integer value. To verify that the ArrayList actually holds different data types we print the data type of the second item to the Immediate window using the `GetType` method.

Next we add a range of values to the ArrayList using the `AddRange` method. Our ArrayList already has the capacity to hold these new items, but if an ArrayList does not have sufficient capacity to hold the number of items being added it automatically expands itself. The `RemoveRange` method enables us to remove several items at once, so next we use this method to remove the first three items we added to it. At this stage the ArrayList object still has a capacity of seven items, but since we no longer need them all we resize it by using the `TrimToSize` method. Using the debug print command to check the capacity of the ArrayList after resizing it should show a capacity of four items. Just to check which values the ArrayList now holds we iterate over all its items using a `For...Each` loop. Finally, the collection of items in the ArrayList is added to a `CheckedListBox` control.

In addition to the ArrayList, the .NET Framework provides additional data structures like `Stack` and `Queue`. The Stack class is a data structure that allows adding and removing objects from one position only. This position is referred to as the “Top” of the stack. The last object placed on the stack is the first one to be removed. This is a Last In First Out (LIFO) data access method. The Queue class is a data structure that allows us to
add objects to the back and remove objects from the front. This is a **First in First Out** (FIFO) data access method.

## Debugging

The most important task in development is to debug non-trivial solutions. The VS IDE offers a large number of tools to assist you in this task. Depending on the complexity of the solution, debugging can be quite difficult and time consuming. One of the best features of the VS IDE is that we can interact with it during debugging sessions.

Selecting the **Debug** menu reveals the available tools and options. As we can see, most of the commands and windows are familiar from Classic VB. During the debugging process, and while in break mode, additional tools become available as shown in Figure 24-10. Although a detailed walkthrough is beyond the scope of this chapter, we focus on the most important new and updated debugging tools that the VS IDE provides. See the Chapter 16, “VBA Debugging,” for a more detailed discussion of the debugging process.

**Figure 24-10** Debugging tools available in break mode

### Set Keyboard Shortcuts

Before we start to explore the many tools for debugging, we customize our keyboard shortcuts. Select **Tools > Options...** from the menu to display the
Options dialog. In the Options dialog tree view, select the **Keyboard** section under **Environment**, as shown in Figure 24-11.

This section allows us to set the mapping scheme for keyboard shortcuts. Changing the mapping scheme to **Visual Basic 6.0** provides access to all the well-known VB6 keyboard shortcuts in the VS IDE. This change is global, meaning it will be applied for all VB.NET solutions in the VS IDE. The keyboard shortcuts mentioned in the rest of this chapter assume this setting has been made in your environment.

### Enable Unmanaged Code Debugging

If we do a lot of interoperability development, that is, calls to COM objects, the option **Enable unmanaged code debugging** gives us the possibility to debug the native code. Select **Project > [Solution Name] Properties...** from the menu to display the Properties window; then select the **Debug** tab and check this option.

### The Exception Assistant

Whenever a runtime exception is thrown, the Exception Assistant highlights the line of code that caused the exception and displays a dialog with suggestions on how to solve the problem. Figure 24-12 shows the Exception Assistant in action.

The Exception Assistant attempts to provide context-sensitive help related to the exception, and it allows the developer to perform certain
actions, such as viewing details of the exception and copying exception information to the Clipboard. For COM exceptions, however, the information provided by the Exception Assistant is of limited value.

We can provide troubleshooting tips for our own exception types by creating an XML file containing the information in the correct ExceptionAssistantContent directory under C:\Program Files\Microsoft Visual Studio 9.0\Common7\IDE\ExceptionAssistantContent.

**The Object Browser (F2)**

The Object Browser is one of the most valuable development resources. The VS IDE ships with a modern Object Browser that can be customized by selecting the Object Browser Settings icon on its toolbar, as shown in Figure 24-13. We can also add components to the Custom Component Set Browsing scope by selecting the Edit Custom Component Set button directly to the right of the Browse drop-down in the Object Browser toolbar.
The **Browse** drop-down is used to limit the scope of items displayed in the Object Browser. One of the selections available is to browse **My Solution**, as shown in Figure 24-13. This option allows us to browse the objects in our solution as well as any outside namespaces the solution references.

### The Error List Window (Ctrl+W Ctrl+E)

The **Error List window** shows errors, warnings and other messages that result from attempting to compile the active project. It detects most common syntax and deployment errors. Figure 24-14 shows an example Error List window displaying some errors. Double-clicking on an item in the list takes you to the module and line of code it refers to.

![Figure 24-14 The Error List window](image)

The keyboard shortcut to display the Error List window requires two steps, Ctrl+W followed by Ctrl+E. It may feel a bit odd to use two instructions to access a feature, but this reflects how many features the VS IDE contains.

### The Command Window (Ctrl+Alt+A) and Immediate Window (Ctrl+G)

The **Command window** and Immediate window overlap each other to some degree, but they actually have two different tasks to accomplish. The Command window allows you to execute VS IDE commands instead of going through the menus and toolbars. It can also execute commands to open other windows.

Suppose we have started a debugging session and we are running in break mode. If we enter the command shown in Listing 24-22 into the Command window the variable bExport will be added to the Watch window.
If we want to see all the command aliases, or command shortcuts, defined by the VS IDE, we can run the command `>Alias` in the Command window to produce a list.

The Immediate window in the VS IDE behaves much like its counterpart in Classic VB. We can assign variables, run procedures, and invoke methods in standard VB.NET syntax in the Immediate window.

**The Output Window (Ctrl+Alt+O)**

The Output window displays compilation results and the text output from several tools such as Debug and Trace. The Show output from: drop-down in the toolbar allows you to show the output from either the debug or the build process. It is also possible to save the output to a text file by clicking anywhere inside the Output window and then using the keyboard shortcut Ctrl+S.

**Break Points (Ctrl+Alt+B)**

To insert a new break point, use the keyboard shortcut Ctrl+B. Compared with its older sibling in Classic VB, the break points feature has been improved significantly in VS.NET. First, VS.NET provides a Breakpoints window that displays the location and settings for all break points in the solution, as shown in Figure 24-15.

Second, we can set conditions for a break point by right-clicking on that break point and selecting Condition... from the shortcut menu. In Figure 24-15 we set a condition for the first break point. When the break point is reached, this condition is evaluated to determine whether it is true or false.
If the condition is true the break point is triggered; otherwise, the break point is skipped.

Third, we can add a **hit count** for a break point by right-clicking on that break point and selecting **Hit Count...** from the shortcut menu. This provides us with an additional parameter to control whether code execution should stop at break points. Figure 24-16 shows us defining a hit count for our first break point in the Breakpoint Hit Count dialog.

The **Call Stack (Ctrl+L)**

The Call Stack window displays the method calls that are currently on the stack. It is a useful debugging tool because it allows you to see the specific execution path that led to the current position in your code.

The **Quick Watch and Watch Windows**

Once our code is in break mode we have access to the Quick Watch and Watch windows. The Watch window, accessed by selecting the **Debug > Windows > Watch** menu while in break mode, provides four different Watch tabs. It is easy to add watches. You can drag and drop an object or expression onto the Watch window or select the object or expression in the code editor, right-click on it, and choose **Add Watch** from the shortcut menu. To delete a watch, select it in the Watch window, right-click on it, and choose **Delete Watch** from the shortcut menu. You can add as many different watches as you want. To access one of the Watch windows during a debugging session, press Ctrl+Alt+W followed by a digit between 1 and 4.

Quick Watch works the same way as the Watch window except that it can only handle one watch variable at the time.

The **Exceptions (Ctrl+Alt+E)**

The Exceptions dialog is an advanced debugging tool that allows us to specify what types of exceptions we want VS.NET to throw during debugging.
The debugger stops whenever the selected type of exception occurs. Figure 24-17 shows this dialog.

![The Exceptions dialog](image)

The *Thrown* option causes the debugger to break unconditionally when the specified exception type occurs. If we check the *Thrown* option for the **Common Language Runtime Exceptions**, we ensure that when a Common Language Runtime exception is thrown it breaks into the debugger, overriding any custom SEH we may have defined. The *User-unhandled* option causes the debugger to stop for the specified exception type only if no error handler is active when the exception occurs.

We can also configure specific exception types below the top-level namespaces by clicking on the plus sign (+) to the left of a namespace. This expands the namespace node to show all exceptions within the namespace that can be configured.

**Conditional Compilation Constants**

Chapter 16 introduced the concept of conditional compilation constants, so in this section we only cover conditional compilation topics that are specific to the .NET platform.

VB.NET provides several predefined conditional compilation constants, including the Boolean constant DEBUG. When DEBUG is set to true we have a **debug build**, and when it is set to false we have a **release build**. When compiling a release builds we do not need to manually remove any debugging information. VS.NET handles this automatically when the DEBUG constant is set to false. Debugging information is also ignored when running a release build in the VS IDE. To compile a release build we
need to use the Configuration Manager. Verify that the Configuration Manager is available in the following manner:

1. Select the Tools > Options... menu from the VS IDE.
2. Select Projects and Solutions from the tree view in the Options dialog.
3. Check the Show advanced build configurations check box.
4. Click the OK button to close the Options dialog.

We can then access the Configuration Manager by selecting Build > Configuration Manager... from the VS IDE menu, as shown in Figure 24-18. By changing the configuration we can switch between debug and the release builds. We can also use the Configuration Manager to specify which platform to target.

We can execute code conditionally based on the value of the DEBUG constant as shown in Listing 24-23.

Listing 24-23  Using DEBUG in Code

#If DEBUG then
'Do some evaluation.
#End If
Using Assertions

The `Debug.Assert` method is used exactly the same way in the VS IDE as it is in Classic VB. Chapter 16 already covered the use of this method, so we do not discuss it further here.

Useful Development Tools

The VS IDE ships with a large number of useful development tools. Although it is beyond the scope of this chapter to discuss them all, we cover some of the most important tools in this section.

Code Region

The **Code Region** feature allows us to expand and collapse different sections, or regions, in our code modules. We can use this feature to create logical groups of methods that expand and collapse together. We can then collapse all regions in a code module that are unrelated to the one we are working with.

To create a region, we enter `#Region` followed by the name of the region in double quotes on a blank line above where the region should start. We then move to the next blank line below the code we want included in the region and enter `#End Region` (or select it from the IntelliSense list when we are prompted). Listing 24-24 shows an example of a code region.

```
#Region "Export data to Excel"
'Many lines of code here
#End Region
```

The Code Snippets Manager (Ctrl+K Ctrl+B)

Code snippets are small, reusable pieces of code. They are stored in a snippet library and managed using the **Code Snippets Manager**. The VS IDE includes a large number of code snippets already written and stored in the Code Snippets Manager. Code snippets are particularly easy to use because they are exposed as part of the VS IDE IntelliSense feature. Code snippets are stored in text files in XML format. This makes it easy to use them on other computers as well as to share them with other developers.
You can insert a code snippet into your code module in the following manner:

1. Place the cursor at the position where you want to insert the code snippet.
2. Right-click and select Insert Snippet... from the shortcut menu.
3. Select the desired category.
4. Select the desired code snippet.

Figure 24-19 shows the Insert Snippet command in action.

Instead of using the menu to insert code snippets, we can use code shortcuts. First we need to find out which code shortcuts are available in the Code Snippets Manager. The Code Snippets Manager can be accessed from the Tools > Code Snippets Manager... menu. Next we type the shortcut text, for instance ForEach, in the code editor and press the Tab key to execute it. Listing 24-25 first shows the shortcut text and then the result after we press the Tab key.

### Listing 24-25 Using a Shortcut to Insert a Code Snippet

```vbnet
' The shortcut.
ForEach

' The result.
For Each Item As String In CollectionObject

Next Item
```

As we can see in Listing 24-25, we need to fix the code snippet before it can be properly used. On first consideration it may seem like too much...
effort to remember all the shortcuts as well as correct the code that is actually inserted into the code editor. However, the code snippets are completely customizable, so it is worth your effort to spend some time and make the changes required to suit your needs.

The built-in Code Snippets tool is rather primitive and doesn’t provide a very user-friendly interface. If you find yourself working extensively with code snippets the free Snippet Editor may be a better tool. As of this writing, it is available at www.codeplex.com/SnippetEditor.

**Insert File as Text**

Insert File as Text is not a standalone tool but rather a built-in function of the VS IDE. It can be used to import code from plain text files. To display the Insert File dialog select Edit > Insert File as Text... from the menu. The default file extension is *.vb so we need to change the file extension to *.txt before we can select a text file. The code in the selected text file is imported into the active code module at the current cursor position. Using text files to manage complete and reusable class modules, standard modules, and methods requires only a simple text editor, making it a portable, light-weight solution.

**Task List (Ctrl+Alt+K)**

The Task List is a simple but handy tool for managing the To-Do list for a solution. Using the only button on its toolbar we can create different tasks and set flags indicating their priority. By right-clicking on the list we can also sort, copy, and delete tasks.

**Automating Excel**

At the most fundamental level, automating Excel from .NET solutions does not differ from automating Excel from Classic VB. What must be taken into consideration is that the .NET Framework cannot communicate directly with Excel because of differences between .NET technology and the COM technology Excel is built on. It is necessary to create a bridge between these two technologies for us to be able to automate Excel from the .NET platform. The bridge between .NET and COM is mostly provided by features contained in two .NET Framework namespaces: System.Runtime.InteropServices and System.EnterpriseServices.
However, there are additional components required to allow interoperability that we need to discuss further.

**Primary Interop Assembly (PIA)**

When we set a reference to a COM type library in a .NET solution, the VS IDE automatically creates a default Interop Assembly (IA). The auto-generated IA is a .NET-based assembly that acts as a wrapper for the COM type library. The IA provides us with basic access to the COM type library, and it contains type definitions (as metadata) of types implemented by COM. A Primary Interop Assembly (PIA) is a prebuilt, vendor-supplied assembly. The difference between an IA and a PIA is more or less semantic.

Microsoft has released PIAs for all Excel versions beginning with Excel 2002 as part of the Microsoft Office PIAs. The PIAs have strong names and are digitally signed by Microsoft. The use of strong names makes it possible to install PIAs in the Global Assembly Cache (GAC). The GAC is a machinewide .NET assembly cache for the CLR. Assemblies that should have only one version on the system should be installed in the GAC.

One important point to understand is that only one version of the PIAs can be used on a system, although multiple versions can be installed side by side in the GAC. In addition, PIAs are registered in the Windows registry. If multiple versions of the PIAs are installed, only the latest version is registered, and the entries for any previous version are overwritten.

When we set a reference to Excel in a .NET solution the VS IDE reads the registry and adds a reference to the PIA instead of generating a new IA. This guarantees that we always use the PIAs if they are available. As a practical matter, when we are automating Excel from .NET we are always developing against the PIA and not the Excel COM type library.

The PIAs are optimized for Excel and you should always use the official Microsoft versions. The PIAs are also Excel version-specific. This means you cannot automate Excel 2002 using the PIA for Excel 2003. Therefore, you must be sure the correct version of the PIA is installed on your development computer. Whether or not the PIA is already installed on a computer depends on the following:

- For Excel 2002 on Windows XP or Windows Vista you need to manually download and install the redistributable PIA package from the
Microsoft Web site. If you run Windows XP, then the .NET Framework must be installed prior to installing the PIA package.

- For Excel 2003 or Excel 2007 on Windows XP, if Microsoft Office has been installed before the .NET Framework, then you must install the PIA package manually. You can either download the redistributable PIA package from the Microsoft Web site or install it from the Office CD.
- For Excel 2003 or Excel 2007 on Windows Vista you do not need to take any action at all. Because version 3.0 of the .NET Framework is shipped with Windows Vista, the PIAs are automatically installed when Office is installed.

Since no official PIA exists for Excel 2000, we must compile our own IA using the TlbImp.exe tool that is shipped as part of the .NET Framework SDK. It takes the Excel9.olb file as its input and generates a .NET assembly as its output. When automating Excel from .NET you should always develop against the earliest versions of the PIA and Excel that you plan to target and the earliest version of the .NET Framework you intend to use.

You need to be aware of the code execution overhead for all kinds of .NET solutions, especially when it comes to interaction between .NET and COM. Compared with Classic VB, .NET solutions require more components and therefore require more overhead to run. These components include:

- The COM interop layer (PIA)
- The CLR
- The .NET Framework

As we see in the next section, there are additional aspects we need to consider to maintain acceptable performance for .NET solutions that automate Excel. If high performance is critical to your solution you may even consider using Classic VB if it is available and is an acceptable development platform.

**Using Excel Objects in .NET Solutions**

Create a Windows Forms solution and name it “Automate Excel.” Add a button to the form and name it “Automate Excel.” Next, add a reference to the Excel 2003 PIA or later. Choose Project > Add Reference... from the VS IDE menu to display the Add Reference dialog. Select the COM tab.
and scroll down to locate the Microsoft Excel 11.0 Object Library as shown in Figure 24-20. The reference is added when you close the dialog by clicking the OK button.

Choose Project > Automate Excel Properties... from the VS IDE menu. Select the References tab, and you see that three new Excel-related references have been added: the Excel Object Library, the Office Object Library, and the VBA Extensibility Object Library. Figure 24-21 shows the current list of references in the solution.

The System references are added by default. These give us access to the most commonly used .NET Framework class libraries. The imported namespaces are automatically included in all new .NET solutions. These are globally available in a solution. Open the Windows Form class module. When working with namespaces like Excel it is a good development practice to create a namespace alias for it at the top of the code module. We also add another Imports statement that is required as shown in Listing 24-26.

Listing 24-26  Namespace Alias and Imports Statements

```
'Namespace alias for Excel.
Imports Excel = Microsoft.Office.Interop.Excel

'To release COM objects and catch COM errors.
Imports System.Runtime.InteropServices
```
Declaring and instantiating some Excel COM objects, like Workbook and Range objects, requires that we **cast** the object reference to the precise type using the **CType** function. This is because the Option Strict setting prevents us from using code that might fail at runtime due to type conversion errors. The VS IDE actually helps us with this task by visually marking the objects that need to be cast.

Next, add a Click event handler for the button. Listing 24-27 shows the code required to get the Excel automation started. Put this code in the button’s Click event. As you can see, we implemented an SEH but intentionally left out any exception handling code. At this stage we also did not add the code required to release any of the Excel objects we used.

**Listing 24-27  Declare and Instantiate Excel Objects**

```vbscript
Dim xlApp As Excel.Application = Nothing
Dim xlWkbNew As Excel.Workbook = Nothing
Dim xlWksMain As Excel.Worksheet = Nothing
Dim xlRngData As Excel.Range = Nothing
Dim sData() As String = ("Hello", "World", "!")
```
Try
  'Instantiate a new Excel session.
  xlApp = New Excel.Application

  'Add a new workbook.
  xlWkbNew = xlApp.Workbooks.Add

  'Reference the first worksheet in the workbook.
  xlWksMain = CType(xlWkbNew.Worksheets(Index:=1), Excel.Worksheet)

  'Reference the range to which we will write some data to.
  xlRngData = CType(xlWksMain.Range("A1:Cl"), Excel.Range)

  'Write the data to the range.
  xlRngData.Value = sData

  'Save the workbook.
  xlWkbNew.SaveAs(Filename:="c:\Test\New.xls")

  'Make Excel visible for the user.
  With xlApp
    .UserControl = True
    .Visible = True
  End With

Catch COMex As COMException
  Catch ex As Exception

End Try

As shown in Listing 24-27, we must explicitly use the Value property of the Excel Range object in VB.NET. This is because VB.NET does not recognize default properties.

Whenever Excel objects are instantiated at runtime the CLR creates so called **Runtime Callable Wrapper (RCW)** for each underlying COM object in the memory. It is the group of RCWs that constitute the
runtime proxies, or bridges, between a .NET solution and the COM type libraries it references. This is important to keep in mind because the more Excel COM objects we use, the more memory our solution consumes at runtime. It is a good development practice to clean up the RCW reference counts so we don’t end up with a large number orphaned RCWs.

Let’s take a closer look at the code in Listing 24-27. Initially it looks like we are only using four objects: the Application object, the Workbook object, the Worksheet object, and the Range object. But we indirectly reference the Workbooks collection, the Worksheets collection, and the Range collection, so we actually use seven objects. The objects used indirectly are out of our control but must be managed anyway.

On the .NET platform the Garbage Collector (GC), is responsible for all memory management. The GC uses a managed memory scheme that periodically traces live references. When the trace is complete, all unreachable objects are released, and the GC reclaims the memory they previously used. The GC operates in a nondeterministic manner, so we never know exactly when it will perform its memory management tasks.

For pure .NET solutions this is not a problem, but it becomes an issue when trying to release COM objects properly. When releasing Excel objects we must be sure to release all the objects we have used. Otherwise, we may end up in a situation where Excel remains in memory and continues to consume resources even after our application has ended.

The first step in a practical solution is to explicitly call the GC from our .NET code. Calling the GC is a time-consuming process, but one that may be necessary when automating Excel because it is the only way to release all the Excel COM objects referenced indirectly. Each RCW has a finalizer that is responsible for releasing its COM object from memory. This finalizer needs to be called twice to fully remove the COM object from memory. Therefore, if we call the GC twice it releases our three indirectly referenced Excel objects.

The second step in a practical solution is to call the Marshal.FinalReleaseComObject method for every Excel COM object. Note that Excel objects must be released in the reverse order in which they were created, with the Excel Application object released last. Listing 24-28 shows the code in our solution used to release all the Excel COM objects. This should normally be performed when we are closing the application.
Listing 24-28  Releasing Excel COM Objects with a Function

' In the calling sub procedure.
'...
Finally

' Calling the Garbage Collector twice.
GC.Collect()
GC.WaitForPendingFinalizers()
GC.Collect()
GC.WaitForPendingFinalizers()

' Releasing the Excel objects.
ReleaseCOMObject(xlRngData)
ReleaseCOMObject(xlWksMain)
ReleaseCOMObject(xlWkbNew)
ReleaseCOMObject(xlApp)

End Try

End Sub

Private Sub ReleaseCOMObject(ByVal oxlObject As Object)
Try
    Marshal.ReleaseComObject(oxlObject)
    oxlObject = Nothing
Catch ex As Exception
    oxlObject = Nothing
End Try

End Sub

Note how we use the custom ReleaseCOMObject function to release the Excel objects and set them to Nothing. This example also shows why the Finally block is so useful; it ensures that the code required to clean up our Excel objects will always run.

The Automate Excel example can be found on the companion CD in \Concepts\Ch24 - Excel & VB.NET\Automate Excel folder. If you just want to run the example, the Automate Excel executable file can be found in the \Concepts\Ch24 - Excel & VB.NET\Excel Automate\Excel Automate\bin\Debug folder on the CD.
Using Late Binding

Whenever possible, we should use early binding and declare all variables as specific types. The reasons for this are simple:

- Our .NET solutions run faster because it is not necessary to perform type conversion on any variables.
- The compiler can detect and display exceptions and therefore prevent runtime exceptions.
- We get IntelliSense support and dynamic help during the development process.

Unfortunately, it is common for developers to have the latest version of an application such as Microsoft Office while end users have earlier versions. However, given access to desktop virtualization software such as VMWare (commercial software) and Microsoft Virtual PC (free tool) it is now much easier for developers to use the same versions of software as the end users they develop for. This makes it possible for developers to use early binding in their applications.

Resources in .NET Solutions

On the .NET platform we can add images, icons, strings, and text files as resources to our solutions. To add resources we select the Resources tab from the .NET solution Properties window and click the Add Resource button on its toolbar. All resources associated with a solution become part of the EXE or DLL file upon compilation of the solution.

NOTE VS 2008 ships with a large group of images and icons. These are contained in the file VS2008ImageLibrary.zip that is located in the folder \Program Files\Microsoft Visual Studio 9.0\Common7\VS2008ImageLibrary\1033.

To work with resources in code, we use My.Resources together with the name of the resource file. Listing 24-29 shows how we use an icon resource in code.
Listing 24-29  Associate an Icon Resource File to a Windows Form

Me.Icon = My.Resources.PetrasIcon

In this example, the Me keyword refers to a Windows Form, and PetrasIcon refers to an icon resource file. The My keyword refers to the My namespace that the .NET Framework makes available for all VB.NET solutions. This namespace exposes seven objects that allow us to work with various resources and features. Table 24-3 lists the My namespace objects along with the purpose of each.

<table>
<thead>
<tr>
<th>Object</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>My.Application</td>
<td>Provides information about the application such as path, assembly information, and environment variables.</td>
</tr>
<tr>
<td>My.Computer</td>
<td>Provides features for manipulating computer components such as audio, the clock, the keyboard, the file system, and so on.</td>
</tr>
<tr>
<td>My.Forms</td>
<td>Provides access to all Windows Forms in the solution.</td>
</tr>
<tr>
<td>My.Resources</td>
<td>Provides access to resources used by the solution.</td>
</tr>
<tr>
<td>My.Settings</td>
<td>Allows reading and storing application configuration settings.</td>
</tr>
<tr>
<td>My.User</td>
<td>Provides access to information about the current user, including whether or not the user belongs to a special user group.</td>
</tr>
<tr>
<td>My.WebServices</td>
<td>Provides features for creating and accessing a single instance of each XML Web service referenced by the solution.</td>
</tr>
</tbody>
</table>

Retrieving Data with ADO.NET

Despite the similarity in the name, ADO.NET is something totally different from classic ADO on the unmanaged platform. For instance, it does not include a Recordset object, and the Excel CopyFromRecordset method is not supported. This is covered in more detail in Chapter 25, “Writing Managed COM Add-ins with VB.NET.” Another major difference is that ADO.NET has strong support for XML data representation. VS 2008 ships with version 3.5 of the ADO.NET class library.
ADO.NET is one of the default namespaces included in all Windows Forms based solutions, so to use it we just need to add Imports statements to the top of code modules from which ADO.NET will be called. However, to complicate things ADO.NET can be used in two different ways: **connected mode** and **disconnected mode**.

Before we can examine these two different approaches we need to first discuss **.NET Data Providers**. Data providers are used to connect to databases, execute commands, and provide us with the results. Each database, like SQL Server, Oracle, MySQL, and so on requires its own unique data provider. Some data providers are available by default in the .NET Framework, including SQL Server, Oracle, and OLE DB. Other data providers can be obtained from specific database vendors. For Microsoft Access and other databases that support ODBC, the OLE DB Data Provider can be used.

Connected mode means that we work with an open connection to the database. In this mode we explicitly use command objects and the **DataReader** object. A DataReader object retrieves a read-only, forward-only stream of data from a database. It can also handle multiple result sets. To do this, the connection must be open during the whole data retrieval process. Connected mode provides a performance advantage if we need to work with database records one at a time because the DataReader object retrieves and stores them in memory. However, the drawback is that connected mode creates more network traffic and requires having an active connection open during the whole database operation.

In Listing 24-30, we use a SQL Server database and therefore we import the namespace **System.Data.SqlClient**, which gives us access to the .NET Data Provider for SQL Server. We also use the ADO.NET class library and therefore we import the namespace **System.Data**.

### Listing 24-30 Using a DataReader Object

```vbnet
'At the top of the code module.
Imports System.Data
Imports System.Data.SqlClient

Friend Function Retrieve_Data_With_DataReader() As ArrayList

' SQL query in use.
Const sSqlQuery As String = _
    "SELECT CompanyName AS Company " & _
    "FROM Customers " & _
```
"ORDER BY CompanyName;"

'Connection string in use.
Const sConnection As String = _
  "Data Source=PED\SQLEXPRESS;" & _
  "Initial Catalog=Northwind;" & _
  "Integrated Security=True"

'Declare and initialize the connection.
Dim sqlCon As New SqlConnection(connectionString:= _
  sConnection)

'Declare and initialize the command.
Dim sqlCmd As New SqlCommand(cmdText:=sSqlQuery, _
  connection:=sqlCon)

'Define the command type.
sqlCmd.CommandType = CommandType.Text

'Explicitly open the connection.
sqlCon.Open()

'Populate the DataReader with data and
'explicit close the connection.
Dim sqlDataReader As SqlDataReader = _
  sqlCmd.ExecuteReader(behavior:= _
    CommandBehavior.CloseConnection)

'Variable for keeping track of number of rows in the
'DataReader.
Dim iRecordCounter As Integer = Nothing

'Get the number of columns in the DataReader.
Dim iColumnsCount As Integer = sqlDataReader.FieldCount

'Declare and instantiate the ArrayList.
Dim DataArrLst As New ArrayList

'Check to see that it has at least one
'record included.
If sqlDataReader.HasRows Then

  'Iterate through the collection of records.
  While sqlDataReader.Read

    For iRecordCounter = 0 To iColumnsCount - 1
'Add data to the ArrayList's variable.
DataArrLst.Add(sqlDataReader.Item(iRecordCounter).ToString())

Next iRecordCounter

End While
End If

'Clean up by disposing objects, closing and releasing variables.
sqlCmd.Dispose()
sqlCmd = Nothing

sqlDataReader.Close()
sqlDataReader = Nothing

sqlCon.Close()
sqlCon.Dispose()
sqlCon = Nothing

'Send the list to the calling method.
Return DataArrLst

End Function

We first create a SqlConnection object and then a SqlCommand object. Next we explicitly open the connection, create the DataReader object, and iterate through the collection of records in the DataReader object by using its Read method. Within the loop we populate an ArrayList object with the data from the DataReader object. Finally, we close and clean up the objects we’ve used and return the data in the ArrayList to the calling method. The Northwind database used in this example can be found on the companion CD in \Applications\Ch24 - Excel & VB.NET\Northwind.

When working in disconnected mode we make use of the DataAdapter, DataSet, and DataTable objects, which are supported by all .NET Data Providers. A DataAdapter acquires the data from the database and populates the DataTable(s) in a DataSet. The DataAdapter object includes commands to automatically connect to and disconnect from the database. It also includes commands to select, insert, update, and delete data. The DataAdapter object runs these commands automatically. The DataSet is an in-memory representation of the data, and like the DataReader object it can handle multiple SQL queries at the same time.
The advantages of using disconnected mode are that it creates less network traffic because it acquires the data in one go, and it does not require an open connection to the database once the data has been retrieved. It also allows us to first update the retrieved data and then return the updated data to the database.

Listing 24-31 shows a complete function, including SEH, which first creates the Connection object together with the DataAdapter object. It then creates and initializes a new DataSet. Next it initializes the DataAdapter object, which automatically establishes a connection, retrieves the data, and closes the connection. The DataSet is filled with the retrieved data and finally the function returns the first DataTable in the DataSet.

Listing 24-31 Using DataAdapter and DataSet Objects

'On top of the code module.
Imports System.Data
Imports System.Data.SqlClient

Friend Function Retrieve_Data_With_DataAdapter() As DataTable

' SQL query in use.
Const sSqlQuery As String = 
  "SELECT CompanyName AS Company " & _
  "FROM Customers " & _
  "ORDER BY CompanyName;"

' Connection string in use.
Const sConnection As String = 
  "Data Source=PED\SQL EXPRESS;" & _
  "Initial Catalog=Northwind;" & _
  "Integrated Security=True"

'Declare the connection variable.
Dim SqlCon As SqlConnection = Nothing

'Declare the DataAdapter variable.
Dim SqlAdp As SqlDataAdapter = Nothing

'Declare and initialize a new empty DataSet.
Dim SqlDataSet As New DataSet

Try
  ' Initialize the connection.
  SqlCon = New SqlConnection(connectionString:= _
sConnection)
'Initialize the DataAdapter.
SqlAdp = New SqlDataAdapter(selectCommandText:= _
_sSqlQuery, _
selectConnection:= _
SqlCon)

'Fill the DataSet.
SqlAdp.Fill(dataSet:=SqlDataSet, srcTable:="PED")

'Return the datatable.
Return SqlDataSet.Tables(0)

Catch Sqlex As SqlException
'Exception handling for the communication with
'the SQL Server Database.
'Tell it to the calling method.
Return Nothing

Finally

'Releases all resources the variable has consumed from
'the memory.
SqlDataSet.Dispose()

'Release the reference the variable holds and
'prepare it to be collected by the Garbage Collector
'(GC) when it comes around.
SqlDataSet = Nothing

SqlCon.Dispose()
SqlCon = Nothing

SqlAdp.Dispose()
SqlAdp = Nothing

End Try
End Function

The function returns a DataTable object from the ADO.NET class, but we
do not need to cast it into a DataTable object from the DataSet class before
returning it. The exception handler catches any exceptions that occur in
the SQL Server Data Provider. In the Finally block we dispose all object variables and set them to nothing. A working example of this solution can be found on the companion CD in \Concepts\Ch24 - Excel & VB.NET\Northwind folder.

ADO.NET may be a new technology for developers who are working with the .NET platform for the first time. But for Microsoft, the latest technology is .NET Language Integrated Query (LINQ), which is part of the .NET Framework 3.5 and was released with VS 2008. LINQ is a set of .NET technologies that provide built-in language querying functionality similar to SQL for accessing data from any data source. Instead of using string expressions that represent SQL queries, we can use a rich SQL-like syntax directly in our VB.NET code to query databases, collections of objects, XML documents, and more.

The future will tell us more about how well LINQ will succeed. Developers who are coming from classic ADO are more likely to first adopt ADO.NET and later perhaps also begin to use LINQ.

Further Reading

When it comes to the .NET Framework, VB.NET, and ADO.NET we have only scratched the surface. These technologies are all book-length topics in their own right. The following books are sources that we have found to be useful for a general introduction to VB.NET and to ADO.NET.

Programming Microsoft Visual Basic .NET Version 2003

Authored by Francesco Balena
ISBN# 0735620598—Microsoft Press
Unfortunately, this book has not been updated since VB.NET 2003 was released. However, it provides an excellent introduction to the .NET Framework and to VB.NET, as well as to other related technologies such as ADO.NET. It explicitly targets Classic VB developers who are moving to the .NET platform.

Visual Basic 2008 Programmer’s Reference

Authored by Rod Stephens
ISBN# 0470182628—Wrox
This book offers a light introduction to VB.NET that explicitly targets beginning to intermediate level developers. This is a practical book about the .NET Framework, VS IDE, and VB.NET, written well in plain English. The only thing that may be annoying is that some screen shots are oversized. Hopefully this will be corrected in later editions of the book.

**Additional Development Tools**

The authors have no financial interest in these tools and are not connected to their vendors. The recommendations are based on our own daily use of these tools as .NET developers.

**MZ-Tools**

MZ-Tools 6.0 is an add-in to the VS IDE. It works with all current versions of VS.NET except for the Express edition. It adds many tools and functions to the VS IDE that are designed to simplify development work and increase productivity. For more information see www.mztools.com.

**VSNETCodePrint**

VSNETCodePrint 2008 is an add-in to the VS IDE that helps developers document their solutions. With this tool we can print, preview, and export a complete solution, selected projects, project items, classes, modules, and procedures in several file formats. It can save you a significant amount of time when you need to document solutions and inspect code. For more information see www.starprint2000.com.

It should be noted that MZ-Tools provides features to generate documentation using either HTML or XML file formats that overlap the features in VSNETCodePrint to some degree but are less advanced.

**Q&A Forums**

There are many general public VB.NET Q&A forums, but the Microsoft MSDN section for VB.NET is one of the best at http://forums.msdn.microsoft.com/en-US/tag/visualbasic/forums/. The VB.NET section at Xtreme VB Talk is also good, and it includes a subforum for .NET Office automation at www.xtremevbtalk.com/forumdisplay.php?f=97.
Practical Example—PETRAS Report Tool .NET

PETRAS Report Tool .NET is a practical case study that demonstrates a more complex VB.NET application than is possible to cover in a single chapter. In Chapter 25, the tool is converted into a managed COM add-in for Excel. The tool is a standalone, fully functional reporting solution. It retrieves data from a SQL Server database (created in Chapter 19, “Programming with Access and SQL Server”) based on the user selection in the main Windows Form. It then populates predefined Excel report templates with the data. It can export reports either to Excel or to XML files. The solution can be found on the companion CD in Applications\Ch24 - Excel & VB.NET\PETRAS Report Tool.NET. Please read the Read Me First.txt file located in the Applications\Ch24 - Excel & VB.NET\ folder. You will find it helpful to open this solution in the VB IDE so that you can reference it while reading this section.

When the tool starts up, it first tries to establish a connection to the database. A custom Windows Form is displayed while the tool is trying to connect. If the connection attempt is successful, the main Windows Form shown in Figure 24-22 is displayed. If the connection attempt fails, an error message is displayed.

Use the following steps to create a report in the main form:

![PETRAS Report Tool .NET user interface](image-url)

**Figure 24-22** PETRAS Report Tool .NET user interface
1. Select a Client.
2. Select a Project.
3. Select the reporting time period by entering a Start date and an End date.
4. Uncheck or keep the fields Activities and Consultants.
5. Click on the Create Report button to preview the report in the DataGrid.
6. Click the appropriate button to export to an Excel report or to an XML file.
7. If export to Excel is selected, Excel is launched and a copy of one of the four predefined report templates is created.
8. If export to XML is selected, a Save File dialog is displayed so you can specify a filename and location where the XML file should be saved.
9. If the export is successful, the selections you made become the new default values for all controls on the Windows Form. It is possible to clear these settings by selecting the Clear Settings button.
10. To close the Windows Form, click the Close button.

The .NET Solution

Although we only use one main Windows Form, our .NET solution includes some additional modules and files. Table 24-4 shows a summary of what the solution contains.

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Type and Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>app.config</td>
<td>XML configuration file containing the connection string</td>
</tr>
<tr>
<td>frmConnecting.vb</td>
<td>Windows Form displayed while connecting to the database</td>
</tr>
<tr>
<td>frmMain.vb</td>
<td>Windows Form that is the main form for the solution</td>
</tr>
<tr>
<td>MCommonFunctions.vb</td>
<td>Standard module containing general functions for the tool</td>
</tr>
<tr>
<td>MDataReports.vb</td>
<td>Standard module containing all database functions</td>
</tr>
<tr>
<td>MExportExcel.vb</td>
<td>Standard module containing all the functions required to export data to Excel</td>
</tr>
<tr>
<td>MExportXML.vb</td>
<td>Standard module containing all the functions required to export data to XML files</td>
</tr>
</tbody>
</table>
As you can see in Table 24-4, the solution does not include any class modules. Creating well-designed class modules is covered in Chapter 25. In addition to the components shown in Table 24-4, the solution uses four different Excel report templates. Depending on the user selections, one of them is used to create the requested report:

- **PETRAS Report Activities.xlt**—Used when only the Activities control is checked
- **PETRAS Report Activities Consultants.xlt**—Used when both the Activities and Consultants controls are checked
- **PETRAS Report Consultants.xlt**—Used when only the Consultants control is checked
- **PETRAS Report Summary.xlt**—Used when neither the Activities nor the Consultants controls are unchecked

If we click the *Show All Files* button in the Solution Explorer toolbar, it displays an expanded tree view. If we then expand the References item in the tree view we can see all references for the solution, as shown in Figure 24-23. Most hidden files are system files that we rarely need to work with, but it’s a good exercise to explore all the files included in the solution.

In any non-trivial real-world application where we initially load a Windows Form, we usually need to ensure that certain conditions are met before loading it. In VB.NET we can use the same approach as with Classic VB. We create a `Main` subroutine in a standard code module that is used as the startup subroutine.

But in VB.NET, we need to change some additional settings in the solution before this will work correctly. After creating the new Windows Forms application, open the solution Properties window, and select the *Application* tab. Figure 24-24 shows the original startup settings for the PETRAS Report Tool.NET solution.

We add a standard code module to the solution that we name MStartup.vb. We add the `Main` subroutine and its code to this module, as shown in Listing 24-32.
Sub Main()
    'Enable Windows XP’s style.
    Application.EnableVisualStyles()

    'Declare and instantiate the Windows Form.
    Dim frm As New frmMain

    'Set the position of the main Windows Form.
End Sub
frm.StartPosition = FormStartPosition.CenterScreen

' Show the main Windows Form.
Application.Run(mainForm:=frm)

' Releases all resources the variable has consumed from
' the memory.
frm.Dispose()

' Release the reference the variable holds and prepare it
' to be collected by the Garbage Collector when it
' comes around.
frm = Nothing
End Sub

Now we return to the Application tab of the solution Properties window, where we uncheck the option Enable application framework and change the Startup object to the Main subroutine as shown in Figure 24-25.

Unchecking the Enable application framework option implicitly removes the option to use Windows XP styles. Therefore, we need enable this option manually in the startup code, which is done in the first line of our Main procedure in Listing 24-32.

The Main subroutine is also a good place to put code to position the Windows Form before it is loaded. The Main subroutine is also an acceptable place to put code for connecting to a database, but in the PETRAS Report Tool.NET we use a different approach that is covered soon. When the user closes the main Windows Form we dispose its class and set the variable to nothing.

**Windows Forms Extender Providers**

The .NET Framework provides so-called extender providers to Windows Forms. These components can only be used with visual controls.
By adding them to our Windows Forms we get additional properties to work with. Extender providers are added to a Windows Form in exactly the same way as regular controls. However, the extender providers appear in the form’s Component Tray rather than on the surface of the form itself.

Figure 24-26 shows the Component Tray for the main form of the PETRAS Report Tool.NET. The components used are the ErrorProvider, HelpProvider, and ToolTip components, for the main Windows Form, the BackgroundWorker component, which we cover later, and the SaveFileDialog component that was introduced earlier in the chapter.

The first extender provider in use is the ErrorProvider, which provides us with the option to set validation errors. It can be used with one or more controls on the Windows Form as each of them have the Validating event.

When a control’s input is not valid the ErrorProvider places an error icon next to the control and displays an error message when the user hovers the mouse over the icon. Listing 24-33 shows how this is implemented in the PETRAS Report Tool.NET solution. As the code shows, we can create a single event that hooks the Validating events of all the targeted controls on the form.

Listing 24-33  The Validating Event Subroutine for Several Controls

```vbnet
Private Sub Client_Project_Validating(ByVal sender As Object, _
    ByVal e As System.ComponentModel.CancelEventArgs) _
Handles cboClients.Validating, _
    cboProjects.Validating
    Const sMESSAGECLIENTERROR As String = _
        "You need to select a client."
    Const sMESSAGEPROJECTERROR As String = _
        "You need to select a project."
    Dim Ctrl As Control = CType(sender, Control)
    If Ctrl.Text = "" Then
```

FIGURE 24-26  Extender providers in the PETRAS Report Tool.NET
Select Case Ctrl.Name

Case "cboClients"
    Me.ErrorProvider1.SetError(control:=Ctrl, _
    value:=sMESSAGECLIENTERROR)

Case Else
    Me.ErrorProvider1.SetError(control:=Ctrl, _
    value:=sMESSAGEPROJECTERROR)

End Select

Else

    Me.ErrorProvider1.SetError(control:=Ctrl, value:="")

End If

End Sub

If one of the controls being validated has the focus when the user clicks the Clear Settings button, the validation handling code is executed. To prevent this we must add one line of code to the load event of the main Windows Form. This is shown in Listing 24-34.

Listing 24-34  Code to Prevent Validation when the Clear Settings Button Is Clicked

Me.cmdClearSettings.CausesValidation = False

We can prevent the entry of bad data into a control by writing handlers for the key press event as well.

Looking more closely at the code in Listing 24-33 may raise the question of why we do not use a control array as we would in Classic VB. This is because VB.NET does not currently support control arrays, and it does not appear as if this feature will be implemented in any future version. The solution shown is the closest workaround in VB.NET. The second extender provider, HelpProvider, is used to associate a help file (either a .chm or .htm file) with our application. Whenever our application is running and has focus, the HelpProvider associates the F1 button with our application’s help
file. For the PETRAS Report Tool.NET we use a simple form-based help system, meaning that we associate the help file with our main Windows Form. It is much easier to set this up using Windows Form properties manually at design time than to do it at runtime with code. The design-time property settings required to create a form-based help system are the following:

- Set the HelpKeyword property on HelpProvider1 to the value About.htm.
- Set the HelpNavigator property on HelpProvider1 to the value Topic.

One property of the HelpProvider that should be set in code is the HelpNameSpace property. Doing this provides us with a more flexible solution because we can change the location of the help file dynamically. Listing 24-35 shows the code in the main Windows Form load event required to set the HelpNameSpace property.

Listing 24-35  Setting the Path and Name to the Help File

```
'Setting the help file in use.
Const sHELPNAMESPACE As String = "PETRAS_Report_Tool.chm"

'Setting the helpfile to the HelpProvider component.
Me.HelpProvider1.HelpNamespace = swsPath + sHELPNAMESPACE
```

The swsPath is a global enumeration member that holds the path to the application EXE file for the PETRAS Report Tool.NET.

The third extender provider is the ToolTip component. It provides us with the option to add a ToolTip to each control in a Windows Form. Whenever the user hovers over a control with the mouse the control’s ToolTip is displayed.

**Threading**

With .NET we can leverage multithreading to create more powerful solutions. It is beyond the scope of this chapter to cover multithreading in detail, but we demonstrate a simple example. The .NET Framework includes an extender provider, BackgroundWorker, which allows us to run code on a separate, dedicated thread, meaning we can run our project in multithreading mode. This extender provider is normally used for time-consuming operations, but as this case shows, we can use it for other tasks as well.
In the PETRAS Report Tool.NET, we use the BackgroundWorker component to run the code that connects to the database. By using two of its events, BackgroundWorker1_DoWork and BackgroundWorker1_RunWorkerCompleted, we attempt to connect to the database in the background and be notified about the outcome. Listing 24-36 shows the code for the load event of the main Windows Form followed by the code for the two events of the BackgroundWorker component.

Listing 24-36  Code in Use for the BackgroundWorker

```vbnet
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    "...

    'Settings for the BackgroundWorker component.
    With Me.BackgroundWorker1
        'Makes it possible to cancel the operation.
        .WorkerSupportsCancellation = True
        'Start the background execution.
        .RunWorkerAsync()
    End With

    'Change the cursor while waiting to BackgroundWorker component has been finished.
    Me.Cursor = Cursors.WaitCursor
End Sub

Private Sub BackgroundWorker1_DoWork(ByVal sender As Object, ByVal e As System.ComponentModel.DoWorkEventArgs) Handles BackgroundWorker1.DoWork
    'Instantiate a new instance of the connecting Windows Form.
    mfrmConnecting = New frmConnecting

    'Position the Windows Form and display it.
    With mfrmConnecting
        .StartPosition = FormStartPosition.CenterScreen
        .Show()
    End With
End Sub
```
'Can we connect to the database?
If MDataReports.bConnect_Database() = False Then

    'OK, we cannot establish a connection to the
database so we cancel the background operation.
Me.BackgroundWorker1.CancelAsync()

    'Let us tell it for the other backgroundWorker
    'event - the RunWorkerCompleted.
    mbIsConnected = False

Else

    'Let us tell it for the other backgroundWorker
    'event - the RunWorkerCompleted.
    mbIsConnected = True

End If

'Close the connecting Windows Form.
mfrmConnecting.Close()

'Releases all resources the variable has consumed
'from the memory.
mfrmConnecting.Dispose()

'Release the reference the variable holds and prepare
'it to be collected by the Garbage Collector (GC) when
'it next time comes around.
mfrmConnecting = Nothing

End Sub

Private Sub BackgroundWorker1_RunWorkerCompleted_
(ByVal sender As Object, ByVal e As System.ComponentModel.RunWorkerCompletedEventArgs)
Handles BackgroundWorker1.RunWorkerCompleted

    'If we have managed to connect to the database then we can continue.
    If mbIsConnected Then

        '...

On its surface, the use of the BackgroundWorker component may look attractive. However, multithreaded application development is complex and easy to get wrong, so it should only be used in situations where it is absolutely necessary to run code outside the main process.

**Retrieving the Data**

A database connection string can be created using several different methods. For the PETRAS Report Tool.NET we create a solutionwide connection string using an application setting. This is accomplished in the *Settings* tab of the solution Properties windows, as shown in Figure 24-27.

![Figure 24-27](image)

We first create a name for the setting and then select the type (*Connection string*). The scope is now automatically set to *Application*. After placing the...
cursor in the Value field a button appears on the right side. Clicking this button displays a very useful built-in wizard for creating connection strings.

If we look in the Solution Explorer window, we notice that a new app.config XML file has been created and added to the solution. The app.config file will not be compiled into the executable file when we develop standalone applications like the PETRAS Report Tool.NET. Instead, it is a separate XML file that is installed alongside the PETRAS Report Tool.NET executable. This allows us to easily update the connection string by simply opening and editing the XML file. When we compile the solution the VS IDE creates an XML file based on the solution name, PETRAS Report Tool .NET.exe.xml, for example, instead of using the name app.config.

When creating a DLL, the app.config file is compiled into the DLL, which makes it more difficult to update the connection string. This is addressed in Chapter 25. Listing 24-37 shows how to read the connection string setting from within our application code.

### Listing 24-37 Reading the Application Setting for the Connection String

```vbnet
'Read the connection string into a module variable.
Private ReadOnly msConnection As String = _
    My.Settings.SQLConnection.ToString()
```

Next we use it to initialize a new SqlConnection object, as shown in Listing 24-38.

### Listing 24-38 Function to Create New SqlConnection

```vbnet
Friend Function sqlCreate_Connection() As SqlConnection

    Return New SqlConnection(connectionString:=msConnection)

End Function
```

All functions that retrieve data using disconnected mode expect the DataSet object to contain one DataTable at the time. We use a module-level DataTable variable to populate the DataGridView control. If the user decides to either create an Excel report or export the data to an XML file, the same DataTable is used as an argument to one of the export functions.
Exporting Data

The MExportExcel.vb module contains all the functions required to export data to Excel using one of the four predefined Excel templates described earlier. The main export function, shown in Listing 24-39, takes several arguments. Since the query has already been executed we can get the results as a DataTable from the DataGridView control on the main Windows Form. The other arguments provide information about the options specified by the user when the data was retrieved from the database.

**Listing 24-39  The Main Export to Excel Function**

```vbnet
Friend Function bExport_Excel(_
    ByVal dtTable As DataTable, _
    ByVal sClient As String, _
    ByVal sProject As String, _
    ByVal sStartDate As String, _
    ByVal sEndDate As String) As Boolean
```

Because the PETRAS Report Tool.NET is a standalone application not related to Excel, we first need to determine whether Excel exists and if so, determine which version of Excel is available. To accomplish this we examine the value of a critical Excel-related registry entry and use it to determine the current Excel version.

The lowest version of Excel that we can support is version 2002, meaning the tool cannot be used if version 2000 is installed. The function uses an enumeration of Excel versions, which is defined in the MSolutionsEnumerationsVariables.vb code module. To provide access to the .NET Framework functions that allow us to read the Windows registry, we import the namespace Microsoft.Win32. We also use regular expressions to complete this task, so the namespace System.Text.RegularExpressions also is imported into the code module. Listing 24-40 shows the code for the function.

**Listing 24-40  Determine Which Version of Excel Is Available**

```vbnet
' At the top of the module.
' To read the Windows Registry subkey.
Imports Microsoft.Win32
' To use regular expressions.
Imports System.Text.RegularExpressions
```
Friend Function shCheck_Excel_Version_Installed() As Short

Const sERROR_MESSAGE As String = 
   "An unexpected error has occurred " + 
   "when trying to read the registry."

'The subkey we are interested in is located in the
'HKEY_CLASSES_ROOT Class.
The subkey's value looks like the following:
'Excel.Application.10
Const sXL_SUBKEY As String = "\Excel.Application\CurVer"

Dim rkVersionkey As RegistryKey = Nothing
Dim sVersion As String = String.Empty
Dim sXLVersion As String = String.Empty

'The regular expression which is interpreted as:
'Look for integer values in the interval 8-9
'in the end of the retrieved subkey's string value.
Dim sRegExpr As String = "^[8-9]$"

Dim shStatus As Short = Nothing

Try
   'Open the subkey.
   rkVersionkey = Registry.ClassesRoot.OpenSubKey 
      (name:=sXL_SUBKEY, writable:=False)

   'If we cannot open the subkey then Excel is not available.
   If rkVersionkey Is Nothing Then
      shStatus = xlVersion.NoVersion
   End If

   'Excel is installed and we can retrieve the wanted
   'information.
   sXLVersion = CStr(rkVersionkey.GetValue(name:=sVersion))

   'Compare the retrieved value with our defined regular
   'expression.
   If Regex.IsMatch(input:=sXLVersion, pattern:=sRegExpr) Then
      'Excel 97 or Excel 2000 is installed.
      shStatus = xlVersion.WrongVersion
   Else

   'An unexpected error has occurred when trying to read the registry.
'Excel 2002 or later is available.
shStatus = xlVersion.RightVersion
End If

Catch GeneralExc As Exception

'Show the customized message.
MessageBox.Show(text:=sERROR_MESSAGE, _
caption:=swsCaption, _
buttons:=MessageBoxButtons.OK, _
icon:=MessageBoxIcon.Stop)

'Things didn't work out as we expected so we set the
'return variable to nothing.
shStatus = Nothing

Finally

If rkVersionKey IsNot Nothing Then

'We need to close the opened subkey.
rkVersionKey.Close()

'Release the reference the variable holds and prepare it
'to be collected by the Garbage Collector (GC) when it
'comes around.
rkVersionKey = Nothing
End If

End Try

'Inform the calling procedure about the outcome.
Return shStatus

End Function

The module MExportExcel.vb also contains a function to verify that the Excel templates exist in the same folder as the executable file.

The function that exports data to an XML file also creates the Schema file for it. Listing 24-41 shows the two lines of code required to generate
these files. We actually use the methods of the DataTable object to generate the XML files. This is because ADO.NET uses XML as its underlying data representation scheme. Both of these XML files can be opened and studied in more detail.

Listing 24-41  Creating XML and Schema Files

```vbnet
... 'Write the data to the XML file.
  dtTable.WriteXml(fileName:=sFileName)

  'Create the Schema file for the XML file.
  dtTable.WriteXmlSchema(fileName:=Strings.Left(
    _
    sFileName, Len(sFileName) - 4) & ".xsd"
  )
...```

Summary

In this chapter, we provided a brief introduction to the .NET Framework, VB.NET, data access using ADO.NET, and Excel automation from VB.NET. Compared to Classic VB, the .NET Framework is a completely new and different platform. It is also a modern, advanced development platform with a great set of tools for creating user-friendly solutions. To fully utilize the .NET platform you must be prepared to invest significant time exploring and learning it. As we all know, there are no real shortcuts to learning new technology. Only hard work can accomplish the task. But the reward, in addition to the new knowledge itself, is that we can leverage all the knowledge from this chapter in the two chapters that follow.
INDEX

Symbols
# character prefix (conditional compilation constants), 512
(:, colon character in Immediate window, 520
 . (dot operator), performance and, 571
= (equal sign) in criteria ranges, 677
<, > (greater than/less than symbols) in criteria ranges, 677
\ (integer division operator), 570
? (question mark character) in Immediate window, 519
3D effects, simulating, 84

A
accelerator keys. See also keyboard shortcuts
creating, 205
for UserForm controls, 386
Access 2002 Desktop Developer's Handbook (Litwin, Getz, Gunderloy), 739
Access 2002 Developer's Handbook Set (Litwin, Getz, Gunderloy), 647
Access databases
adding data (time sheet example application), 652-656
advantages of, 620
connecting to, 620-622
time sheet example application, 648-649
deleting data, 629-630
inserting data, 625-626
modifying data, 626-629
Northwind sample database, installing, 615
retrieving data, 622-625
time sheet example application, 650-652
upsizeing to SQL Server, 642-646
Access object library, 726-729
Application object, 726
DAO.Database object, 726
DoCmd object, 727
event application, 727-729
access restrictions, checking network group membership, 1095-1096
accessing Application object from automation add-ins, 800-802
action panes, 999
Activate event, error handling, 489
activating error handlers, 468
active, error handlers as, 468
Active Directory Service Interfaces object library, 1095
ActiveConnection property (ADO Command object), 605, 628
ActiveDocument (Word), referencing, 712
ActiveSheet property, performance and, 573
ActiveX, 710
ActiveX controls
adding to Windows Forms, 826
advantages of, 100
forms (VB6) support for, 760
ActiveX Data Objects. See ADO
ActiveX DLLs, 742
advantages of using, 758-774
Clipboard object, 773
code protection, 758
forms (VB6) versus UserForms, 759-762, 764-769
object oriented programming support, 769-772
Printer object, 773
resource files, 773
Screen object, 774
COM add-ins. See COM add-ins
compiling, 744, 750
form display example, 751-758
in-process communication, 774
loading icons with resource file, 802-807
one-way communication example, 744-747
projects, creating, 742-744
referencing, 745-746
registering, 744
setting references, 747
two-way communication example, 747-751
adAsyncExecute
(ExecuteOptionEnum constant value), 603
adCmdStoredProc
(CommandTypeEnum constant value), 603
adCmdTable
(CommandTypeEnum constant value), 603
adCmdTableDirect
(CommandTypeEnum constant value), 603

1107
<table>
<thead>
<tr>
<th>Add-in Designer</th>
<th>785-786</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advanced tab</td>
<td>789</td>
</tr>
<tr>
<td>General tab</td>
<td>788-789</td>
</tr>
<tr>
<td>registry key management</td>
<td>788-789</td>
</tr>
<tr>
<td>registry keys</td>
<td>1102</td>
</tr>
</tbody>
</table>

Add-in Express for Microsoft Office and .NET, 962-963

Add-in Manager
- managed automation add-ins, selecting, 938-940
- registry keys, 1100
- XLLs and, 1042-1043

Add-ins
- application-specific, 18-19, 118-125

application, 125-137
- automation and, 782
- COM add-ins. See COM add-ins

Add-inSpy, 963

AddRange method (VB.NET arrays), 844

AddInstance object events, 792-794
- Initialize, 792
- OnAddInsUpdate, 794
- OnBeginShutdown, 794
- OnConnection, 792-793
- OnDisconnection, 794
- OnStartupComplete, 793
- Terminate, 794

AddInSpy, 963

AddRange method (VB.NET arrays), 844

adExecuteNoRecords (ExecuteOptionEnum constant value), 603, 626

adLockBatchOptimistic (LockTypeEnum constant value), 611

adLockOptimistic (LockTypeEnum constant value), 611

adLockPessimistic (LockTypeEnum constant value), 611

AddRange method (VB.NET arrays), 844

AddInstance object events, 792-794
- Initialize, 792
- OnAddInsUpdate, 794
- OnBeginShutdown, 794
- OnConnection, 792-793
- OnDisconnection, 794
- OnStartupComplete, 793
- Terminate, 794

AddInSpy, 963

AddRange method (VB.NET arrays), 844

adExecuteNoRecords (ExecuteOptionEnum constant value), 603, 626

adLockBatchOptimistic (LockTypeEnum constant value), 611

adLockOptimistic (LockTypeEnum constant value), 611

adLockPessimistic (LockTypeEnum constant value), 611

ADO (ActiveX Data Objects), 598

Command object, 605-607
- ActiveConnection property, 605, 628
- ActiveConnection property, 605, 628
- CommandText property, 605
- CommandType property, 605, 628
- CreateParameter method, 605-606, 628

ADO.NET, 864-870

connected node, 865-867
- data providers, 865

ADO.NET, 864-870

connected node, 865-867
- data providers, 865
Alt key, checking state of, 350-351
application architecture, 18-19
application-specific add-ins, 18-19
API calls, 342-343
API calls, 342-343
API calls, 331
Alias clause, 342-343
buffers, filling, 360
constants, finding value of, 333-334
declarations, finding, 333
Declare statement, 331-332
documentation, finding, 332
encapsulating, 335-337
file system-related functions, 355
deleting files to Recycle Bin, 360-361
folders, browsing for, 361-371
special folders, locating, 357-360
UNC paths, changing to, 356-357
user ID, finding, 355-356
handles, 334-335
keyboard-related functions, 349
key presses, testing for, 352-355
key states, checking, 350-351, 371-373
passing strings to, 356
screen-related functions, 337
pixel size, determining, 338-340
screen resolution, reading, 337-338
window-related functions, 340
messages, sending, 346-348
related windows, finding, 343-346
window classes, 341
window icons, changing, 348-349
window styles, changing, 349
windows, finding, 342-343
App.config XML file, 883
AppDomain, 953
application architectures
application-specific add-ins, 18-19
best practices, 40-42
logical tiers, separating, 41-42, 616-617
codeless applications, 14-15
compared, 24-25
dictator applications, 20-23
debugging, 22
requirements of, 21-22
general purpose add-ins, 16-18
selecting, 13, 24-25
self-automated workbooks, 15-16
technical implementations
of, 23
application communication
in-process, 774
out-of-process, 774
application contexts, 244-247
application control. See controlling applications
application development platform, Excel as, 4-7
Application Domain, 953
application instances. See instances (of applications)
application list (Add-in Designer), 789
application manifest, 992, 1017
Application object (Access object model), 726
Application object (Excel object model)
accessing from automation add-ins, 800-802
in VSTO add-ins, 986-987
in VSTO workbooks, 1010
Calculation property, 571
Cursor property, 520
EnableCancelKey property, 22
EnableEvents property, 528
IgnoreOtherApplications property, 21
IgnoreRemoteRequests property, 149
MacroOptions method, 112
retrieving reference to, 930
Run function, 109
ScreenUpdating property, 571
ShowWindowsInTaskBar property, 21
Version property, 142
MacroOptions method, 112
retrieving reference to, 930
Run function, 109
ScreenUpdating property, 571
ShowWindowsInTaskBar property, 21
Version property, 142
application properties, setting to default values, 130
application versions (Add-in Designer), 789
application-centric project templates, 977-979
application-level event handling (time sheet example application), 138
opening and initializing (time sheet example application), 125-127
organization (time sheet example application), 138
Packaging, 1099
add-in installation requirements, 1100-1102
installation location, selecting, 1099-1100
installation with Windows Installer, 1104
manual installation, 1103
Setup.xls workbook installation, 1103
template installation requirements, 1100
separating from data with XML, 254-256
stages of, 107-110
development/maintenance stage, 107-108
runtime stage, 109
shutdown stage, 110
startup stage, 108-109
AppointmentItem object (Outlook object model), 736
architectures.
See application architectures
arrays
control arrays, 761-769, 878
index arrays, 566-567
looping, 57
redimensioning, 563-565
searching, binary searches, 563-565
sorting
combining with binary searches, 565
QuickSort procedure, 560-563
Variant, 55
performance and, 572-573
VB.NET, 835, 839-845
watching, 529-531
XLOPER data type and, 1048-1049
artificial keys, 592-594
As New syntax, 55
As Object declarations, 569
ASCII values, changing, 779
asking questions (creative thinking), 556
ascensions, 820. See also DLLs
code-behind assemblies, 979-980
referencing in VSTO add-ins, 991-992
Assembly Information dialog, 902
Assembly Registration tool, 900
assembly version numbers, 902
AssemblyInfo.vb file, 902
 AssemblyLocation custom document property, 980
AssemblyName custom document property, 980
assertions, 540-541
assigning values. See initializing
associating icons with controls, 213-215, 228-237, 796, 802-807, 863-864
macros with shapes, 88
asynchronous database connections, 602
atomic values, defined, 581
attributes (XML), 252
Authenticode certificates, 953, 960-961
auto-completion of variable names, 31
auto-expanding charts, 692-694
auto-generated references for managed COM add-ins, 897-899
auto-instantiation, 379, 770
AutoDual type (ClassInterface attribute), 930
automatic calculation, disabling, 571
automation, 709-710, 775-783
add-ins and, 782
front loaders, 782-783, 808-815
managed automation add-ins creating, 928-933
limitations of, 933-940
.NET Framework and, 855-863
early binding/late binding, 863
Excel object usage, 857-862
PIA (Primary Interop Assembly), 856-857
Roman numeral conversion example, 775-782
VSTO automation add-ins, 1006
XLLs and, 1061
automation add-ins, 799-802
accessing Application object, 800-802
calling, 800
creating, 799-800
installation requirements, 1102
automation clients, 710
Automation Server, 929
automation servers, 710
availability of applications, determining, 722
axes
creating complex, 701-702
multiple axes, 690
axis scales, calculating, 706-708
B
Backdrop application context, 244
backgrounds
preparing for user interface, 151-153
transparent backgrounds for icons, setting, 214-215
BackgroundWorker component, 877-880
backing up project versions, 66
backward compatibility, 319.
See also cross-version applications of application instances, 722
maintaining, 62
object libraries, 713-715
VB.NET, 820
BDC (Business Data Catalog), 976
Before setting (command bar definition table), 215, 221
BeforeUpdate events, 388
Begin Group setting (command bar definition table), 215
best practices
application architecture, 40-42
logical tiers, separating, 41-42, 616-617
change control, 65-67
for circular references, 685
code comments, 45-49
internal comments, 47-49
module-level comments, 46
procedure-level comments, 46-47
updating, 49
code readability, 50-52
command bar design, 198-199
controlling applications, 711-714
application-specific class modules, 714
development for earliest supported version, 713
property/method calls, 712-713
variable declarations, 711-712
deleting toolbars, 127
naming conventions, 27-29, 31-40
defined names, 39
embedded objects, 38
Excel UI elements, 37-39
exceptions to, 39-40
modules, classes,
UserForms, 36
procedures, 35-36
sample of, 28-34
shapes, 37-38
Visual Basic Projects, 37
worksheets, chart sheets, 37
for On Error Resume Next statement, 471
for procedural programming, 43-45
arguments, limiting, 45
duplicate code, eliminating, 44
capsulation, 44
functional decomposition, 43
modules, organizing code in, 43
size limits on procedures, 44
Ribbon UI design, 278-289
Add-Ins tab, 279
control custom image management, 284-286
global callback handlers, 286-287
validation, 287-289
keytips, 284
shared tabs, 279-284
work processes support, 278-279
UserForms design, 375-384
  business logic, separating from, 376-379
  classes versus default instances, 379-381
  properties and methods, exposing, 382-384
  simplicity, 375-376
VBA programming, 52-65
  defensive coding, 62-65
  module directives, 52-53
  variables and constants, 54-61
wizard dialog design, 407-408
BI (Business Intelligence), 976
  binary file format, ZIP archives versus, 274
  binary searches, 563-565
  BinarySearch method, VB.NET arrays, 842
binding. See data binding
bit masks, 351
  adding to resource files, 804-806
  creating for icons, 228-230
  loading from resource files, 806-807
blank lines in code, 50-51
block scope (VB.NET), 835-836
blogs
  managed COM add-in information, 962
  VSTO information, 1026
BOF property (ADO Recordset object), 607
Bookmark object (Word object model), 729
bookmarks, populating, 729
Bookmarks.dot, 723
Boolean variables, redundant comparisons of, 571
bootstrapper packages, 904
borders, 84
time sheet example application, 104
Break in Class Module setting, 508
break mode
  error trapping settings, 507-508
  run mode versus, 507
  Break on All Errors setting, 507
  debug mode versus, 509
  Break on Unhandled Errors setting, 508
  break points, setting/removing, 512-513
  Break When Value Changes setting (Watch Type options), 528
  Break When Value Is True setting (Watch Type options), 527-528
  breaking the rules (creative thinking), 554-555
  Breakpoints window, 849-850
  breaks in axes scales, creating, 702
  browsing for folders, 361-369
time sheet example application, 369-371
  buffers, 356
  filling, 360
Business Data Catalog (BDC), 976
Business Intelligence (BI), 976
  business logic
  isolating, 44
  separating from UserForms design, 376-379
  business logic tier
  data access tier, relationship with, 616
  defined, 41
  business systems, 976
ByRef
  arguments, 62
  ByVal versus (performance optimization), 554
  passing strings, 569
  in VB.NET, 838
byte-counted strings, 1036, 1047
ByVal
  arguments, 62
  ByVal versus (performance optimization), 554
  passing strings, 569
  in VB.NET, 838
C
C API
  functions in XLLs, 1052-1053
  object oriented C++ wrapper for, 1063
  XLLs. See XLLs
C strings, 1047
C++ keywords, XLOPER data type and, 1061
C-strings, 1036
calculated fields, 670-672
calculated items, 670-672
calculated property (Application object module), 571
calculations. See data processing
call stack, 465-468, 475, 485-488, 496-499, 521
  Call Stack window, 521-522, 830
callback functions in XLLs, 1040-1044
  xAddlnManagerInfo, 1042-1043
  xAutoAdd, 1044
  xAutoClose, 1041-1042
  xAutoFree, 1044
  xAutoOpen, 1040-1041
  xAutoRegister, 1043
  xAutoRemove, 1044
callback handlers, 286-287
  for control custom image management, 285
  for shared tabs, 281-283
callback procedures for light weight UI design, 307
callbacks, 363-368
calling default object properties, 63
canceling program execution, 484-485, 491-495
capacity constraints, reasons for using databases, 578
Caps Lock key, checking state of, 350-351
captions, adding to toolbar buttons, 224
CAS (Code Access Security), 1017
cascading lists for data validation, 90-92
case studies. See time sheet example application
casting interfaces, 447
object references, 859
Catch statement (structured exception handling), 830
catching errors, 480
category entry (XLL function table), 1038
category numbers for UDFs, 113
CCell class module, 166-167
methods, 169-170
adding, 172
property procedures, 169
CCells class module, 173
WithEvents object variable, declaring, 177
CControlEvents class module, 238-240
cell comments
as help text, 86-87
time sheet example application, 104
cells
changing with UDFs, 116
data input cells, clearing all, 124-125, 134
positioning UserForms, 400-402
ranges of, reading/writing, 572-573
central error handler, 481-488
in error handling demo program, 494-496
time sheet example application, 496-506
Certificate Authority, 1097
change control, 65-67
change documentation with code comments, 66
chart items, determining positional information, 704-706
chart sheets
naming conventions, 37
referencing, 65
chart types, combining, 687-690
charts
axis scales, calculating, 706-708
chart types, combining, 687-690
complex axes, creating, 701-702
cordinate systems, converting among, 702-704
defined names in auto-expanding charts, 692-694
plotting functions, 696-698
scrolling/zooming in time series, 694
SERIES function, 691-692
setting up links, 691-692
transforming coordinate systems, 694-696
in dictator applications, 157
displaying on UserForms, 397-398
multiple axes in, 690
positional information, determining, 704-706
in PowerPoint, 733
step charts, creating, 699-701
CheckBoxList controls, 844
checksum formulas, sum of digits calculation in, 682
.chm files, 1086
circular references, 683-686
class modules (VBA), 443. See also interfaces; objects
application-specific class modules, 714
Break in Class Module setting, 508
data access tier, creating for, 617-620
encapsulation, 172
events
application-level event handling, 190-193
raising, 189-188
trapping, 177-182
Implements keyword, 438-440
interfaces. See interfaces naming conventions, 36
object creation with, 166-168
organizing code in, 43
polymorphism, 443-448
purpose of, 165
size limits of, 43
structure of, 108-170
Terminate method, 182-184
trigger classes, creating, 185-188
class modules (VB6), adding methods, 744
class names for Office applications in CreateObject function, 718
Class View window (VS IDE), 945
classes
default instances versus, 379-381
error handling in, 488-489
naming conventions, 36
VB.NET, 940-947
adding to solutions, 941
creating well-designed, 941-945
properties, 946-947
watching, 529-531
window classes, 341
Classes list (Object Browser), 534
Classic ADO. See ADO
Classic VB. See VB6
ClassInterface attribute, 930
classes (SQL)
in DELETE statements, 598
in INSERT statements, 596-597
in SELECT statements, 595-596
in UPDATE statements, 597
Clear method (Err object), 466
clearing
data input cells, 124-125
time sheet example application, 134
Err object, 467
Recent documents list, 292-293
ClickOnce, 975
ClickOnce application cache, 1025
ClickOnce deployment model, 982, 1016-1025
client version of .NET Framework, 819
client-server databases, 579
ClipBoard object (VB6), 773
close button, disabling in UserForms, 396
Close method (ADO Connection object), 601
Close method (ADO Recordset object), 609
CloseCurrentDatabase method (Access Application object), 726
closing application instances, 718-719
time sheet example application, 135-137
PowerPoint instances, 721
Windows Forms, 828
XLLs, 1041-1042
CLR (Common Language Runtime), 818

cmDeleteTime object (Web Services time sheet example application), 1075
cmInsertTime object (Web Services time sheet example application), 1075
code, stepping through. See stepping through code
Code Access Security (CAS), 1017
code comments, 45-49
documenting changes with, 66
internal comments, 47-49
module-level comments, 46
procedure-level comments, 46-47
updating, 49
code execution in Immediate window, 519-520
code listings. See listings
code protection in VB6, 758
code readability, best practices, 50-52
Code Region, 853
code reuse, 435-437. See also custom interfaces
code security. See security
code shortcuts, 854
code snippets, 853-855
Code Snippets Manager, 853-855
code templates, 108
code-behind assemblies, 979-980
codeless applications, 14-15
CodeNames
naming conventions, 37
referencing sheets by, 65
Collection object
iterating, 570
methods, 170
mixed object types, 443
collections
creating, 170-177, 771-772
default properties and member processing, solving problems of, 175-177, 771-772
colon character(;) in Immediate window, 520
column headers, worksheet UI setting, 120
column-relative named ranges, 73
columns
hidden columns, worksheet UI setting, 119
program columns defined, 70-71
time sheet example application, 102
COM (Component Object Model), 710
COM add-ins, 889
installation requirements, 1102
loading/unloading, 989-990
managed COM add-ins, 820
blogs for information, 962
development tools, 962-963
shimming, 952-961
time sheet example application, 963-972
registering/unregistering manually, 940
VB6, 783-787
Add-in Designer, 788-790
AddInInstance object events, 792-794
advantages of using, 798-799
automation add-ins, 799-802
checking for installation, 788
command bar architecture, 795-796
command bar event hooks, 795
converting Excel add-ins to, 797
custom toolbar faces, 796
enabling/disabling, 787
Hello World example, 783-787
installing for multiple users, 791-792
as multi-application, 798
registering, 790
security, 798
separate threading, 798-799
XLLs and, 1061
COM communications, VB.NET and, 817. See also automation, .NET Framework and
COM exceptions, 832
COM Shim Wizard, 889, 954-961
combination charts, creating, 687-690
ComboBox control, 425
default behavior, 426
as drop-down pane, 427-429
as file name box, 426-427
sizing in Ribbon UI, 295-296
text box, 426
custom toolbar definition table, 200-219
Before setting, 215, 221
Begin Group setting, 215
Command Bar Name setting, 204, 221
Control Caption setting, 204-205, 221-225
Control ID setting, 210, 224-226
Control Style setting, 212-213, 224
Control Type setting, 211
custom menu with submenus example, 220-223
custom right-click command bar example, 226-228
custom toolbar example, 223-226
Face ID setting, 213-215, 222-228, 236-237
IsEnabled setting, 209
IsMenubar setting, 206
IsTemporary setting, 209
ListRange setting, 218
Lists setting, 218
Mask setting, 228
OnAction setting, 209
Parameter setting, 217
Picture setting, 228
Position setting, 205-206, 226
Protection setting, 207-209
Shortcut Text setting, 216
State setting, 217-218, 222
Tag setting, 216-217, 233-235
Tooltip setting, 216
Visible setting, 206, 223
Width setting, 206-207
Command Bar Name setting (command bar definition table), 204, 221
command bars. See also controls; menus; toolbars
for COM add-ins architecture for, 795-796
custom toolbar faces, 796
combining with Ribbon UI, 304
heavy weight design, 307-319
light weight design, 304-307
creating in managed COM add-ins, 909-918
deleting, 208-209
design best practices, 195-199
docking positions, specifying, 205
error handled command bar builder, 219
extracting logic to loader add-ins, 308-312
icon design, 198
right-click menus, removing, 294
separator bars in, 198
table-driven command bars, 190-219
associating icons with controls, 228-232, 796
command bar definition table, 200-219
custom menu with submenus example, 220-223
custom right-click command bar example, 226-228
custom toolbar example, 223-226
event hooks, 232-241, 795
table-driven command bar builder, 199-200
time sheet example application, 241-247
Command object (ADO), 605-607
ActiveConnection property, 605, 628
CommandText property, 605
CommandType property, 605, 628
CreateParameter method, 605-606, 628
Execute method, 606-607, 628
Parameters collection, 607, 628, 637
Command window, 848
CommandBars object model, 909
CommandLineSafe DWORD value, 905
CommandText property (ADO Command object), 605
CommandType property (ADO Command object), 605, 628
CommandTypeEnum constant values, list of, 603
comments
cell comments
as help text, 86-87
time sheet example application, 104
code comments, 45-49
documenting changes with, 66
internal comments, 47-49
module-level comments, 46
procedure-level comments, 46-47
updating, 49
common data type system (CTS), 818
Common Language Runtime (CLR), 818
Common Language Runtime Exceptions, 851
communication with DLLs. See ActiveX DLLs in-process, 774
out-of-process, 774
compiling
ActiveX DLLs, 744, 750
help project files, 1090
release builds, 852
complex chart axes, creating, 701-702
complex error handling system, 476-488
central error handler, 481-488
time sheet example application, 496-506
entry point procedures in, 477
procedure error handlers, 477-480
trivial procedures, 480-481
Component Object Model (COM), 710
Component One's Doc-to-Help, 1085
Component Tray, 877
conditional compilation constants, 511-512, 851-852
conditional formatting, 92-98
dynamic tables, creating, 93-96
error conditions, highlighting, 96-98
time sheet example application, 105
Configuration Manager, 852
correlation settings (Visual Studio IDE), 822-823
configuring environment during startup process, 148-151
Connect class, creating GUID, 935-937
ConnectComplete event, 604
connected mode (ADO.NET), 865-867
Connection class
creating managed COM add-ins, 893-897
modifying for Ribbon user interface, 921
connection failures, error handling with Resume statements, 473
Connection object (ADO), 600-604
Close method, 601
ConnectionString property, 600-601, 622
ConnectionTimeout property, 601
destroying (time sheet example application), 649
enabling connection pooling, 632-633
Errors collection, 604, 634
events, 604
Execute method, 602-604, 626
initializing, 618
time sheet example application, 648
Open method, 602
State property, 601
stored procedures and, 636
connection pooling, 632-633
connection strings (time sheet example application), 882-883
Connections. See also Connection object (ADO)
to Access databases, 620-622
time sheet example application, 648-649
to SQL Server databases, 630-631
connection pooling, 632-633
error handling, 633-635
to Web Services, 1068-1071
ConnectionString property (ADO Connection object), 600-601, 622
ConnectionTimeout property (ADO Connection object), 601
conPETRASDiConnection object (Web Services time sheet example application), 1075
consolidating data, 672-673
constants
in API calls, finding value of, 333-334
best practices, 54-61
in central error handler, 483-485
conditional compilation constants, 511-512, 851-852
defined constants in command bar definition table, 201
Excel4 function return values, 1051
named constants, 72-73
naming conventions example, 33
purpose of, 57-58
viewing value of, 58
in XLOPER data type, 1046
for XLOPER data type error values, 1048
constructors in VB.NET, 941-943
ContactItem object (Outlook object model), 736
ContainerControl object, 1008
Content Type items (in Open XML), 275
Context options (Watch window), 526-527
contexts, application, 244-247
control arrays, 761-769, 878
Control Caption setting (command bar definition table), 204-205, 221-225
Control ID setting (command bar definition table), 210, 224-226
custom image management, 284-286
copying, 762
differentiating, 216-217
disabling in Ribbon UI, 292
drag-and-drop operations, 431
event hooks for, 232-241, 795
Paste Special toolbar example, 235-241
Tag property and, 233-235
controlling applications automation, 709-710
best practices, 711-714
application-specific class modules, 714
development for earliest supported version, 713
property/method calls, 712-713
variable declarations, 711-712
early binding versus late binding, 714-716
instances, 717-722
application availability, determining, 722
closing, 718-719
creating, 717-718
multiversion support, 722
referring to existing, 720-721
performance issues, 723-725
referencing object libraries, 710-711
controls. See also command bars; custom task panes (CTPs)
accelerator keys, creating, 205
ActiveX adding to Windows Forms, 826
forms (VB6) support for, 760
associating icons with, 213-215, 228-237, 796, 802-807, 863-864
captions, adding, 224
ComboBox, 425
default behavior, 426
as drop-down pane, 427-429
as file name box, 426-427
text box, 426
creating, 205
Frame
creating wizard dialogs, 409
as custom drop-down panel, 429
host controls, 1006-1008
ListObject, 1013-1016
NamesRange, 1011-1012
IDs, determining, 210
keyboard shortcuts, 216
label controls, simulating splitter bars with, 405-406
locking versus disabling on UserForms, 398-399
MultiPage
creating wizard dialogs, 409-411
Windows Common Controls and, 430
naming conventions, 38
pasting, 762
tab order, setting, 826
in UserForms
accelerator keys, 386
data binding, 386
data validation, 388-392
event handling, 386-388
exposing properties and methods of, 382-384
layering, 385
naming, 384
positioning, 385
tab order, 386
when to use, 98-100
Windows Common Controls, 430-431
Windows Forms controls, 1008
converting
between pixels and points, 338-340
cordinate systems, 694-696, 702-704
Excel add-ins to COM add-ins, 797
hexadecimal format to VBA, 334
ranges to Lists, 664
cordinate systems, converting among, 694-696, 702-704
copy functionality, handling, 154-156
Copy Local property
(auto-generated references), 897-898
Copy to range (advanced filters), 674
CopyFromRecordset method
(Range object), 624
copying controls, 762
filtered data, 674
to/from arrays (VB.NET), 842
count parameter (Excel function), 1051
COUNTA function, 693-694
COUNTIF function, 680
counting visible workbooks
(time sheet example application), 136
CreateCellsCollection procedure, 171
instantiating collections, 174
Terminate method, 184
trapping events, 179
CreateItem method (Outlook Application object), 736
CreateObject function, 717-718
CreateParameter method
(ADO Command object), 605-606, 628
creative thinking for improving performance, 551-556
asking questions, 556
breaking the rules, 554-555
data, knowledge of, 555-556
jigsaw puzzle example, 551-552, 554
“think outside the box” example, 552-554
tools, knowledge of, 556
criteria ranges (advanced filters), 674-678
database functions in, 679
cross-process calls, performance issues, 723-725
cross-version applications, 303
combining command bars and Ribbon UI, 304
heavy weight design, 307-319
light weight design, 304-307
file system access in, 320-326
installing, 330
macro-free files and, 319-320
Public profile, 329
standard user accounts, 328-329
User Account Control (UAC), 326-328
CTPs (custom task controls), 998-1006
Ctrl key, checking state of, 350-351
Ctrl+Alt+A keyboard shortcut
(Command window), 848
Ctrl+Alt+B keyboard shortcut
(Breakpoints window), 849-850
Ctrl+Alt+E keyboard shortcut
(Exceptions dialog), 850-851
Ctrl+Alt+K keyboard shortcut
(Task List), 555
Ctrl+Alt+O keyboard shortcut
(Output window), 549
Ctrl+Alt+W keyboard shortcut
(Watch window), 850
Ctrl+F8 keyboard shortcut (Step to Cursor command), 516, 543
Ctrl+F9 keyboard shortcut (Set Next Statement command), 516-517, 543
Ctrl+G keyboard shortcut
(Immediate window), 517, 542, 849
Ctrl+L keyboard shortcut
(Call Stack window), 521, 543, 850
Ctrl+Shift+F2 keyboard shortcut
(return to last position), 543
Ctrl+Shift+F8 keyboard shortcut
(Step Out command), 515, 543
Ctrl+Shift+F9 keyboard shortcut (clearing break points), 542
Ctrl+W Ctrl+E keyboard shortcut (Error List window), 848
CTS (common data type system), 818
CType function, 859
CTypeTrigger class module, 185-188
CurrentDb property (Access Application object), 726
Cursor property (Application object module), 520
CursorLocation property (ADO Recordset object), 608
cursors changing to hourglass, 546 defined, 608
CursorTypeEnum constant values, list of, 611
Custom Actions Editor, 908
Custom document properties, 161-163 adding (time sheet example application), 137
custom errors, raising, 474, 484
custom icon images in Ribbon user interface, 925-927
custom interfaces, 434 defining, 437-438 Implements keyword, 438-440
IntelliSense and, 448-460 plug-in architecture of, 460-461 robustness of, 448 sorting arrays example, 440-443
time sheet example application, 462
custom task panes (CTPs), 998-1006
Custom UI Editor, creating templates, 299
custom wizards, See dynamic UserForms; wizard dialogs
custom worksheet functions, See XLLs
customized toolbars, storing and restoring, 147 customizing user interface for dictator applications, 151-156. See also modifying CustomUI Editor, 276 customUI folder (Ribbon UI), 277
customUI XML part, 290-291
cut functionality, handling, 154-156 CVErr values, 933
data daActivities object (Web Services time sheet example application), 1075
daClients object (Web Services time sheet example application), 1075
daConsultants object (Web Services time sheet example application), 1075
DAO.Database object (Access object model), 726
daProjects object (Web Services time sheet example application), 1075
data exporting with ADO, 948-952 knowledge of (creative thinking), 555-556
pre-processing for performance optimization, 557
separating from applications with XML, 254-256
volume of, effect on performance, 558-560
XML data files from financial model example, 268
importing/exporting, 255-256, 262-263
data access and storage tier defined, 41
physical design, 617-620 reasons for using, 616-617
data access technologies ADO. See ADO defined, 599 explained, 598
data area for dynamic lists, 77
data arrangement. See data structures
data binding controls in UserForms, 386
data consolidation, 672-673
data coordinates, converting among mouse and drawing object coordinates, 702
data entry cells, handling cut, copy, paste functionality, 154-156
data entry forms, worksheets as, 4-5
data handling features. See data structures
data input cells, clearing all, 124-125
time sheet example application, 134
data manipulation. See data processing
data point markers, images as, 702
data processing, 667
Advanced Filters, 673-678
database functions in, 679 array formulas, 680-683
PDO formulas, 683-686
data consolidation, 672-673
database functions, 678-679 on formulas, 667
PivotCaches, 668
PivotTables, 668-672
data providers .NET, 565
data retrieval with ADO.NET, 864-870
time sheet example application, 882-883
data stores, worksheets as, 5
data structures, 661-662
Lists, 664
QueryTables, 664-667
structured ranges, 662-663 formulas in, 667 unstructured ranges, 662
data types explicit versus implicit conversions, 568 matching, 568
naming conventions, 29-30
Variant, 54-55
VB.NET, 838-839
in XLLs, 1037
XLOPER, 1044-1050
arrays and, 1048-1049
C++ keywords and, 1061
constants defined in, 1046
error values, 1048
memory management, 1049-1054
numeric data in, 1047
string data in, 1047
xlCoerce function, 1052-1053
xlFree function, 1052
xlGetName function, 1053
data validation, 63, 88-92. See also validation
cascading lists for, 90-92
for controls in UserForms, 388-392
time sheet example application, 104
unique entries, enforcing, 89
data validation lists, 590
Data Warehouses, 977
DataAdapter object, 867-870
mapping schema fields to, 1075
database connections. See Connection object (ADO)
database functions, 678-679
databases. See also external data
Access databases
adding data, 625-626,
632-636
advantages of, 620
connecting to, 620-622,
648-649
deleting data, 629-630
inserting data, 625-626
modifying data, 626-629
retrieving data, 622-625,
650-652
up sizing to SQL Server, 642-646
adding data to, 596-597
client-server databases, 579
deleting data from, 597-598
duplicate rows in, 580
file-based databases, 579
modifying data in, 597
normalization, 579-587
exceptions to, 586-587
first normal form, 580-581
second normal form, 582-583
third normal form, 584-586
Northwind sample database, 615-616
primary keys, natural versus artificial, 592-594
processing data from. See data processing
reasons for using, 578
referential integrity, 587-592
relational databases, 578-579
relationships, 587-592
many-to-many, 590-591
one-to-many, 590-591
one-to-one, 588-589
resources for information, 613-614, 647-648
retrieving data from, 595-596
SQL. See SQL
SQL Server databases
advantages of, 630
connecting to, 630-631
connection pooling, 632-633
default instances versus named instances, 642
disconnected recordsets, 640-642
error handling connections, 633-635
multiple recordsets, 638-640
parameter refreshing, 637-638
security types, 631
stored procedures, 635-637
worksheets versus, 577-578
DataEntry application context, 244
DataReader object, 865-867
DataSet object, 867-870
creating from schemas, 1075
DataTable object, 835, 867-870
dates in criteria ranges, 677
DAVERAGE function, 679
DCOM (Distributed Component Object Model), 710
deb ug builds, 851
DEBUG conditional compilation constant, 851
debug mode, 508-512
conditional compilation constants, 511-512
Stop statement, 510-511
supporting, 149-151
user-defined debug mode, 509-510
Debug toolbar, displaying, 514
Debug.Assert method, 540-541
Debug.ini, 157, 369
Debug.Print statement, 518
debugging. See also error handling
assertions, 540-541
break mode, error trapping settings, 507-508
break points, setting/removing, 512-513
Call Stack window, 521-522
debug mode, 508-512
conditional compilation constants, 511-512
Stop statement, 510-511
user-defined debug mode, 509-510
dictator applications, 22
frequency of, 65
Immediate window, 517-520
code execution in, 519-520
Debug.Print statement, 518
variable evaluation in, 519
keyboard shortcuts, list of, 542-543
Locals window, 532-533
message box debugging, 517
Object Browser, 533-537
properties (VB.NET), 947
with Resume statements, 473
run mode versus break mode, 507
Set Next Statement command, 516-517
stepping through code, 513-516
Step Into command, 514-515
Step Out command, 515
Step Over command, 515
Step to Cursor command, 516
test harnesses, building, 537-540
VB.NET solutions, 845-853
Breakpoints window, 849-850
Call Stack window, 850
Command window, 848
conditional compilation con-
stants, 851-852
Error List window, 848
Exception Assistant, 846-847
Exceptions dialog, 850-851
Immediate window, 849
keyboard shortcuts, setting, 845
Object Browser, 847-848
Output window, 849
unmanaged code, enabling debugging, 846
Watch/Quick Watch windows, 850
Watch window, 522-532
Context options, 526-527
editing watches, 525-529
modifying value expressions, 524-525
Quick Watch window, 531-532
setting watches, 522-524
Watch Type options, 527-529
watching arrays, UDTs, classes, 529-531
XLLs, 1060-1061
Decimal data type (VB.NET), 839
declarations
for API calls, finding, 333
defined, 6
declarative programming
language, worksheet functions as, 6-7
Declarative Referential Integrity
(DRI), 644
Declare statement for API calls, 331-332
declaring arguments, 62-63
object variables, 55
variables with conditional compilation
constants, 511
including object libraries in, 711-712
VB.NET, 834-836
WithEvents object variable, 177
default instances
classes versus, 379-381
SQL Server name, 642
default interfaces, 434
default object properties, calling, 63
default properties for collections, 175-177, 771-772
default values, setting application properties to, 130
defensive coding, 62-65
defined constants in command bar definition table, 201
defined names, 71-78
in advanced filters, 675
in charts
auto-expanding charts, 692-694
plotting functions, 696-698
scrolling/zooming in time series, 694
SERIES function, 691-692
setting up links, 691-692
transforming coordinate systems, 694-696
for linking PivotTables to QueryTables, 671
named constants, 72-73
named formulas, 76-77
named ranges, 73-75
naming conventions, 39
scope of, 77-78
time sheet example applica-
tion, 102-103
defining custom interfaces, 437-438
DELETE FROM clause (SQL DELETE statement), 598
DELETE statement (SQL), 597-598
for Access databases, 629-630
deleting command bars, 208-209
database data, 597-598
from Access databases, 629-630
files to Recycle Bin, 360-361
toolbars, 127
dependencies, detected dependency files, 900
dependency checks, 142-143
deployment manifest, 992, 1017
signing, 1022
deployment models for VSTO, 1016
ClickOnce, 1016-1025
derived data
defined, 585
normalization and, 587
described format, XML as, 250
Description property (Err object), 466
descriptions
for COM add-ins, 789
for function library add-ins, creating, 115-116
design
command bar best practices, 198-199
cross-version applications
heavy weight design, 307-319
light weight design, 304-307
data access tier, 617-620
icon design for command bars, 198
Ribbon UI best practices, 278-289
UI design. See UI design
Double data type, performance and, 573
drag-and-drop functionality between controls, 431
disabling, 154-156
drawing object coordinates
converting among data and mouse coordinates, 702
locating chart items within, 704-706
drawing objects, naming conventions, 38
DRI (Declarative Referential Integrity), 644
drop-down pane, ComboBox control as, 427-429
DropButtonClick event, 426
drop-down controls, adding to toolbars, 225
DSOFile.dll, 162
dummy XY series, creating, 701-702
duplicate code, eliminating, 44
duplicate rows in databases, 580
dynamic lists
defined, 76
elements of, 77
dynamic tables, creating with conditional formatting, 93-96
dynamic UserForms, 411
event handling, 416-419
scroll regions in, 415
subset UserForms as, 411
table-driven dynamic wizards, 411-415
dynamically modifying worksheet UI, 124-125
EnableCancelKey property
(Application object), 22
EnableEvents property
(Application object), 528
enabling
- circular references, 684
- COM add-ins, 787
- connection pooling, 632
- error handlers, 468
- keyboard shortcuts, 823
- screentips, 823
- unmanaged code debugging, 846
- of API calls, 335-337
defined, 44
IntelliSense and, 448-460
of UserForms, 382-384
encryption passwords, 783
end of file (EOF), 624
End Try statement (structured exception handling), 830
Enterprise Resource Planning (ERP) systems, 977
entry point procedures
in complex error handling system, 477
for light weight UI design, 305
for heavy weight UI design, 311, 315
simple error handling in,
475-476
enumeration constants, mapping help topic IDs to, 1092
enumeration members, CCell class module, 167
enumerations, 32
assigning values to, 34
naming conventions example, 34
UserForms and, 383
EnumWindows API call, 363
environment modifications during startup process, 148-151
EOF property (ADO Recordset object), 607, 624
equal sign (=) in criteria ranges, 677
ERP (Enterprise Resource Planning) systems, 977
Err object, 466-467
clearing, 467
raising custom errors, 474
error bars, creating step charts with, 699-701
error conditions, highlighting with conditional formatting, 96-98
error handled command bar builder, 219
error handlers
activating, 468
central error handler, 481-488
time sheet example application, 496-506
defined, 467-468
enabling, 468
procedure error handlers, 477-480
scope, 468-469
derror handling. See also debugging; exception handling
catching errors, 480
in classes and UserForms, 488-489
closing application instances, 718-719
complex error handling system, 476-488
central error handler, 481-488, 496-506
entry point procedures in, 477
procedure error handlers, 477-480
trivial procedures, 480-481
custom errors, raising, 474, 484
demo program, 490-496
Err object, 466-467
clearing, 467
error handlers
activating, 468
defined, 467-468
enabling, 468
scope, 468-469
importance of, 465
On Error statements, 469-472
On Error GoTo
<Label>, 470
On Error GoTo 0, 472
On Error Resume Next, 470-472
Resume statements, 472-474
debugging with, 473
Resume <Label>, 474
Resume Next, 473
simple error handling, 475-476
single exit point principle, 475
SQL Server database connections, 633-635
trapping errors, 480
unhandled errors versus handled errors, 465
Error List window, 848
error log file, 485
in error handling demo program, 495-496
error numbers, availability of, 474
error trapping settings, 507-508
error values in XLOPER data type, 1048
Error Exit label, 475
ErrorProvider component, 877
errors, ignoring (time sheet example application), 136
Errors collection (ADO Connection object), 604, 634
ETC (Evil Type Coercion), 55
EVALUATE function, 698
evaluating variables/expressions in Immediate window, 519
event procedures, error handling in, 488-489
events
Addin Instance object, 792-794
Initialize event, 792
OnAddInsUpdate event, 794
OnBeginShutdown event, 794
OnConnection event, 792-793
OnDisconnection event, 794
OnStartupComplete event, 793
Terminate event, 794
ADO Connection object, 604
ADO Recordset object, 612
application-level event handling (time sheet example application), 190-193
raising, 180-188
trapping, 177-182, 492
workbook events, add-ins and, 784
evidence in VSTO security model, 1017
Evil Type Coercion (ETC), 55 examples. See time sheet example application
Excel
as application development platform, 4-7
multiple instances of, 781
supported versions, 9-10
Excel 2007 SDK, 1030, 1062
Excel = Microsoft.Office.Interop.Excel namespace, 992
Excel developers
categories of, 2-4
defined, 3
Excel Function Wizard, registering UDFs with, 112-114
Excel object library,
referencing, 776
Excel object model, 7. See also objects
Excel security, 1094-1095
Excel Services, 976
Excel versions
maintaining backward compatibility with, 62
targeting for managed COM add-ins, 909
Excel4 function, 1050-1051
Excel9.0lb file, 857
exception handling, 829-833. See also error handling exceptions
COM exceptions, 832
defined, 829
nested exceptions, 833
Exceptions dialog, 850-851
Exchange Server, 976
exclamation point character (!), volatile functions, 1038
EXE applications (VB6), 775-783
front loaders, 782-783, 808-815
out-of-process communication, 774
Roman numeral conversion example, 775-782
Execute method (ADO Command object), 606-607, 628
Execute method (ADO Connection object), 602-604, 626
ExecuteComplete event, 604
ExecuteOptionEnum constant values, list of, 603
execution point changing, 516-517
defined, 514-515
exit points, single exit point principle, 475
explicit data type conversions, 568
exporting data with ADO, 948-952
time sheet example application, 884-887
XML data files, 256, 262-263
Express Edition (SQL Server), 630
expressions
evaluating in Immediate window, 519
value expressions, modifying, 524-525
watching
  in arrays, UDTs, classes, 529-531
eediting watches, 525-529
setting watches, 522-524
extender providers for Windows Forms, 876-879
Extensibility namespace, 893
external data, importing into QueryTables, 664-667
external data retrieval, performance and, 537
ExtractIcon API call, 348
extracting command bars logic to loader add-ins, 308-312

F
F2 keyboard shortcut (Object Browser), 533, 543, 847
F5 keyboard shortcut (run code), 514, 542
F8 keyboard shortcut (Step Into command), 514-515, 542
F9 keyboard shortcut (setting break points), 513, 542
Face ID setting (command bar definition table), 213-215, 222-228, 236-237
FetchComplete event, 612
FetchProgress event, 612
Fields (in .NET Framework), 941
FIFO (First In First Out) data access method, 845
file formats, selecting, 275
file name box, ComboBox control as, 426-427
file system access in cross-version applications, 320-326
File System Editor, 908
file system-related API calls, 355
deleting files to Recycle Bin, 360-361
folders, browsing for, 361-369
time sheet example application, 369-371
special folders, locating, 357-360
UNC paths, changing to, 356-357
user ID, finding, 355-356
File Types Editor, 908
file-based databases, 579
filename extensions, MIME types and, 1021
files, deleting to Recycle Bin, 360-361
FileSystemObject (FSO) object, 320
methods, 321
FillDocument.dot, 723
filter pane, ComboBox control as, 427-429
Filter property (ADO Recordset object), 608
filters, Advanced Filters, 673-678
database functions in, 679
Finalize method (destructors), 943
finalizers, 861
Finally statement (structured exception handling), 830
Financial Applications Using Excel Add-in Development in C/C++ (Dalton), 1062
financial model example, 256-257
preventing results import, 269
XML data file from, 268
XML Maps, 259-267
XSD file, 263-265
creating, 257-259
FindWindow API call, 342
FindWindowEx API call, 343
First in First Out (FIFO) data access method, 845
first normal form, 580-581
floating-point arithmetic, integer arithmetic versus, 570
folders
  browsing for, 361-369
time sheet example application, 369-371
special folders, locating, 357-360
Folders property (Outlook MAHFolder object), 736
For...Each loops
  iterating collections, 570
  referencing collections, 176
foreign keys
  defined, 581
  explained, 587-588
form-based user interfaces, worksheet-based user interfaces versus, 154-156
formatting
  conditional formatting, 92-98
dynamic tables, creating, 93-94, 96
error conditions, highlighting, 96-98
time sheet example application, 105
with styles, 78-83
tables, 85-86
forms (VB6). See also
  UserForms; Windows Forms
displaying, 751-758
as modeless, 756
Ruby Forms, 759
as top-level windows, 756
UserForms versus, 759-769
ActiveX control support, 760
collection arrays, 761-769
Forms controls, advantages of, 100
forms packages, 759
Forms toolbar controls, advantages of, 100
formula columns in QueryTables, 670-672
formulas
  assigning to shapes, 88
data processing on, 667
named formulas, 76-77
in structured ranges, 667
forums for VB.NET information, 871
forward compatibility
  application instances, 722
object libraries, 713-715
Frame control
creating wizard dialogs, 409
as custom drop-down
panel, 429
in UserForms, 385
Framework. See .NET Framework
Friend keyword (VB.NET), 834
FROM clause (SQL SELECT
statement), 595
Access database example, 623
front loaders, 782-783, 808-815
FSO (FileSystemObject)
object, 320
methods, 321
fully qualified object variable
names, 56
fully qualifying property/method
calls, 712-713
fully relative named ranges, 73
function categories for managed
automation add-ins, 934
function library add-ins, 110-117.
See also UDFs
names and descriptions,
creating, 115-116
function return value system
(error handling), 499
function tables in XLLs,
1035-1039
functional decomposition, 43
functions
code comments in, 46
as declarative programming
language, 6-7
naming conventions, 35-36
plotting in charts, 696-698
XLL-based. See XLLs
full help entry (XLL
function table), 1038
full help text entry (XLL
function table), 1038
G
GAC (Global Assembly
Cache), 856
garbage collection (GC), 818,
861, 943
gbDEBUG_MODE constant
(central error handler),
484, 509-510
GDI+ (Graphics Device
Interface), 286
general purpose add-ins, 16-18,
117-118
General tab (Add-in Designer),
788-790
Get blocks (VB.NET properties),
946-947
GET.CHART.ITEM XLM
function, 704-706
GetCurrentProcessID API
call, 343
GetCustomUI function
(IRibbonExtensibility
interface), 922
GetDC API call, 339
GetDefaultFolder() property
(Outlook NameSpace
object), 736
GetDesktopWindow API
call, 343
GetDeviceCaps API call,
338-339
GetDirectory API call, 368
getEnabled callback
for Ribbon UI, 314-316
troubleshooting, 288
GetKeyState API call, 350-351
GetLowerBound method
(VB.NET arrays), 841
GetNamespace property
(Outlook Application
object), 736
GetObject function, 720-721
GetOpenFilename API
call, 361
GetSaveAsFilename API
call, 361
GetSetting property
(ThisWorkbook
object), 17
GetStaticData function (Web
Services time sheet exam-
ple application), 1073
GetSystemMetrics API call
calling, 335
constants, finding value of, 333
declaration, 332
encapsulating, 336
screen resolution, reading, 337
GetTempPath API call, 358-360
GetType function, hiding,
934-935
GetType method (VB.NET
arrays), 844
GetUpperBound method
(VB.NET arrays), 841
GetUserName API call,
355-360
GetValue method (VB.NET
arrays), 842
GetWindowLong API call, 349
GetWindowText API call,
343
glHANDLED_ERROR
constant (central error
handler), 484
Global Assembly Cache
(GAC), 856
global callback handlers, 286-287
for shared tabs, 281-283
global format, XML as, 250
GlobalMultiUse instancing type,
770-771
glUSER_CANCEL
constant (central error
handler), 484
graphics
background graphics, preparing
for user interface, 151-153
displaying on UserForms,
397-398
Graphics Device Interface
(GDI+), 286
greater than/less than symbols
(>, <) in criteria ranges,
677
gridlines, simulating, 84
GROUP BY clause (SQL
SELECT statement), 596
GUID, 714
for Connection class, creating,
933-937
managed COM add-ins
registry keys, 899
H
handled errors, 465
handles, 334-335
window handles, 340
handling events. See event handling
HAVING clause (SQL SELECT statement), 596
headers, worksheet UI setting, 120
heavy weight cross-version UI design, 307-319
help files, 1085-1086. See also HTML Help Workshop creating, steps for, 1086 explained, 1086 form-based help system (time sheet example application), 879
help project files compiling, 1090 creating, 1086 setting initial options, 1087-1088
Index, creating, 1088-1091
Table of Contents, creating, 1088-1091
topic files creating list of, 1089-1090 displaying from VBA, 1092-1094
ID numbers for, 1090-1092 introductory file, creating, 1088 
“No Help Available” file, creating, 1088 writing content for, 1091
help project files, 1086. See also help files compiling, 1090 creating, 1086 setting initial options, 1087-1088
help system, Object Browser and, 533
help text, cell comments as, 86-87
HelpContext property (Err object), 466
HelpFile property (Err object), 466
HelpNameSpace property (HelpProvider component), 879
HelpProvider component, 877
help_topic entry (XLL function table), 1038
hexadecimal format, converting to VBA, 334
hidden columns, worksheet UI setting, 119. See also program columns
hidden rows, worksheet UI setting, 119. See also program rows
hidden, See also visibility
GetType function, 934-935
Ribbon UI, 294-295
UserForms, 381
windows, 823
hierarchical format, XML as, 250
high-order bits, 351
Highlight method (Cell object), 174
highlighting error conditions with conditional formatting, 96-98
hit counts, defined, 850
HKEY_CLASSES_ROOT\ CLSID registry key, 899
HKEY_CURRENT_USER\ Software\ Microsoft\ Office \10.0\ExcelAddins, 1102
HKEY_CURRENT_USER\ Software\ Microsoft\ Office \10.0\ExcelOptions, 1102
HKEY_CURRENT_USER\ Software\ Microsoft\ Office \11.0\ExcelAdd-in Manager, 1102
HKEY_CURRENT_USER\ Software\ Microsoft\ Office \11.0\ExcelAdd-ins, 900, 992, 1102
HKEY_CURRENT_USER\ Software\ Microsoft\ Office \ExcelAddIn\FirstAddIn. Connect, 905
HKEY_LOCAL_MACHINE\ SOFTWARE\ Microsoft\ Office\ExcelAddIns, 1102
hooking events for controls, 232-241, 795
Paste Special toolbar example, 235-241
Tag property and, 233, 235
host applications, 710
host controls, 1006-1008
ListObject, 1013-1016
NamedRange, 1011-1012
host items, 1006-1008
hourglass, changing cursor to, 546
HTML Help Workshop, installing, 1086. See also help files
HtmlHelp API function, 1092
hWnd (window handle), 340
I
IA (Interop Assembly), 856
icon design for command bars, 198
icons. See also bitmaps; resources adding to toolbar buttons, 225 associating with controls, 213-215, 228-232, 236-237, 796, 802-807, 863-864
custom image management, 294-286 creating bitmaps for, 228-230 custom icons images in Ribbon user interface, 925-927 transparent backgrounds, setting, 214-215 window icons, changing, 348-349
ICustomTaskPaneConsumer interface, 999
ID/Tag property combination, hooking events into, 234
identifiers, defined, 31
identifying workbooks with custom document properties, 161-163
IDisposable interface, 943
IDs
for controls, determining, 210
hooking events into, 234
IDTExtensibility2 interface, 890-891
event procedures, 893
If...ElseIf...End statement, performance and, 571
IFERROR function, 111
example (XLLs), 1057-1060
IgnoreOtherApplications property (Application object), 21
IgnoreRemoteRequests property (Application object), 149
ignoring errors (time sheet example application), 136
If() function, performance and, 571
ImageCombo control, 430
ImageList control, 430
images, as data point markers, 702. See also icons
Immediate window, 210, 517-520, 849
code execution in, 519-520
Debug.Print statement, 518
variable evaluation in, 519
Implements keyword, 438-440
implicit data type conversions, 568
importing
code from text files, 855
external data into
QueryTables, 664-667
namespaces, 827-828
PerfMon results, 550
XML data files, 253, 262-263
XML results, preventing, 269
XSD, 255
Imports statement, 858
VB.NET solutions, 828
in-place activation, 980
in-process communication, 774
inclusion lists, 1017
indentation of code, 51
index arrays, 566-567
Index file (in help files), creating, 1088-1091
indexes, unique, 594
INDIRECT() function, 683
infinite loops, 473-474, 510
avoiding, 64-65
initial load behavior of COM add-ins, 789-790
Initialize event
AddInInstance object, 792
error handling, 489-492
initializing, 174
ADO Connection object (time sheet example application), 648
applications (time sheet example application), 125-127
Connection object (ADO), 618
object variables, 55
user interface workbooks (time sheet example application), 128-130
variables (VB.NET), 834-836
INNER JOIN statement (SQL), 595
InnerException property (nested exceptions), 833
InprocServer32 key, 899
Insert File as Text, 855
INSERT INTO clause (SQL), 595
INSERT INTO clause (SQL), 596-597
INSTRT statement (SQL), 596-597
for Access databases, 625-626
time sheet example application, 652-656
inserting code snippets, 854
installation
COM add-ins for multiple users, 791-792
cross-version applications, 330
digital certificates, 1018
distributing application updates, 1104-1105
distributing applications, 1104
file locations, selecting, 1099-1100
HTML Help Workshop, 1086
manual installation, 1103
Northwind sample database, 615
PerfMon utility DLLs, 547
requirements
for add-ins, 1100-1102
for templates, 1100
Setup.xls workbook installation, 1103
VSTO project templates, 982
VSTO solutions, 1025
Web Services Toolkit, 1069
with Windows Installer, 1104
installed applications, determining, 722
installers. See setup projects
instances (of applications), 717-722
application availability, determining, 722
closing, 718-719
creating, 717-718
of Excel, starting multiple, 781
multiversion support, 722
of Outlook, starting, 813
referring to existing, 720-721
of Word, starting, 813
instances (of classes)
creating (VB.NET), 836-837
types of (ActiveX DLLs), 769-771
instantiation, 174
auto-instantiation, 379
Int32 values, 933
integer arithmetic, floating-point arithmetic versus, 570
Integer data type (VB.NET), 839
integer division operator (\), 570
intellectual property. See security
IntelliSense, 448-460
early bound object variables and, 60
interfaces. See also class modules; objects
casting, 447
custom interfaces, 434
defining, 437-438
Implements keyword, 438-440
IntelliSense and, 448-454, 456-460
plug-in architecture of, 460-461
robustness of, 448
BILLABLE_HOUR type structure, 653
BILLABLE_HOURS UDT, code using, 529
binary search algorithm, 563-565
BinarySearch method, 843
bInsertTimeEntry function, 653-654
blank lines in code, 50
block scope variable declaration, 835
block scope within Do…Loop, 835
bInsertTimeEntry function, 653-654
buffer usage, ignoring buffer length variable, 356
building blocks of SEH, 830
building XML to submit to Web Service, 1081-1083
bWordAvailable and bOutlookAvailable functions, 811-812
calculating reasonable chart axes scales, 706-708
callback for button in Ribbon class module, 987
callback for getImage attribute, 256
callback function for custom icons images, 927
callback handler in first add-in, 281-282
callback handler in second add-in, 283
callbacks for interaction with Windows file picker dialog, 364-368
callbacks for sheet navigation, 297-298
callbacks to invalidate buttons, 289
CAuthor class, 436
CAuthor class implementing IContactDetails interface, 444-445
CCell class module, 166-167
CCell class module with new methods added, 172
CCell class module, 173
CControlEvents class module, 239-240
CDataAccess class, 617-618
CDataAccess class usage, 619
CDialogHandler ShowVB6Form method, 768
central control routine to handle navigation between forms, 424-425
central error handler, 482-483
central error handler implementing re-throw system, 502-504
Change event of NamedRange1 control, 1011
changes to CCell class module to trap ChangeColor event, 182
changes to CCell class module to trap ChangeColor event of CTypeTrigger, 186
changes to CCells class module to assign references to CTypeTrigger to cell objects, 186-188
changes to CCells class module to raise events, 180-181
changing settings at code module level, 823
changing to UNC path, 357
changing width of Name drop-down list, 347-348
checking for installed applications, 722
checking for, starting, and closing Outlook, with error handling, 720-721
checking network group membership, 1095-1096
checking object's interfaces, 447-448
CHElloWorld code module updated to support forms, 754-755
list formula for dynamic lists, 77
list of topic files (in help files), creating, 1089-1090
list ranges (advanced filters), 674
listings
accessing Excel Application object from ThisAddIn module, 987
Accessing the Excel Application object from class module, 987
add what using Command window, 849
adding custom icons to CommandBarButton, 231-232
adding index keywords to HTML file, 1091
additions to CCells class module for event trapping, 178-179
AddMoreRows procedure, 133-134
advanced filtering with VBA, 675
alignment and indentation of code, 51
Analyze method of Cell object, 170
AnalyzeActivateCell procedure, 168
AppFunction code, using Excel Application within automation addins, 801-802
ArrayList object, working with, 843-844
assigning event-handler classes to controls created at runtime, 417-418
associating icon resource file to Windows Form, 864
auto-generated attributes in the Connect class, 899
BackgroundWorker code usage, 890-892
base connection string syntax for SQL Server, 630
bCauseAnError function, 492-493
class to handle TextBox's events, 416-419
classic approach, 837
ClassInterface attribute, 930
clear data entry area feature, 124
clearing Most Recently Used file list, 293
closing CTPs, 1003
cmdConvert_Click event procedure, 779-780
cmdOK_Click event procedure, 753, 811
code region, 553
combining sort and binary search, 565
common callback handler, 287
compare two arrays, 558-559
complete updated CHelloWorld code module, 749
complex error handler, 479
conditionally disabling On Error Resume Next, 509
configuring the Excel environment for dictator applications, 149-151
connecting to SQL Server with integrated security and support for connection pooling, 632
connection string for Access 2003, 621
connection string for Access 2007, 621
connection string for SQL Server with integrated security, 631
connection string for SQL Server with standard security, 631
constructor with arguments, 944
control array demo form specific code, 765-767
control structures, bad example, 48
control structures, good example, 49
controlling Word, 711
converting from mouse coordinates to data and drawing object coordinates, 703-704
copying input flows list to export copy, 266-267
copying worksheets, 320
CProgressBar class implementing IProgressBar interface, 456-459
create new instance and dispose of instance of class, 944
CreateCellsCollection procedure in MEntryPoints module with Terminate method, 184
creating and deleting custom menu, 967-970
creating collection of cell objects, 171
creating ListObject host control and populating with data, 1013-1015
creating new instance of Word, 717-718
creating new SqlConnection, 883
creating structured ranges from ADO recordsets, 663
creating the Report Options panels (table-driven dynamic wizards), 413-415
creating toolbar, 913-914
creating XML and schema files, 887
CTP Load event, 1001-1002
CTP, making available when add-in is loaded, 999-1000
CTypeTrigger class module, 185
custom cleanup code, 995
custom UDF interface IPEDFunctions, 936
custom worksheet functions, 1035
DataAdapter and DataSet object usage, 868-869
DataReader object usage, 865-867
DEBUG, using in code, 852
Debug.Assert example, 540
declare and instantiate objects, 836
declare variables and assign values to them, 834
declaring and instantiating Excel objects, 859-860
declaring arrays and initializing later, 841
declaring objects with correct object library, 712
default properties, 63
deleting Access data, 629-630
deleting files to Recycle Bin, 360-361
deleting toolbar, 915
destroying the Connection object, 649
determine which version of Excel is available, 884-886
disabling controls by locking them, 399
disconnected recordsets, creating and using, 641-642
DisplayDLLForm procedure, 757
displaying charts on UserForms, 398
displaying exception descriptions, 832
displaying help file from message box, 1092
displaying help file with HtmlHelp API function, 1093
displaying Windows Form in Excel, 971-972
distinguishing controls using Tag setting, 216
DllMain function, 1040
HttpListener checking for custom document properties, 162
dual variable declarations using conditional compilation constants, 511
early binding example, 59
early versus late binding, 715-716
capsulating GetSystemMetrics API function and related constants, 336
entire function wrapped in On Error Resume Next, 471
entry point handling in new application structure, 312
EntryPoint subroutine, 493-494
enumerating arrays, 841
enumeration for help topic IDs, 1092
error handling connection attempts, 633-634
error handling demo UserForm, 490-491
example XML file, 251
example XSD file, 252
executing stored procedures as method of Connection object, 636-637
ExitApplication procedure, 135
exporting data to Excel worksheet, 948-952
extracting multiple recordsets from ADO Recordset object, 639-640
Fields collection usage, 612
file search function, 324-326
finding Excel main window handle, 344-345
finding size of pixels, 339
finding workbook window handle, 345-346
Form_Load event procedure, 777
forTimeDiff named formula, 103
FP struct, 1039
FProgressBar form module implementing IProgressBar interface, 452-453
framework for well-designed classes, 941-943
function table entry for IFERROR function, 1059-1060
GeneralDemo procedure, 221
generated class to connect to Maths Web Service, 1070-1071
generated code in Connection class, 894
generic bubble-sort, 435
generic BubbleSort procedure for classes that implement ISortableObject, 441
generic sorting procedure usage for collection of CAuthors, 442
Get and Set blocks with different scopes, 947
GET.CHART.ITEM usage to locate a chart item’s vertices, 705-706
GetStaticText function, 1076
handle the ellipsis in file name combo, 426-427
HandleDropDown procedure, 225
HandleRegistration function, 1055-1056
HandleTextBox procedure, 225
handling controls’ events, 387-388
handling cut, copy, and paste for data entry worksheets, 155-156
“How World” add-in using Auto_Open in standard module, 784
“How World” add-in using workbook events in ThisWorkbook module, 784
Hello World and Goodbye World messages, 986
IPlugInForm interface class, 461
IProgressBar interface allows choice between form or class, 459-460
IProgressBar interface class, 451
ISortableObject interface class, 438
Item property and NewEnum method from CCells collection, 771
key states, checking, 350-351
hiding typeinfo in function list, 935
hooking command bar control Click event, 916-917
how many 1’s are in a binary number?, 557
Icon property procedure, 806
IContactDetails interface class, 444
IFERROR function, 1057-1059
IFERROR user-defined function, 111, 800
implementing application contexts, 245-246
implementing COM add-in CommandBar architecture, 795-796
Imports statement, 828, 930
Imports statements required for CommandBar handling, 909
include calls to start and stop monitoring, 549
infinite loops, avoiding, 64
initializing the Connection object, 648-649
inserting data into Access, 625-626
installing add-ins using object model, 1103
instantiating event handler in Auto_Open procedure, 238
internal comments, bad example, 47
internal comments, good example, 48
IPlugInForm interface class, 461
IProgressBar interface allows choice between form or class, 459-460
IProgressBar interface class, 451
ISortableObject interface class, 438
Item property and NewEnum method from CCells collection, 771
key states, checking, 350-351
preventing user from closing UserForm, 396
preventing validation when Clear Settings button is clicked, 878
procedure to compare two alternatives, 567-568
procedure with automatic PerfMon calls added, 548
procedure with manual PerfMon calls added, 548
procedure with simple error handler, 467
procedure-level comment example, 47
process both arrays within one loop, 559-560
ProExcelDev Maths Web Service, 1067
progress bar UserForm usage, 422
ProgressBar class, 449-450
properties, using, 946
property procedure usage in UserForms, 383-384
providing Excel with namespace, 270-271
QuickSort procedure for one-dimensional string arrays, 561-563
read-only property, 947
read-write property, 946
reading and writing variant arrays, 572-573
reading screen resolution, 338
reading the application setting for connection string, 883
reading user's login ID, 355-356
recordset navigation, 699
referencing Cells collection in For...Each loop, 177
referencing collections in For...Each loops, 176
referring to ActiveDocument, 712-713
refreshing QueryTables when opening workbooks, 666-667
register and unregister assemblies for use by COM, 938-939
registering UDFs with Application.
MacroOptions, 113
releasing Excel COM objects with a function, 862
removing right-click menu in Excel 2007, 294
ResetAppProperties procedure, 130
restoring Excel settings during shutdown, 145-146
restoring Excel toolbars during shutdown, 147-148
retrieving data from Access, 622-623
retrieving holiday dates from Outlook Calendar, 737-739
revised XML markup for custom icons images, 926
Ribbon XML for PETRAS time sheet application, 313-314
Ribbon XML for PNG image, 285
Ribbon XML for sheet navigation, 297
Ribbon XML in first add-in for shared tab, 280-281
Ribbon XML in second add-in for shared tab, 282-283
Ribbon XML to disable Excel Options and Exit Excel commands, 292
Ribbon XML to hide New, Open, and Save commands, 292
Ribbon XML to invalidate buttons, 288
Ribbon XML to size comboBox controls, 296
RibbonX callback procedure, 307
RibbonX markup for demonstration application, 306-307
RibbonX sample customization, 276
running Access report using Excel data, 727-729
sample Debug.Print statements, 518
scope of error handler, 468-469
SERIES function examples, 691
setting path and name to help file, 879
setting window icons, 348-349
several Catch blocks and Finally block, 831-832
shortcuts, using to insert code snippets, 854
show selected name, 828
ShowControlArraysForm procedure, 768
ShowDLLMessage procedure, 746
ShowHelp procedure, using from form’s Help button, 1094
showing pop-ups for list boxes, 399
showing splash screen at startup, 420-421
showing UserForm next to active cell, 401-402
ShowMessage method, 744
ShutdownApplication procedure, 135-136
simple error handler example, 476
simple stored procedure, 635
single callback handler for several control objects, 286-287
sortable CAuthor class, 440
sorting and listing mixed classes that implement ISortableObject and IContactDetails, 446-447
SORTSEARCH_INDEX user-defined type, usage of, 566-567
SpecifyConsolidationFolder procedure, 370-371
SQL DELETE statement, 598
SQL INSERT statement, 596
SQL SELECT statement, 595
SQL UPDATE statement, 597
standard procedures to include in all modeless forms, 423-424
starting and closing Word, with error handling, 719
startup and shutdown events for worksheet, 1011
startup events in VSTO work-
book solutions, 1010
StateDemo procedure, 222
stored procedure that returns
multiple recordsets, 638
StoreTimeSheet function, 1077-1078
storing Excel settings in the
Registry, 143-145
Sub Main procedure, 814
Sub Main stub procedure, 809
subroutine and function error
handlers, 477-479
Terminate method in CCell
class module, 183
Terminate method in CCells
class module, 183
test harness for
ReturnPathAndFilename
procedure, 538-539
testing for key press, 353-354
ThisAddIn class for
HandleCTP example, 1004-1005
ToggleButton_Click event
procedure, 1005
trivial procedures don’t
require error
handlers, 481
turning labels into splitter
bars, 405-406
txtConvert_KeyPress event
procedure, 778
type mismatch error, 434
unloading Windows
Forms, 829
updated ClassInterface
attribute, 937
updated CreateCellsCollection
procedure in
MEntryPoints module for
event trapping, 179
UpdateShipper parameter
query, 626
updating Access data, 627-628
updating defined names and
refreshing PivotCaches
when QueryTable is
refreshed, 671-672
user interface layer determines
response, 377
user interface support layer
determines response, 378-379
user-defined function in
automation add-in, 931-933
UserForm controls, using
directly, 382
UserForm’s default instance, 380
UserForms as classes, 380
Using keyword to
automatically call
Dispose method, 945
validating controls, 389-392
Validating event subroutine for
several controls, 577-578
variables, interfaces, and
classes, 434
Variant arrays, 55
version checking, 142
viewing code in text
editors, 176
when to use On Error Resume
Next, 470-471
Workbook_SheetBeforeRight
Click event handler, 227
WriteDLLMessage
procedure, 750
WriteToTextFile and
ReadFromTextFile proce-
dures, 537-538
writing selected data to active
worksheet, 1002
writing settings to the user
interface worksheets, 122-123
xlAddInManagerInfo function, 1042-1043
xlAutoClose function, 1042
xlAutoOpen function, 1041
XLL function table, 1036-1037
XLOPER data type, 1045-
1046
XML data file from NPV
model, 268
XML data file produced from
model, 262
XML for Ribbon user inter-
face, 919-920
XML output from
GetStaticData function, 1074
XML passed to
StoreTimeSheet function, 1074-1075
XSD file for model, 264-265
XSD file for NPVModelData
element, 258
ListObject controls, 1013-1016
ListRange setting (command bar
definition table), 218
Lists, 255, 664
cascading lists for data
validation, 90-92
converting ranges to, 664
dynamic lists
defined, 76
elements of, 77
Lists setting (command bar
definition table), 218
ListView control, 430
load behavior of COM add-ins,
789-790
Load event (Windows
Forms), 826
loader add-ins
creating for Ribbon UI,
312-318
extracting command bars logic
to, 308-312
loading
bitmaps from resource files,
806-807
COM add-ins, 989-990
VSTO add-ins, 993-994
XLLs, 987-989
Local folder (Windows Vista), 329
localization with resource
DLLs, 790
LocalLow folder (Windows
Vista), 329
Locals window, 532-533
locating user interface workbooks (time sheet example application), 137
locking controls on UserForms, 398-399
LockTypeEnum constant values, list of, 611
logical tiers of application, separating, 41-42, 616-617
Longre, Laurent, 1057-1062
looping recordsets, 624
loops
array bounds, avoiding hard coding, 57
counters in Next statements, 57
infinite, 473-474, 510
avoiding, 64-65
nested, effect on performance, 558-559
optimizing, 560
performance and, 557
running in Immediate window, 520
low-order bits, 351
lvalue expressions, modifying, 524-525
M
macro security, 1097-1098
macro-free file format, 319-320
macro-optimization, 556-567
binary searches, 563-565
combining sorts and binary searches, 565
order of execution, effect of, 538-539
pre-processing data, 557
QuickSort procedure, 560-563
SORTSEARCH_INDEX UDT, 566-567
tightening loops, 560
macrofun.exe file, 698
macrofun.hlp file, 574
MacroOptions method
(Application object), 112
macros
associating with shapes, 88
XLM functions, 698
XML macros, hiding Ribbon UI using, 294-295
macro_type entry (XLL function table), 1038
MailItem object (Outlook object model), 736
managed automation add-ins creating, 928-933
limitations of, 933-940
managed code, 820
managed COM add-ins, 820, 889
blogs for information, 962
building user interface, 908-927
command bar handling, 909-918
Ribbon user interface handling, 918-927
creating, 891-908
auto-generated references, 897-899
Connection class module, 893-897
project file, creating, 891
registry settings, 899-900
setup projects, 900-908
development tools, 962-963
selecting, 890-891
shimming, 952-961
COM Shim Wizard, 954-961
isolation, 952-953
security, 953-954
time sheet example application, 963-972
VSTO add-ins versus, 984
VSTO project templates versus, 982
managed UDFs, 1006
managed VSTO add-ins, 979
ManagerXLL, 1063
Manifest registry entry, 992
manually installing, 1103
many-to-many relationships, 390-591
MAPI data store, 736
MAPIFolder object (Outlook object model), 736
mapping
numeric IDs to enumeration constants, 1092
to help topic files, 1090
schema fields to DataAdapters, 1075
XSD files, 259-267
margin indicator bar, 513
Mask setting (command bar definition table), 225
masks, creating bitmaps for, 229-230
matching data types, 568
MCommandbars code module (time sheet example application), 158
member processing for collections, 175-177, 771-772
member variables, defined, 33
Members list (Object Browser), 535
memory leaks, 335
avoiding, 182-184
memory management in .NET Framework, 860-861
XLOPER data type, 1049-1054
MEEntryPoints class module
instantiating collections, 174
Terminate method, 184
time sheet example application, 138, 158
trapping events, 179
menu structure for time sheet example application, 158
menus. See also command bars adding to Worksheet Menu Bar, 220-223
combining modeless UserForms with, 423-425, 460-461
customizing in dictator applications, 156
defining, 206
pop-up menus in UserForms, 398-400
sublevels, 198
message boxes debugging, 517
displaying (ActiveX DLL example), 744-747
title bar text, 747
messages, sending between windows, 346-348
method calls, fully qualifying, 712-713
methods, 169-170
adding to CCell class module, 172
to class modules (VB6), 744 of Collection object, 170
defined, 834
GLOBALS class module
creating collections, 171
time sheet example application, 138, 158
micro-optimization, 567-574
comparing alternatives for, 567-568
in Excel, 571-574
in VBA code, 568-571
Microsoft ActiveX Data Objects 2.X Library, referencing, 618
Microsoft Excel 11.0 Object Library, 898
Microsoft HTML Help Workshop. See HTML Help Workshop
Microsoft Jet 4.0 OLE DB Provider, 600, 620
Microsoft Office 12 Access Database Engine OLE DB Provider, 600, 621
Microsoft Office 2003 Web Services Toolkit. See Web Services Toolkit
Microsoft Office Soap Type Library. See Soap type library
Microsoft OLE DB Provider for ODBC, 601
Microsoft OLE DB Provider for SQL Server, 600, 630
Microsoft Outlook Programming (Mosher), 739
Microsoft Scripting Runtime, file system access with, 320-326
Microsoft Virtual PC, 863
Microsoft Visual Basic for Applications Extensibility 5.3 object library, 899
Microsoft Visual Studio Tools for the Office System Power Tools, 1026
Microsoft VSTO portal, 1026
Microsoft XML, 290
Microsoft.Office.Tools namespace, 999
Microsoft.Office.Tools.Excel namespace, 992, 1006
Microsoft.Office.Tools.Excel.v9.0 assembly, 992
Microsoft.Office.Tools.v9.0 assembly, 992
MID() function, 683
MIME types, 1021
modal UserForms, 419
mode-view-controller (MVC), 1009
modeless, forms (VB6) as, 756
modeless UserForms, 420
combining with menu items, 423-425, 460-461
as progress bars, 421-422
time sheet example application, 431
as splash screens, 420-421
modifying database data, 597
in Access databases, 626-629
styles, 82
toolbar, adding style drop-down, 82-83
worksheet UI dynamically, 124-125
module directives, 52-53
module scope in VB.NET, 833-834
Module setting (Watch Context options), 526-527
Module Variables entry (Locals window), 533
module-level comments, 46
modules. See class modules
MOpenClose code module (time sheet example application), 138, 158
MOSS (Office SharePoint Server), 976
Most Recently Used (MRU) file list, 292-293
mouse, setting break points with, 513
mouse coordinates, converting among data and drawing object coordinates, 702
MoveFirst method (ADO Recordset object), 609-610
MoveLast method (ADO Recordset object), 609-610
MoveNext method (ADO Recordset object), 609-610
MovePrevious method (ADO Recordset object), 609-610
MRU (Most Recently Used) file list, 292-293
mssconnect.dll file, 430
msscoree.dll, 896-899, 952
MSDN Library, 332
searching, 346
MSDN Web site, 739, 871
msFILE_ERROR_LOG constant (central error handler), 485
MSForms, 759
MSGraph object library, 733
msoBarBottom enumeration member, 205
msoBarFloating enumeration member, 205
msoBarLeft enumeration member, 205
msoBarNoChangeDock enumeration member, 207
msoBarNoChangeVisible enumeration member, 207
msoBarNoCustomize enumeration member, 208
msoBarNoHorizontalDock enumeration member, 208
msoBarNoMove enumeration member, 208
msoBarNoProtection enumeration member, 208
msoBarNoResize enumeration member, 208
msoBarNoVerticalDock enumeration member, 208
msoBarPopup enumeration member, 205
msoBarRight enumeration member, 206
msoBarTop enumeration member, 206
msoButtonDown enumeration member, 212
msoButtonAutomatic enumeration member, 212
msoButtonCaption enumeration member, 212
msoButtonDown enumeration member, 217
msoButtonIcon enumeration member, 212
msoButtonIconAndCaption enumeration member, 212
msoButtonIconAndCaptionBelow enumeration member, 212
msoButtonIconAndWrapCaption enumeration member, 212
msoButtonIconAndWrapCaptionBelow enumeration member, 212
msoButtonMixed enumeration member, 217
msoButtonUp enumeration member, 217
msoButtonWrapCaption enumeration member, 212
msoComboLabel enumeration member, 213
msoComboNormal enumeration member, 213
msoControlButton enumeration member, 211
msoControlComboBox enumeration member, 211
msoControlDropDown enumeration member, 211
msoControlEdit enumeration member, 211
msoControlPopup enumeration member, 211
MSQuery, 665
msSILENT_ERROR constant (central error handler), 484
mssoap30.dll, 1068
MStandardCode code module (time sheet example application), 138, 158
MSystemCode code module (time sheet example application), 138, 158
MUtilities code module (time sheet example application), 138
MVC (mode-view-controller), 1009
MWorkspace code module (time sheet example application), 138
MZTools, 542, 871

N
N() function, 681
Name argument (ADO CreateParameter method), 628
named constants, 72-73
named formulas, 76-77
named instances, SQL Server name, 642
named ranges, 73-75
NamedRange control, 1007-1012
names for function library add-ins, creating, 115-116
namespace aliases, 858
NameSpace object (Outlook object model), 736
namespaces, 818
importing, 827-828
for shared tabs, 280
in XML, 253, 270-271
naming controls in UserForms, 384
naming conventions, 27-40
for defined names, 39
for embedded objects, 38
for Excel UI elements, 37-39
exceptions to, 39-40
fully qualified object variable names, 56
for modules, classes, UserForms, 36
for procedures, 35-36
sample of, 28-34
for shapes, 37-38
for VSTO, 112
for Visual Basic Projects, 37
for worksheets, chart sheets, 37
native VSTO templates, 979
NativeWindow class, 971
natural keys, 592-594
nested exceptions, 533
nested loops, effect on performance, 558-559
.NET data providers, 865
.NET Framework, 817-819, 975.
See also ADO.NET; VB.NET; VS.NET; VSTO
automation and, 855-863
early binding/late binding, 863
Excel object usage, 855-862
PIA (Primary Interop Assembly), 856-857
client version, 819
managed COM add-ins development tools, 962-963
shimming, 952-961
time sheet example application, 963-972
resources for information, 870-871
time sheet example application, 872-877
versions of, 818-819
.NET Language Integrated Query (LINQ), 570
network group membership, checking, 1095-1096
network-related API calls. See file system-related API calls
New keyword, 717
newsgroups, 1062
Next statements, loop counters in, 57
NextRecordset method (ADO Recordset object), 610
“No Help Available” help file, creating, 1088
non-basic code entry in Call Stack window, 522
non-key columns defined, 580
usage of, 585
normalization, 579-587
exceptions to, 586-587
first normal form, 580-581
second normal form, 582-583
third normal form, 584-586
Northwind sample database, 615-616
NoteItem object (Outlook object model), 736
Nothing keyword (VB.NET), 836
null-terminated strings, 1036, 1047
Num Lock key, checking state of, 350-351
Number property (Err object), 466
availability of error numbers, 474
number sequences, generating, 682
numbered lines adding to code, 823
displaying in XML Editor, 921
numeric data in XLOPER data type, 1047
numeric IDs, mapping to enumeration constants, 1092
to help topic files, 1090
NUM_REGISTER_ARGS constant (XLL function table), 1036
O
Object Browser, 533-537, 847-848
Object data type (VB.NET), 835, 839
Object libraries
Access object model, 726-729
Application object model, 726
DAO.Database object, 726
DoCmd object, 727
example application, 727-729
Application object, 726
forward compatibility, 713-715
fully qualifying property/method calls, 712-713
including in variable declarations, 711-712
MSGraph object model, 733
Outlook object model, 736-739
Application object, 736
text object, 737-739
items collection, 736
MAPIFolder object, 736
NameSpace object, 736
PowerPoint object model, 732-735
Application object, 732
charts in, 733
text object, 733-735
Presentation object, 732
Text object, 732
Slide object, 732
referring to, 710-711
setting references in ActiveX DLLs, 747
Word object model, 729-732
Application object, 729
Bookmark object, 729
Document object, 729
text object, 730-732
Range object, 729
Object Library box (Object Browser), 534
Object Linking and Embedding (OLE), 710
object models
ADO, 599
Excel, 7
referencing, 776
XML, 269-267
object oriented C++ wrapper for Excel C API, 1063
object references casting, 859
removing, 182-184
object types, validating, 65
object variables declaring and initializing, 55
early binding versus late binding, 59-61
fully qualified names, 56
performance advantages of, 571
object-oriented programming.
   See OOP
objects. See also class modules;
   interfaces
code reuse and, 435-437
   collections, creating, 170-177,
   771-772
creating, 166-168
   class modules as template
for, 165-170
reasons for, 165
default properties, calling, 63
   application-level event hand-
   ling, 190-193
   raising, 180-188
   trapping, 177-182
initializing, 174
   instances
   creating, 836-837
   VB.NET, 836-837
   instantiating, 174
   interfaces and, 433-434
   in .NET solutions, 857-862
ObjectStateEnum constants,
values for, 601
ODBC, 599
Office 2003 PIAs, 904
Office 2007 Compatibility
   Pack, 163
Office 2007 cross-version applica-
   tions. See cross-version applica-
   tions
Office 2007 CustomUI
   Editor, 276
Office = Microsoft.Office.Core
   namespace, 992
Office Application Clients, 976
Office applications
   class names for CreateObject
function, 718
controlling. See controlling appli-
cations
   object libraries. See object
libraries
registering, 719
resources for information, 739
Office button, 292
Office Developer Center, 739
Office Development with Visual
   Studio blog, 1026
Office Fluent User Interface
   Developer Portal, 301
Office Forms Service, 976
Office Open XML (OOXML),
   273-277
Office PerformancePoint
   Server, 976
Office product suite. See Office
   Application clients
Office server-side programming,
   981
Office Servers, 976
Office SharePoint Server, 976
Office System, 976-977
Office versions, running multiple
   with VSTO, 983
OFFSET function, 693-696
OLE (Object Linking and
   Embedding), 710
OLE DB, 599
OLE DB providers, specifying,
   600-601
for Access databases, 620
OLE in-place activation, 980
On Error Goto <Label> state-
   ment, 467-470
On Error GoTo 0 statement, 472
On Error Resume Next state-
   ment, 470-472, 489
disabling, 509
On Error statements, 469-472
   On Error GoTo <Label>, 470
   On Error GoTo 0, 472
   On Error Resume Next, 470-472
onAction attribute (Ribbon con-
   trols), 304
onAction callback for Ribbon UI,
   314, 316
OnAction setting (command bar
   definition table), 209
OnAddInsUpdate event
   (AddinInstance object), 794
OnBeginShutdown event
   (AddinInstance object), 794
OnConnection event
   (AddinInstance object), 792-793
OnConnection event procedure
IDTExtensibility2 interface,
   893
   managed COM add-ins, 910
OnDisconnection event
   (AddinInstance object), 794
OnDisconnection event procedure
IDTExtensibility2 interface,
   893
   managed COM add-ins, 912
one-to-many relationships,
   589-590
one-to-one relationships,
   588-589
one-way communication example
   (ActiveX DLLs), 744-747
onLoad callback for Ribbon UI,
   314-316
OnStartupComplete event
   (AddinInstance object), 793
OOP (object-oriented
   programming)
   ActiveX DLL support for,
   769-772
   VB.NET support for, 820
   VBA and, 5
OOXML (Office Open XML),
   273-277
Open method (ADO Connection
   object), 602
Open method (ADO Recordset
   object), 610-612, 624
Open XML, 273-277, 1009
Open XML Package Editor, 1027
OpenCurrentDatabase method
   (Access Application
   object), 726
opening
   applications (time sheet exam-
   ple application), 125-127
   user interface workbooks (time
   sheet example applica-
   tion), 128-130
OPEN entries (registry keys), 78
OpenXMLDeveloper.org site, 301
OPER data type, 1049-1050
operational requirements, reasons for using databases, 578
operations, defined, 6
operRes parameter (Excel4 function), 1050
optimization. See also performance
macro-optimization, 556-567
binary searches, 563-565
combining sorts and binary searches, 565
order of execution, effect of, 558-560
pre-processing data, 557
QuickSort procedure, 560-563
SORTSEARCH_INDEX UDT, 566-567
tightening loops, 560
micro-optimization, 567-574
comparing alternatives for, 567-568
in Excel, 571-574
in VBA code, 568-571
Option Base 1 statement, 53
Option Compare Binary, Option Compare Text versus, 569
Option Compare setting (VB.NET development settings), 822
Option Compare Text statement, 53
Option Compare Binary versus, 569
Option Explicit setting (VB.NET development settings), 822
Option Explicit statement, 52
Option Inher setting (VB.NET development settings), 822
Option Private Module directive, 52, 115
Option Strict setting (VB.NET development settings), 822
OR operations in criteria ranges, 676
ORDER BY clause (SQL SELECT statement), 596
order of 1 (procedure processing time), 558
order of execution, effect on performance, 558-560
order of N (procedure processing time), 558
order of N^2 (procedure processing time), 558
organization of applications (timesheet example application), 138
organization of data. See data structures
out-of-process communication, 774
Outlook, referring to instances of, 720-721
starting instances of, 813
Outlook object library, 736-739
Application object, 736
example application, 737-739
Items collection, 736
MAPIFolder object, 736
NameSpace object, 736
Output window, 849
P
packaging applications, 1099
add-in installation requirements, 1100-1102
installation location, selecting, 1099-1100
installation with Windows Installer, 1104
manual installation, 1103
Setup.xls workbook installation, 1103
template installation requirements, 1100
PAGE.SETUP.XLM function, 574
PageSetup object, performance and, 574
parameter refreshing (SQL Server), 637-638
Parameter setting (command bar definition table), 217
ParameterDirectionEnum constant values, list of, 606
Parameters collection (ADO Command object), 607, 628, 637
parameters for Excel4 function, 1050
parent windows, changing, 756
Part items (in Open XML), 275
Pascal strings, 1036, 1047
passing data with user-defined types, 620
strings to API calls, 356
passwords
Excel security, 1094-1095
securing, 783
paste functionality, handling, 154-156
Paste Special toolbar example, 235-241, 797
pasting controls, 762
Path property (auto-generated references), 898
paths, changing to UNC paths, 356-357
PeekMessage API call, 352-354
PerfMon utility, 546-551
PerfMonitor.dll, 547
PerfMonOffice.dll, 547
PerfMonVB6.dll, 547
performance. See also optimization
cross-process calls, 723-725
early bound object variables, 60
improving by creative thinking, 551-556
asking questions, 556
breaking the rules, 554-555
data knowledge of, 553-556
jigsaw puzzle example, 551-554
<table>
<thead>
<tr>
<th>Physical Design of Data Access Tier, 617-620</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIA (Primary Interop Assembly), 856-857</td>
</tr>
<tr>
<td>Office 2003 PIAs, 904</td>
</tr>
<tr>
<td>Picture Setting (command bar definition table), 228</td>
</tr>
<tr>
<td>physical design of data access tier, 617-620</td>
</tr>
<tr>
<td>PIA (Primary Interop Assembly), 856-857</td>
</tr>
<tr>
<td>Office 2003 PIAs, 904</td>
</tr>
<tr>
<td>Picture setting (command bar definition table), 228</td>
</tr>
</tbody>
</table>

**Index**

- “Think outside the box” example, 552-554
- Tools, knowledge of, 556
- Macro-optimization, 556-567
- Binary searches, 563-565
- Combining sorts and binary searches, 563
- Order of execution, effect of, 558-560
- Pre-processing data, 557
- QuickSort procedure, 560-563
- SORTSEARCH_INDEX UDT, 566-567
- Tightening loops, 560
- Micro-optimization, 567-574
- Comparing alternatives for, 567-568
- In Excel, 571-574
- In VBA code, 568-571
- PerfMon utility, 546-551
- Target response times, 545
- Tricks for illusion of, 546
- Permanent assertions, 541
- Personal Information Exchange (.pfx) files, 955

**PETRAS. See time sheet example application**

**PETRAS Report Activities Consultants.xls, 874**

**PETRAS Report Activities.xlt, 874**

**PETRAS Report Activities.xlt, 874**

**PETRAS Report Activity.xlt, 874**

**PETRAS Report Tool.NET (time sheet example application), 817, 872-857**

**converting to managed COM add-in, 963-972**

**PETRAS.asmx, 1073**

**PetrasAddin.xla, 157, 369**

**PetrasConsolidation.xlt, 157, 369**

**PetrasIcon.ico, 369**

**PetrasReporting.xla, 157, 369**

**PetrasTemplate.xla, 157**

**PetrasTemplate.xlt, 369**

**.pfx files, 955**

**PowerPoint object library, 732-735**

**Application object, 732**

**charts in, 733**

**example application, 733-735**

**Presentation object, 732**

**Shape object, 732**

**Slide object, 732**

**pre-processing data for performance optimization, 557**

**precedence tree, defined, 6**

**prefixes**

- Data types, 29-30
- Defined names, 39
- Drawing objects, 38
- Embedded objects, 38
- Preparing background graphics for user interface, 151-153

**Prerequisites**

- Comparing versions, 897
- For setup projects, 902-904
- Presentation layer, worksheets as, 4-5
- Presentation object (PowerPoint object model), 732
- Presentations collection, 721, 732
- Pressed keys, testing for, 352-355
- Preventing importing XML results, 269
- Primary axes (charts), 690
- Primary Interop Assembly (PIA), 856-857
- Office 2003 PIAs, 904
- Primary keys defined, 580
- Natural versus artificial, 592-594
- Printer object (VB6), 773
- Private instanting type, 769
- Private keyword (VB.NET), 833
- Pro SQL Server 2005 Database Design and Optimization (Davidson, Kline, Windisch), 613

**Procedural Programming Best Practices**, 43-45

- Arguments, limiting, 45
- Business logic isolation, 44
duplicate code, eliminating, 44
encapsulation, 44
functional decomposition, 43
modules, organizing code in, 43
size limits on procedures, 44
procedure entry (XLL function table), 1037
procedure error handlers, 477-480
Procedure setting (Watch Context options), 527
procedure-level comments, 46-47
procedures. See also functions; subroutines
adding PerfMon utility calls to, 548-549
arguments
declaring, 62-63
validating, 63
naming conventions, 35-36
order of execution, effect on performance optimization, 558-560
property procedures, 168-169
trivial procedures, 480-481
wrapping in On Error Resume Next statement, 471
processing data. See data processing
processing in dictator applications, 156
Professional ADO 2.5 Programming (Sussman et al.), 613
professional Excel developers, defined, 4
Professional Excel Development Web site, 12
Professional Excel Timesheet Reporting and Analysis System. See time sheet example application
Professional SQL Server 2005 Programming (Vieira), 648
ProvDl limitations of, 938
managed COM add-ins registry keys, 900
program columns
defined, 70-71
time sheet example application, 102
program execution, canceling, 484-485, 491-495
program listings. See listings
program rows
defined, 70-71
time sheet example application, 102
programming languages
declarative, worksheet functions as, 6-7
in VS.NET, 819
Programming Microsoft Visual Basic .NET Version 2003 (Balena), 870
progress bars, 421-422
custom interface example, 449-460
time sheet example application, 431, 462
when to display, 546
project protection. Stop statements and, 510-511
project templates (VSTO), 977, 979-983
application-centric, 979
document-centric, 979-981
installing and running, 982
managed COM add-ins versus, 982
selecting, 981
projects. See also Visual Basic Projects
ActiveX DLL projects, creating, 742, 744
managed COM add-ins, creating, 891
versions of, saving, 66
XLLs, creating, 1030-1034
PromptingLevel subkey, 1018
properties
default collections properties, 175-177, 771-772
default object properties, calling, 63
testing before setting, 573
VB.NET classes, 946-947
property calls, fully qualifying, 712-713
Property Get procedures, 169
Property Let procedures, 169
property procedures, 168-169
Property Set procedures, 169
Protected Friend keyword (VB.NET), 834
Protected keyword (VB.NET), 834
protected projects, Stop statement and, 510-511
protection (worksheet UI setting), 119
Protection setting (command bar definition table), 207-209
providers (OLE DB), 590
specifying, 600-601
for Access databases, 620
Public keyword (VB.NET), 833
Public profile (Windows Vista), 329
public variables, 58-59
PublicNotCreatable instancing type, 769
publishing
VSTO workbooks, 1022-1024
Web Services, 1068
pxlInput argument (PxlCoerce function), 1053
Q
Q&A forums for VB.NET information, 871
QAT (Quick Access Toolbar), 292
Qualified ID, 281
queries. See SQL
QueryClose event (UserForms), 396
QueryTables, 664-667
calculated fields/items
question mark character (?), variable evaluation in
Immediate window, 519
questioning assumptions (creative thinking), 556
questions, resources for information, 11-12
Quick Access Toolbar (QAT), 292
Quick Watch window, 531-532, 850
QuickSort procedure, 560-563
quiet mode for UAC (User
Account Control), 328

R
Raise method (Err object), 474
Raise property (Err object), 466
raising
custom errors, 474, 484
events, 180-188
RAM, determining current
usage, 183
Range object
CopyFromRecordset
method, 624
Word object model, 729
ranges
in advanced filters, 674
array formulas and, 680-683
converting to Lists, 664
data consolidation, 672-673
named ranges, 73-75
reading/writing, 572-573
structured ranges, 662-663
formulas in, 667
unstructured ranges, 662
RCW (Runtime Callable
Wrapper), 860
RDBMS (relational database
management systems), 977
re-throw system (error
handling), 501
Recent documents list, clearing,
292-293
recompiling, See compiling
Recordset object (ADO), 607-612
BOF property, 607
Close method, 609
CursorLocation property, 608
disconnected recordsets, 640
EOF property, 607, 624
events, 612
Fields collection, 612
Filter property, 608
Move methods, 609-610
multiple recordsets, 639
NextRecordset method, 610
Open method, 610, 612, 624
Sort property, 608
recordsets
disconnected recordsets,
640-642
looping, 624
multiple recordsets, 638-640
Recycle Bin, deleting files to,
360-361
redimensioning arrays, 841
Reference Name property
(auto-generated references), 897
references
auto-generated references for
managed COM add-ins, 897-899
to existing application
instances, 720-721
setting in ActiveX DLLs, 747
in VSTO add-ins, 991-992
referencing
ActiveDocument (Word), 712
ActiveX DLLs, 745-746
collections in For...Each
loops, 176
Excel object library, 776
Microsoft ActiveX Data
Objects 2.X Library, 618
object libraries, 710-711
sheets by CodeNames, 65
referential integrity, 587-592
DRI (Declarative Referential
Integrity), 644
refreshing
advanced filters, 675
parameters (SQL Server),
637-638
QueryTables, 665
screen, disabling, 571
regsvr32.exe, 900, 940
regions, 853
registering
ActiveX DLLs, 744
COM add-ins, 790
manually, 940
COM shim DLL files, 959
custom worksheet functions,
1054-1057
Office applications, 710
UDFs with Excel Function
Wizard, 112-114
Registry Editor, 905
User/Machine Hive registry
section, 958
registry keys
Add-in Designer, 788-790, 1102
Add-in Manager, 1100
managed automation
add-ins, 929
managed COM add-in setup
projects, 905-906
managed COM add-ins, 899-900
VSTO add-ins, 992-993
VSTO security, 1018
regsvr32 command, 960
regsvr32.exe, 791
related windows, finding with
API calls, 343-346
relational database management
systems (RDBMS), 977
relational databases, 578-579. See
also databases
Relationship items (in Open
XML), 275
relationships, 587-592
in criteria ranges, 678
many-to-many, 590-591
one-to-many, 589-590
one-to-one, 588-589
in XML, 250
relative named ranges
defined, 73
types of, 73
release builds, 851
RemoveRange method (VB.NET
arrays), 844
removing
break points, 512-513
function registrations,
1054-1057
object references, 182-184
right-click menus, 294
reports. See also PETRAS
Report Tool.NET
(time sheet example
application)
in dictator applications, 157
time sheet example application, 157-160, 162-163
central error handler for, 499-506
database handling, 656-659
multiple document interface, 194
progress bars, adding, 431
Shift key, checking state of, 371-373
table-driven command bars, 243-247
requirements
for add-ins, 783
desktop environment requirements, when to use
VSTO, 984
of dictator applications, 21-22
for installation
of add-ins, 1100-1102
of templates, 1100
resizing UserForms, 403-404
resolutions, screen
adapting UserForms to, 402-403
reading, 337-338
resource DLLs, 790
resource files, 773
adding bitmaps to, 804-806
adding to projects, 802-804
loading bitmaps from, 806-807
resources for information, 11-12
databases, 613-614, 647-648
managed COM add-ins, 962
.NET Framework, 870-871
Office applications, 739
VSTO, 1026
XLLs, 1062-1063
resources in VB.NET solutions, 863-864
response times, targets for, 545
restoring
toolbar customizations, 147
user settings, 143-148
Results application context, 244
results of PerfMon utility, importing, 350
results presentation. See charts; reports
Resume <Label> statement, 474
single exit point, implementing, 475
Resume Next statement, 473
Resume statements, 472-474
debugging with, 473
Resume <Label>, 474
Resume Next, 473
retrieving
data
with ADO.NET, 864-870
from data access tier, 620
time sheet example application, 882-883
database data, 595-596
from Access databases, 622-625, 650-652
reusing code, 435-437
reusing variables, avoiding, 54
RHS variable name, 445
Ribbon designer tool, 890
Ribbon IDs Tool window, 1027
Ribbon UI, 273, 909
best practices, 278-289
Add-Ins tab, 279
custom image management, 284-286
global callback handlers, 286-287
invalidation, 287-289
keytips, 284
shared tabs, 279-284
work processes support, 278-279
combining with command bars, 304
heavy weight design, 307-308, 310-319
light weight design, 304-307
creating
for dictator applications, 291-294
loader add-in for, 312-318
in managed COM add-ins, 918-927
customUI folder, 277
hiding, 294-295
sheet navigation in, 296-298
sizing comboBox controls, 295-296
table-driven customization,
289-291
template creation, 299
websites for information, 300
Ribbon Visual Designer, 995-998
toggle buttons for custom task
panes (CTPs), 1003
RibbonX, 273-274
for light weight UI design, 306
RibbonX: Customizing the
Office 2007 Ribbon
(Martin et al), 300
right-click command bar example,
226-228
right-click menus, removing, 294
Roaming folder (Windows Vista), 329
roaming user profiles, 329
robustness of custom
interfaces, 448
Roman numeral conversion
example, 775-780, 782
root element (XML), 252
row headers, worksheet UI setting, 120
ROW() function, 683
row-relative named ranges, 73
rows
adding to user interface workbooks (time sheet example application), 133-134
duplicate rows in
databases, 580
hidden rows (worksheet UI setting), 119
program rows
defined, 70-71
time sheet example application,
102
Ruby Forms, 759
rules, breaking (creative thinking), 554-555
Run function (Application object), 108
run mode, break mode
versus, 507
running VSTO add-ins, 993-995
Runtime Callable Wrapper (RCW), 860
runtime errors. See also error handling
Err object, 466–467
types of, 463
runtime stage (applications), 109
runtime versions of ActiveX controls, 760

S
Sams Teach Yourself SQL in 10 Minutes (Forta), 614
satellite DLLs, 790
SaveFileDialog component, 836, 877
SaveSetting property (ThisWorkbook object), 17
saving
debugging output files, 849
project versions, 66
user interface workbooks (timesheet example application), 131–133
schema validation for Ribbon user interface, 920
schemas, creating DataSets from, 1073. See also XSD
scope
defined names, 77–78
error handlers, 468–469
properties (VB.NET), 947
variables, 38–39
VB.NET, 833–834
watch expressions, 526–527
scope specifiers in naming conventions, 29
Screen object (VB6), 774
screen refresh, disabling, 571
screen resolution
adapting UserForms to, 402–403
reading, 337–338
screen-related API calls, 337
pixel size, determining, 338–340
screen resolution, reading, 337–338
screentips, enabling, 823
ScreenUpdating property (Application object), 571
scroll area (worksheet setting), 119
Scroll Lock key, checking state of, 350–351
scroll regions in dynamic UserForms, 415
scrolling in time series, 694
SDK, 333. See also Excel 2007 SDK
Search combo box (Object Browser), 535
searching
arrays
binary searches, 563–565
VB.NET, 842
MSDN Library, 332, 346
second normal form, 582–583
secondary axes (charts), 690
security
code protection in VB6, 758
COM add-ins, 798
digital signatures, 1097–1098
encrypting passwords, 783
Excel, 1094–1095
macro security, 1097–1098
managed COM add-ins, 953–954
network groups, checking membership, 1095–1096
Public profile, 329
SQL Server databases, 631
standard user accounts, 328–329
User Account Control (UAC), 326–328
VB6 EXE front loaders as, 783
VSTO, 1016–1019
Internet security zone, 1019–1020, 1022
SEH (structured exception handling), 818, 829–833
Select Case statement
error handling and, 480
performance and, 571
SELECT clause (SQL SELECT statement), 595
SELECT statement (SQL), 595–596
for Access databases, 622–625
time sheet example application, 650, 652
selecting
application architectures, 13, 24–25
file formats, 275
installation location, 1099–1100
managed automation add-ins in Add-in Manager, 938–940
VSTO project templates, 981
Selection property, performance and, 573
selections
object type of, validating, 65
performance and, 571–572
self-automated workbooks, 15–16
sending messages between windows, 346–348
SendMessage API call, 346, 352
separating data and application with XML, 254, 256
separator bars, 198
creating, 215
SERIES function, 691–692
server-side programming for Office, 981
server-side VSTO solutions, 1008–1009
ServerDocument class, 981, 1009
Set blocks (VB.NET properties), 946–947
SET clause (SQL UPDATE statement), 597
Set Next Statement command, 516–517
Set Transparent Color control, 214
SetCurDir API call, 337
SetIcon API call, 348
settings management (worksheet UI), table-driven approach to, 118–124
settings, storing and restoring, 143–148
setup projects
creating in COM Shim Wizard, 956, 958
for managed COM add-ins, 900-908
setup solutions, 893
Setup.xls installation workbook, 1103
SetWindowLong API call, 349
Shape object (PowerPoint object model), 732
shapes, 87-88
naming conventions, 37-38
Shared Add-in Template, 889-891
creating automation add-ins, 928
Shared Add-in Wizard, 891, 893
Connection class module, 893-897
creating automation add-ins, 928
shared tabs for add-ins, 279-284
SHBrowseForFolder API call, 363-368
sheet navigation in Ribbon UI, 296-298
sheet visibility (worksheet UI setting), 120
sheets, See chart sheets; worksheets
SHFileOperation API call, 360-361
SHGetFolderPath API call, 358, 360
Shift key, checking state of, 350-351
time sheet example application, 371-373
Shift+F2 keyboard shortcut (procedure definition), 543
Shift+F8 keyboard shortcut (Step Over command), 515, 542
Shift+F9 keyboard shortcut (Quick Watch window), 531, 543
shimming managed COM add-ins, 952-961
COM Shim Wizard, 954-961
isolation, 952-953
security, 953-954
shims, 889
Short data type (VB.NET), 839
shortcut keys. See accelerator keys; keyboard shortcuts
Shortcut Text setting (command bar definition table), 216
shortcut_text entry (XLL function table), 1038
Show Hidden Members setting (Object Browser), 535
ShowWindowsInTaskBar property (Application object), 21
shutdown code, On Error Resume Next statement in, 472
shutdown process for dictator applications, 142-151
restoring user settings, 143-148
shutdown stage (applications), 110
Sign Tool, 960
signatures, 953, 960-961, 1097-1098
signing deployment manifest, 1022
silent errors, 484
simple error handling, 475-476
simplicity in UserForms design, 375-376
simulating splitter bars in UserForms, 405-406
single exit point principle, 475
Sort property (ADO Recordset object), 374
sorting arrays combining with binary searches, 565
time sheet example application, 873-874, 876
SortSearch Index UDT, 566-567
Source property (Err object), 467
special folders, locating, 357-360
splash screens, 420-421
splitter bars, simulating in UserForms, 405-406
 spreadsheets. See worksheets
Spy++ utility, 340
SQL (structured query language), 594
DELETE statement, 597-598
for Access databases, 629-630
INSERT statement, 596-597
for Access databases, 625-626, 652-656
SELECT statement, 595-596
for Access databases, 622-625, 650-652
UPDATE statement, 597
for Access databases, 626-629
Command window, 548
conditional compilation constants, 851-852
Error List window, 548
Exception Assistant, 846-847
Exceptions dialog, 850-851
Immediate window, 549
keyboard shortcuts, setting, 845
Object Browser, 847-848
Output window, 549
unmanaged code, enabling debugging, 846
Watch/Quick Watch windows, 850
resources in, 863-864
time sheet example application, 573-574, 576
Sort property (ADO Recordset object), 608
sorting arrays combining with binary searches, 565
time sheet example application, 873-874, 876
SortSearch Index UDT, 566-567
Source property (Err object), 467
special folders, locating, 357-360
splash screens, 420-421
splitter bars, simulating in UserForms, 405-406
 spreadsheets. See worksheets
Spy++ utility, 340
SQL (structured query language), 594
DELETE statement, 597-598
for Access databases, 629-630
INSERT statement, 596-597
for Access databases, 625-626, 652-656
SELECT statement, 595-596
for Access databases, 622-625, 650-652
UPDATE statement, 597
for Access databases, 626-629
SQL Native Client, 601
SQL Server databases
advantages of, 630
connecting to, 630-631
connection pooling, 632-633
default instances versus named
instances, 642
disconnected recordsets,
640-642
error handling connections,
633-635
multiple recordsets, 638-640
Northwind sample database,
installing, 615
parameter refreshing, 637-638
security types, 631
stored procedures, 635-637
upsizing Access databases to,
642-646
Stack class (VB.NET), 844
standard format, XML as, 250
standard security for SQL Server
databases, 631
standard user accounts, 328-329
starting
add-ins, 784
multiple instances of
Excel, 781
Outlook instances, 813
PowerPoint instances, 721
Word instances, 813
XLLs, 1040-1041
starting point
for dynamic lists, 77
for relative named ranges, 73
startup process for dictator appli-
cations, 142-151
environment modifications,
148-151
storage user settings, 143-148
version and dependency
checks, 142-143
startup stage (applications),
108-109
State property (ADO Connection
object), 601
State setting (command bar
definition table),
217-218, 222
states, checking key states with
API calls, 350-351,
371-373
StaticData.XSD, 1073
step charts, creating, 699-701
Step Into command, 514-515
Step Out command, 515
Step Over command, 515
Step to Cursor command, 516
stepping through code, 513-516
Step Into command, 514-515
Step Out command, 515
Step Over command, 515
Step to Cursor command, 516
Stop statement, 510-511
stored procedures (SQL Server),
635-637
StoreTimeSheet function (Web
Services time sheet exam-
ple application), 1073
storing
toolbar customizations, 147
user settings, 143-148
strict type checking early bound
object variables, 60
string versions (string-handling
functions), 569
string-handling functions, variant
versus string versions, 569
strings
C-strings, 1036
checking length of, 569
length prefixes, 1043
Option Compare Text, avoid-
ing, 569
Pascal strings, 1036
passing ByVal versus
ByRef, 569
passing to API calls, 356
string-handling functions, vari-
ant versus string versions,
569
in VB.NET, 839
in XLOPER data type, 1047
Strong Name Key (.snk) files, 955
strong names, 856
creating, 954-955
explained, 954
strong typing, 822
strongly typed format,
XML as, 250
structure of dictator applications,
141-142
structured exception handling
(SEH), 818, 829-833
structured format, XML as, 249
structured query language.
See SQL
structured ranges, 662-663
formulas in, 667
structures, API call usage,
352-355
styles, 78-83
adding drop-down to toolbar,
82-83
creating custom, 79-81
modifying, 82
time sheet example applica-
tion, 103
window styles
changing, 349
modifying for UserForms,
392-396
Sub New (constructors), 943
subroutines
code comments in, 46
naming conventions, 35-36
subset UserForms as
dynamic, 411
sum of digits calculation, 682
SUM() function, 683
SUMIF function, 680
supported versions of Excel, 9-10
supporting
debug mode, 149-151
XML, 269
switching. See casting
System.EnterpriseServices
namespace, 855
System.IO namespace, 922
System.Reflection
namespace, 921
System.Runtime.InteropServices
namespace, 855, 893
System.Windows.Forms name-
space, 986, 1008
T
T-SQL, 635
tab order for controls
in UserForms, 386
setting, 826
Table of Contents file
(in help files), creating,
1088, 1091
table-driven command bar builder, 199-200
table-driven command bars, 199-219
associating icons with controls, 225-232, 796
command bar definition table, 200-219
custom menu with submenus example, 220-223
custom right-click command bar example, 226-228
custom toolbar example, 223-226
event hooks, 232-241, 795
table-driven command bar builder, 199-200
time sheet example application, 241-247
table-driven dynamic wizards, 411-415
table-driven methodology defined, 119
to worksheet UI settings management, 118-124
table-driven Ribbon UI customization, 289-291
TableDefs collection, 726
tables
dynamic tables, creating with conditional formatting, 93-94, 96
formatting, 85-86
Tag property, resizing
UserForms, 403
Tag setting (command bar definition table), 216-217, 233-235
tags (XML), 252
target applications, 710
target response times, 545
targeting Excel versions for managed COM add-ins, 909
Task List, 855
task panes, custom task panes (CTPs), 996-1006
TaskItem object (Outlook object model), 736
technical support, resources for information, 11-12
template workbooks for application-specific add-ins, 18
creating (time sheet example application), 189-190
templates
class modules as, 168-170
creating in Custom UI Editor, 299
installation requirements, 1100
native VSTO templates, 979
project templates (VSTO), 977-983
application-centric, 979
document-centric, 979-981
installing and running, 982
managed COM add-ins versus, 982
selecting, 981
VSTO templates. See workbooks (VSTO)
Terminate event
AddinInstance object, 794
error handling, 489
On Error Resume Next statement in, 472
Terminate method, 182-184
test harnesses
building, 537-540
defined, 64
testing
for key presses, 352-355
properties before setting, 573
text boxes
ComboBox control as, 426
performance expectations for, 546
text editors, collections, default properties and member processing, 176-177
text files, importing code from, 855
Thawte, 1097
"think outside the box" example (creative thinking), 552-554
third normal form, 584-586
ThisWorkbook object
events, 784
GetSetting property, 17
IsAddin property, 17, 115
SaveSetting property, 17
threading
with COM add-ins, 798-799
multithreading (time sheet example application), 879-882
Throw statement (structured exception handling), 833
time series, scrolling/zooming in, 694
time sheet example application, 8-9, 100-101
adding data to Access database, 652-656
application organization, 138
application-specific add-ins, 125-137
borders, 104
cell comments, 104
central error handler for, 496-506
conditional formatting, 105
connecting to Access databases, 648-649
custom interfaces, 462
data validation, 104
database handling changes in reporting application, 656-657, 659
defined names, 102-103
event handling class module, 190-193
folders, browsing for, 369, 371
heavy weight cross-version UI design, 307-308, 310-319
hidden rows/columns, 102
menu structure, 138
multiple document interface, 194
PETRAS Report Tool.NET, 817, 872-887
converting to managed COM add-in, 963-972
progress bars, adding, 431
reporting application for, 157-163
retrieving data from Access database, 650, 652
Shift key, checking state of, 371-373
styles, 103
table-driven command bars, 241-247
template workbooks, creating, 189-190
Web Services, 1072-1083
Timer calls, 568
TimeSheet.XSD, 1073
timestamping digital signatures, 1098
timestamps, 961
title bar text in message boxes, 747
TlbImp.exe, 857
toolbar buttons, disabling, 192
Toolbar List command bar, disabling, 208
toolbars
building in managed COM add-ins, 909-918
time sheet example application, 127
custom toolbar example, 223-226
customizing in dictator applications, 156
storing and restoring customizations, 147
Debug toolbar, displaying, 514
deleting, 127
docking, 198
Paste Special toolbar example, 235-241
style drop-down, adding, 82-83
tools, knowledge of (creative thinking), 536
ToolTip component, 577
Tooltip setting (command bar definition table), 216
top-level windows, 342
forms (VB6) as, 756
topic files (in help files)
creating list of, 1089-1090
displaying from VBA, 1092-1094
ID numbers for, 1090
in enumerations, 1092
mapping, 1090
introductory file, creating, 1088
“No Help Available” file, creating, 1088
topics (help files), 1086
ToShort method (exception handling), 832
total rows in Lists, 664
Transact SQL, 635
transferring VB6 applications to VB.NET, 820
TranslateMessage API call, 354
transparency in image files, 925
transparent backgrounds for icons, setting, 214-215
trapping errors, 480
settings for, 507-508
trapping events, 177-182
TrinToSize method (VB.NET arrays), 844
trivial procedures, 480-481
troubleshooting getEnabled callback, 288
Trust Center in Office, 1017
Trusted Publishers certificate store, 1018
Trusted Root Certification Authority certificate store, 1018
TrustManager registry key, 1018
Try statement (structured exception handling), 830
TweakUAC, 328
twips, defined, 338
two-way communication example (ActiveX DLLs), 747-751
Type argument (ADO CreateParameter method), 628
type library, 717. See also object libraries
Type mismatch errors, 433
Type property (auto-generated references), 897
TypeName() function, 447
type_of function, 447
type_text entry (XLL function table), 1037-1038
U
UAC (User Account Control), 326-328
UDFs (user-defined functions), 110-117. See also function library add-ins
category numbers for, 113
critical details, 116-117
disadvantages of, 117
topics of, 110-112
managed UDFs, 1006
naming conventions, 112
registering with Excel
    Function Wizard, 112-114
UDTs (user-defined types) naming conventions example, 33
    passing data with, 620
    watching, 529-531
UI (user interface). See also UI design
for cross-version applications, 304
heavy weight design, 307-308, 310-319
light weight design, 304-307
customizing for dictator applications, 151-156
for managed COM add-ins, building, 908-927
Ribbon UI, 273
    best practices, 278-289
    creating for dictator applications, 291-294
    creating loader add-in for, 312-318
    customUI folder, 277
    hiding, 294-295
    sheet navigation in, 296-298
    sizing comboBox controls, 295-296
table-driven customization, 289-291
template creation, 299
websites for information, 300
UserForms. See UserForms
UI design
   borders, 84
   time sheet example application, 104
   cell comments, 86-87
   time sheet example application, 104
   conditional formatting, 92-94, 96-98
   dynamic tables, creating, 93-94, 96
   error conditions, highlighting, 96-98
   time sheet example application, 105
   controls, 98-100
   data validation, 88-92
   cascading lists for, 90-92
   time sheet example application, 104
   unique entries, enforcing, 89
   defined names, 71-78
   named constants, 72-73
   named formulas, 76-77
   named ranges, 73-75
   scope of, 77-78
   time sheet example application, 102-103
   dynamically modifying, 124-125
   principles of, 69-70
   program rows/columns, 70-71
   time sheet example application, 102
   settings management, table-driven approach to, 115-124
   shapes, 87-88
   styles, 78, 80-83
   adding drop-down to toolbar, 82-83
   creating custom, 79-81
   modifying, 82
   time sheet example application, 103
   table formatting, 85-86
   UIS (user interface support) layer, 377
   opening and initializing, 128-130
   saving, 131-133
   user name setting, changing, 87
   user selections. See selections
   user settings, storing and restoring, 143-148
   unhandled errors, 465
   UnHighlight method (Cell object), 174
   Unicode versus ANSI in API calls, 342-343
   unique entries, enforcing with data validation, 89
   unique indexes, 594
   unloading
      COM add-ins, 989-990
      XLAs, 987-989
   unmanaged code, 820
   enabling debugging, 846
   unmanaged COM add-ins, 889.
      See also COM add-ins
   unregistering
      COM add-ins manually, 940
      COM shim DLL files, 960
   unstructured ranges, 662
   UPDATE clause (SQL UPDATE statement), 597
   UPDATE statement (SQL), 597
      for Access databases, 626-629
   updates
      distributing, 1104-1105
      for VSTO workbooks, 1023-1024
   updating
      charts automatically, 692-694
      code comments, 49
      upsizing Access databases to SQL Server, 642-646
      Upsizing Wizard (Access), 642, 644-646
   User Account Control (UAC), 326-328
   user controls, 999
   user ID, finding, 355-356
   user interface. See UI
   user interface design. See UI design
   User Interface Editor, 908
   user interface support (UIS) layer, 377
   user interface workbooks
      (time sheet example application)
      adding rows to, 133-134
      clearing data entry cells, 134
      locating, 137
   unstructured ranges, 662
   unhandled errors, 465
   UnHighlight method (Cell object), 174
   Unicode versus ANSI in API calls, 342-343
   unique entries, enforcing with data validation, 89
   unique indexes, 594
   unloading
      COM add-ins, 989-990
      XlAs, 987-989
   unmanaged code, 820
   enabling debugging, 846
   unmanaged COM add-ins, 889.
      See also COM add-ins
   unregistering
      COM add-ins manually, 940
      COM shim DLL files, 960
   unstructured ranges, 662
   UPDATE clause (SQL UPDATE statement), 597
   UPDATE statement (SQL), 597
      for Access databases, 626-629
   updates
      distributing, 1104-1105
      for VSTO workbooks, 1023-1024
   updating
      charts automatically, 692-694
      code comments, 49
      upsizing Access databases to SQL Server, 642-646
      Upsizing Wizard (Access), 642, 644-646
   User Account Control (UAC), 326-328
   user controls, 999
   user ID, finding, 355-356
   user interface. See UI
   user interface design. See UI design
   User Interface Editor, 908
   user interface support (UIS) layer, 377
   user interface workbooks
      (time sheet example application)
      adding rows to, 133-134
      clearing data entry cells, 134
      locating, 137
forms (VB6) versus, 759-762, 764-769
ActiveX control support, 760
control arrays, 761-769
graphics, displaying, 397-398
hiding, 381
modal, 419
modeless, 420
combining with menu items, 423-425, 460-461
as progress bars, 421-422, 431
as splash screens, 420-421
naming conventions, 36
pop-up menus in, 399-400
positioning next to cells, 400-402
resizing, 403-404
screen resolutions, adapting to, 402-403
splitter bars, simulating, 405-406
window styles, modifying, 392-396
wizard dialogs, 407
creating, 409, 411
design best practices, 407-408
combining with menu items, 423-425, 460-461
as progress bars, 421-422, 431
as splash screens, 420-421
naming conventions, 36
pop-up menus in, 399-400
positioning next to cells, 400-402
resizing, 403-404
canceling program execution, 484-485, 491, 494-495
defined, 2
Users folder (Windows Vista), 328
Using keyword, 945
utility add-ins. See general add-ins
utility modules, defined, 121

V
Validating event, 877
validation, 16-18, 117-118
of arguments, 63
data validation lists, 590
variables
auto-completing names of, 31
avoiding reusing, 54
best practices, 54-61
declaring
with conditional compilation constants, 511
including object libraries in, 711-712
VB.NET, 834-836
evaluating in Immediate window, 519
initializing (VB.NET), 834-836
interfaces and, 434
member variables, defined, 33
naming conventions
example, 33
object variables
declaring and initializing, 55
early binding versus late binding, 59-61
fully qualified names, 56
performance advantages of, 571
passing as Double data type, 573
RHS (Right Hand Side), 445
scope, 58-59
watching
in arrays, UDTs, classes, 529-531
editing watches, 525-529
setting watches, 522, 524
variant arrays, performance and, 572-573
Variant data type, 54-55
Variant versions (string-handling functions), 569
VB.NET (Visual Basic .NET), 817-820
arrays, 839-845
backward compatibility, 820
ByVal or ByRef argument passing, 838
classes, 940-947
adding to solutions, 941
creating well-designed, 941-945
properties, 946-947
COM communications and, 817. See also automation, .NET Framework and data types, 838-839
exception handling, 829-833
exporting data with ADO, 948-952
managed automation add-ins creating, 929-933
limitations of, 933-940
managed COM add-ins building user interface, 908-927
creating, 891, 893-908
object instances, creating, 836-837
Q&A forums, 871
resources for information, 870-871
scope, 833-834
Shared Add-in template, 889
time sheet example application, 872-874, 876-887
transferring VB6 applications to, 820
variables, declaring/initializing, 834-836
VB6 versus, 817
versions of, 820
Visual Studio IDE (VS IDE), 821-823
Web Services, creating, 1066-1068
wizards, 838
VB.NET solutions, 823
creating, 824-829
debugging, 845-846, 848-853
Breakpoints window, 849-850
Call Stack window, 850
Command window, 848
conditional compilation constants, 851-852
Error List window, 848
Exception Assistant, 846-847
Exceptions dialog, 850-851
Immediate window, 849
keyboard shortcuts, setting, 845
Object Browser, 847-848
Output window, 849
unmanaged code, enabling debugging, 846
Watch/Quick Watch windows, 850
resources in, 863-864
time sheet example application, 573-874, 876
VB6 (Visual Basic 6), 5, 741
ActiveX DLLs, 742
advantages of using, 758-774
COM add-ins. See COM add-ins
compiling, 744, 750
form display example, 751-758
in-process communication, 774
loading icons with resource file, 802-807
one-way communication example, 744-747
projects, creating, 742-744
referencing, 745-746
registering, 744
setting references, 747
two-way communication example, 747-751
collections, default properties and member processing, 175-176, 711-772
COM add-ins, 783-787
Add-in Designer, 788-790
AddinInstance object events, 790-794
advantages of using, 798-799
automation add-ins, 799-802
checking for installation, 788
command bar architecture, 795-796
command bar event hooks, 795
converting Excel add-ins to, 797
custom toolbar faces, 796
enabling/disabling, 757
Hello World example, 783-787
installing for multiple users, 791-792
as multi-application, 798
registering, 790
security, 798
separate threading, 798-799
EXE applications, 775-783
front loaders, 782-783, 808-815
out-of-process communication, 774
Roman numeral conversion example, 775-780, 782
forms. See forms (VB6)
obtaining, 741
transferring applications to VB.NET, 820
VB.NET versus, 817
VBA versus, 5
VB6 Resource Editor, 802
VBA (Visual Basic for Applications). 5
OOP (object-oriented programming) and, 5
uses of, 5-6
VB6 versus, 5
in VSTO workbooks, 1019
VBA developers, defined, 3
VBA programming best practices, 52-65
defensive coding, 62-65
module directives, 52-53
variables and constants, 54-61
VBE Tools Control Nudger toolbar, 385
vbObjectError constant, 474
VeriSign, 1097
version checks, 142-143
version control, 65-67
Version property (Application object), 142
Version property (auto-generated references), 897
versions. See also cross-version applications
of applications (Add-in Designer), 789
of Excel, support for, 9-10
of projects, saving, 66
of .NET Framework, 818-819
of VB.NET, 820
vertical partitioning, 588
viewing value of constants, 58
Vista
cross-version applications. See cross-version applications
Public profile, 329
standard user accounts, 328-329
User Account Control (UAC), 326-328
Visual Basic 2008 Programmer’s Reference (Stephens), 871
Visual Basic 6. See VB6
Visual Basic Development Settings (in Visual Studio IDE), 821
Visual Basic for Applications. See VBA
Visual Basic Projects, naming conventions, 37
Visual Basic.NET. See VB.NET
Visual Studio, creating XLL projects, 1030-1032, 1034
Visual Studio IDE. See VS IDE
Visual Studio Tools for Office. See VSTO
Visual Studio Tools for Office (Carter and Lippert), 1026
Visual Studio.NET. See VS.NET
white space in code, 50-52
Whitechapel, Andrew, 962
Width setting (command bar definition table), 206-207
wildcard characters in criteria ranges, 677
win32api.txt file, 333
window classes, 341
window handles, 340
window styles for UserForms, modifying, 392-396
window-related API calls, 340
messages, sending, 346-348
related windows, finding, 343-346
window classes, 341
window icons, changing, 348-349
window styles, changing, 349
windows, finding, 342-343
changing icons for, 348-349
changing styles for, 349
finding with API calls, 342-343
finding related with API calls, 343-346
hiding, 823
parent windows, changing, 756
sending messages between, 346-348
Windows API, exceptions to naming conventions, 39
Windows API calls. See API calls
Windows Common Controls, 430-431
Windows Forms, 825. See also custom task panes (CTPs)
adding ActiveX controls to, 826
closing, 828
controls, 1008
displaying, 971-972
extender providers, 876-879
Load event, 826
Windows Installer, 1104
deployment model, 982, 993
Windows integrated security for SQL Server databases, 631
Windows Script Networking object library, 1095
Windows SharePoint Services, 976
Windows versions needed for .NET Framework, 819
Windows Vista,
cross-version applications. See cross-version applications
Public profile, 329
standard user accounts, 328-329
User Account Control (UAC), 326-328
Windows Workflow Foundation (WF), 976
Windows XP styles, 876
With blocks, performance advantages of, 571
WithEvents assignments, 234
WithEvents object variable, declaring, 177
wizard dialogs, 407
creating, 409, 411
design best practices, 407-408
table-driven dynamic wizards, 411-415
wizards in VB.NET, 838
WnDProc message-handling procedure, 346
Word, starting instances of, 813
Word MVP Web site, 740
Word object library, 729-732
ActiveDocument, referencing, 712
Application object, 729
Bookmark object, 729
Document object, 729
element application, 730-732
Range object, 729
referencing, 710
WordArt, displaying on UserForms, 397-398
work processes, support for, 278-279
workbook events, add-ins and, 784
workbook-level defined names, 77-78
workbooks
add-ins as, 17
identifying with custom document properties, 161-163
installation workbooks, 1103
self-automated, 15-16
template workbooks
for application-specific add-ins, 18
creating, 189-190
user interface workbooks. See user interface workbooks
visible workbooks, counting, 136
workbooks (VSTO), 1006
creating, 1009-1011
host controls, 1006, 1008
ListBox controls, 1013-1016
NamedRange controls, 1011-1012
server-side solutions, 1008-1009
Windows Forms controls, 1008
worksheet functions, XLL-based.
See XLLs
Worksheet Menu Bar, adding custom menu with sub-menus, 220-221, 223
worksheet-based user interfaces, form-based user interfaces versus, 154-156
worksheet-level defined names, 77-78
worksheets
adding to VSTO workbooks, 1010-1011
as data entry forms, 4-5
as data stores, 5
databases versus, 577-578
functions as declarative programming language, 6-7
naming conventions, 37
referencing, 65
sheet navigation in Ribbon UI, 296-298
UI design
borders, 84, 104
cell comments, 86-87, 104
conditional formatting, 92-98, 105
controls, 98-100
data validation, 88-92, 104
defined names, 71-78, 102-103
dynamically modifying, 124-125
principles of, 69-70
program rows/columns, 70-71, 102
settings management, 118-124
shapes, 87-88
styles, 78, 80-83, 103
table formatting, 85-86
Worksheets property, performance and, 573
wrapper functions in Web Services, 1071-1072
wrapping procedures in On Error Resume Next statement, 471
writing help file contents, 1091
X
x,y coordinates, converting polar coordinates to, 694, 696
XL-Dennis blog, 962
xlAddInManagerInfo function (XLLs), 1042-1043
XLAs, loading/unloading, 987-989
xlAutoAdd function (XLLs), 1044
xlAutoClose function (XLLs), 1041-1042
xlAutoFree function (XLLs), 1044
xlAutoOpen function (XLLs), 1040-1041
xlAutoRegister function (XLLs), 1043
xlAutoRemove function (XLLs), 1044
.xlb files, 147
.xcall.h file, 1030, 1045
.xcall32.lib file, 1030
xlCoerce function, 1052-1053
xlerrDiv0 constant (XLOPER error value), 1048
xlerrNA constant (XLOPER error value), 1048
xlerrName constant (XLOPER error value), 1048
xlerrNull constant (XLOPER error value), 1048
xlerrNum constant (XLOPER error value), 1048
xlerrRef constant (XLOPER error value), 1048
xlerrValue constant (XLOPER error value), 1048
xlfsn parameter (Excel4 function), 1050
xlFree function, 1052
xlGetName function, 1053
XLL+, 1062
XLLs
advantages of using, 1029
C API functions called in, 1052-1053
COM automation and, 1061
debugging, 1060-1061
defined, 1029-1030
Excel4 function, 1050-1051
IFERROR function example, 1057-1060
K data type arguments, 1039
projects, creating, 1030-1034
registering functions in, 1054-1057
resources for information, 1062-1063
structure of, 1034, 1036-1044
callback functions, 1040-1044
DllMain function, 1039-1040
function table, 1035-1039
XLOPER data type, 1044-1050
C++ keywords and, 1061
XLM functions, 698
XLM macros
hiding Ribbon UI using, 294-295
registering UDFs with, 114
xmacro.exe, 698
XLOPER data type, 1044-1050
arrays and, 1048-1049
C++ keywords and, 1061
constants defined in, 1046
error values, 1048
memory management, 1049-1054
numeric data in, 1047
string data in, 1047
xlCoerce function, 1052-1053
xlFree function, 1052
xlGetName function, 1053
xlretAbort constant (Excel4 function return value), 1051
xlretFailed constant (Excel4 function return value), 1051
xlretInvCount constant (Excel4 function return value), 1051
xlretInvCount constant (Excel4 function return value), 1051
xlretSuccess constant (Excel4 function return value), 1051
xlretUncalced constant (Excel4 function return value), 1051
.xls files, 275
.xls files, 319-320
.xType argument (xlCoerce function), 1053
.xTypeBigData constant (XLOPER data type), 1047
.xTypeBool constant (XLOPER data type), 1046
.xTypeErr constant (XLOPER data type), 1046
.xTypeFlow constant (XLOPER data type), 1047
.xTypeNat constant (XLOPER data type), 1047
.xTypeMissing constant (XLOPER data type), 1047
.xTypeMulti constant (XLOPER data type), 1047
.xTypeNil constant (XLOPER data type), 1047
.string data in,
XML data files from financial model example, 268
XML Editor, 920
XML file formats, Open XML, 1009
XML in Office Developer Portal, 300
XML Maps, 259-267
XML markup, 1067
XML parsers, 290
XML parts, customUI, 290-291

XML Schema Definition files.
See XSD

XML Source Task Pane, 255
XMLDataQuery method, 267
XMLMappedRange control, 1007
XMLMapQuery method, 267
XPaths, 267

XSD (XML Schema Definition files), 250, 290
creating for financial model example, 257-259
example XSD file, 252-254
financial model example, 263-265
importing, 255
Xtreme VB Talk, 871

Z
z-order, changing, 385
ZIP archives
components of, 275
defined, 274
zooming in time series, 694