DEVELOPING OFFICE APPLICATIONS USING VBA

Table of Contents

Microsoft® Office application development is typically the process of customizing an Office application to perform some function or service. Developing an Office application can range from writing a simple Microsoft® Visual Basic® for Applications (VBA) procedure to creating a sophisticated financial analysis and reporting application. An Office developer is anyone who uses the programmability features of Office to make an application do something better, faster, or more efficiently than it could be done before. An Office application is an application that uses an Office application or component as part of its overall architecture.

Every custom application is, in some sense, an answer to a particular problem or requirement. When you understand the problem, the success of your application will depend on your ability to deliver a response that uses appropriate tools tailored to the experience level of the people who will be using your application.

The Benefits of Office Programmability

There are currently more than 2.5 million Office developers creating custom applications that use the applications or components in Microsoft® Office. The term "Office developer" includes developers who work exclusively in one or more of the Office applications. It also includes developers working in any language that can access the objects exposed by Office applications. For more than ten years, Microsoft has been making improvements to the Office suite that make it possible for developers to quickly and easily build and deploy custom desktop applications. These improvements are the reason why Office applications continue to play such an important role in custom application development:

- **Users and businesses already use the Microsoft Office suite of applications.** Most users already have Office on their desktop. The most recent surveys indicate that more than 40 million people regularly use Office to get their work done. Building applications based on the Office platform makes it possible for developers to target this large base of users. Also, even if you are not developing within the Office development environment, it is still a good idea to take advantage of the objects exposed by Office applications so that your custom applications can leverage existing, proven, and tested Office functionality.

- **Office supports programmable objects and an integrated development environment.** Each Office application exposes its functionality through programmable objects, and each also supports the ability to integrate with other applications by using Automation (formerly OLE Automation). Most applications share the same programming language (Visual Basic for Applications) and integrated development environment (the Visual Basic Editor). Applications created with VBA run in the same memory space as the host application and therefore execute faster. The programmable objects and powerful development tools in Office let developers build applications that tightly integrate applications and seamlessly share data and information. In addition, distribution of Office-based applications is simplified because VBA code and Microsoft® ActiveX®controls are part of the application document or project.

- **Faster development cycles mean more affordable applications.** The use of a single language and development environment also makes application development faster. What you learn while programming one application applies when working with another application. VBA code written for one application can often be reused in an application that works with a different application. Developers' skills become more valuable because they can work across many applications. Reducing the number of development environments or languages that developers must learn means that the time and cost of creating custom applications is reduced. Reliance on existing components eliminates the need to develop or test large portions of the application. This lets developers quickly build robust applications that previously might have been cost-prohibitive.

- **Users become part of the application.** Because applications are created and run in an environment familiar to the user, support costs are kept to an absolute minimum. Building an application based on Office technologies makes it possible for you define the application in a context with which your users are already familiar. This makes it possible for greater user participation in the application-design process and can dramatically reduce training and support costs. It takes less time and effort to customize applications your users already own than to build new applications from scratch. The more familiar users are with the application, the easier it is going to be for them to understand and use your application.

1. OFFICE OBJECTS AND OBJECT MODELS

Each Microsoft® Office XP application contains a powerful set of tools designed to help you accomplish a related set of tasks. For example, Microsoft® Access provides powerful data-management and query capabilities, Microsoft® Excel provides mathematical, analytical, and reporting tools, Microsoft® Outlook® provides tools for sending and receiving e-mail, for scheduling, and for contact and task management, and Microsoft® Word makes it possible for you create and manage documents, track versions of documents among different users, and create forms and templates. As powerful as these and the other Office applications are on their own, you also can integrate the features from two or more Office applications into a single solution to amplify and focus users' productivity.

The key technology that makes individual Office applications programmable and makes creating an integrated Office solution possible is the Component Object Model (COM) technology known as automation.

Automation makes it possible for a developer to use Microsoft® Visual Basic® for Applications (VBA) code to create and control software objects exposed by any application, dynamic-link library (DLL), or Microsoft® ActiveX® control that supports the appropriate programmatic interfaces. VBA and automation make it possible for you to program individual Office applications, as well as to run other applications from within a host application. For example, you can run a hidden instance of Excel from within Access to perform mathematical and analytical operations on your Access data. The key to understanding automation is to understand objects and object models: what they are, how they work, and how they work together.

Note
To master the use of automation in your Office solutions, you must have a detailed working knowledge of the applications you are integrating. That is the kind of knowledge and experience that can be gained only with further application-specific training and hands-on experience.

In This Section

Integrated Office Solution Development
Use the COM software architecture or VBA to develop an integrated Office solution.

Objects, Collections, and Object Models: Technology Backgrounder
Understand how to reference objects in an application's object model and how to use the objects and features available to build your solution.

Office Application Automation
Learn how to automate one Office application from another.

Integrated Office Solution Development

The ability to develop an integrated Office solution heavily depends on two technologies:

- The Component Object Model (COM) software architecture
- Microsoft® Visual Basic® for Applications (VBA)

The COM software architecture makes it possible for software developers to build their applications and services from individual software components collectively referred to as COM components or simply components. COM components consist of the physical, compiled files that contain classes, which are code modules that define programmable objects. There are two types of COM components: in-process components and out-of-process components. In-process components are either DLLs or Microsoft® ActiveX®controls (.ocx files) and can run only within the process of another application. Out-of-process components are .exe files and run as freestanding applications. A COM component can serve either or both of the following roles in application development:

- Sharing its objects with other applications. This role is called being an Automation server.
- Using other components' objects. This role is called being an Automation client. In earlier documentation, this role was called being an Automation controller.

The Microsoft® Windows® operating system and Microsoft® Office XP suite of applications are examples of products that have been developed by using the COM software architecture. Just because software is developed by using COM does not mean that it can be programmed by using VBA. However, if an application or service supports automation, it can expose interfaces to the features of its components as objects that can be programmed from VBA, as well as many other programming languages. To support automation, an application or service must provide either or both of two methods of exposing its custom interfaces:

- By providing the IDispatch interface. In this way, the application or service can be queried for further information about its custom interfaces. Applications and services that support the IDispatch interface provide information about their custom interfaces at run time by using a method called late binding.
- By making it possible for direct access at design time to the member functions in its virtual function table, or vtable, that implement its interfaces. Applications and services that support direct access to custom interfaces support what is called vtable binding or early binding.

An application can be said to support automation if it supports either one, but not necessarily both, of these methods. Most contemporary applications and services provide support for both methods and are referred to as supporting dual interfaces.

To support early or late binding, an application or service also must supply a type library (also known as an object library). A type library is a file or part of a file that describes the type of one or more objects. Type libraries do not store objects; they store type information. By accessing a type library, a programming environment can determine the characteristics of an object, such as the interfaces supported by the object and the names and addresses of the members of each interface. With this information, the programming language can be used to work with the exposed interfaces.

In the VBA programming environment, you can establish a connection to a type library, which is called establishing a reference to a type library. After you establish a reference to a type library, you can view information about the objects made available through the type library by using the Object Browser. Establishing a reference to a type library also makes it possible for VBA to perform error-checking at compile time to ensure code written against the type library is free from errors because of improper declarations or from passing values of the wrong type. Additionally, referencing a type library makes it possible for you to take advantage of VBA features that simplify writing code, such as automatic listing of the properties and methods of objects exposed by the type library. Furthermore, referencing a type library makes your code run faster, because information about the objects you are programming is available to VBA at design time; this information can be used to optimize your code when it is compiled.

The VBA programming environment can be incorporated into applications that support automation to make them programmable. The suite of Microsoft® Office XP applications, incorporate the VBA programming environment and are written to support both kinds of automation interfaces. Additionally, many other software components, such as Microsoft® ActiveX® controls and DLLs, expose their functionality to VBA programmers through automation interfaces.

Using the objects, properties, and methods exposed through automation interfaces, you can use VBA code running in modules associated with the currently open document, template, database, Microsoft® FrontPage®-based web, or add-in to automate that application. VBA and automation make it possible to record simple macros to automate keystrokes and mouse actions (in applications that support macro recording), and to create sophisticated integrated solutions, such as document management, accounting, and database applications.

To produce even more powerful integrated applications, you can use VBA code running in one application to create and work with objects from another installed application or component. For example, if you are developing a solution in Microsoft® Access and you want to use mathematical or other functions available only in Microsoft® Excel, you can use VBA to create an instance of Excel and use its features from code running in Access.

You can think of automation as a nervous system that makes programmatic communication and feedback between applications and components possible, and as "glue" that makes it possible for you integrate features from Office applications and other software components into a custom solution.

The VBA support for automation provides Office developers with incredible flexibility and power. By taking advantage of automation, you can use the features exposed through the object models of the entire Office suite of applications (as well as any third-party applications and components that support automation interfaces) as a set of business-application building blocks. By taking advantage of the pre-built components exposed through automation, you do not have to develop your own custom components and procedures every time you want to get something done. In addition to shortening the development time for your solution, using pre-built components means you can take advantage of the thousands of hours of design, development, and testing that went into producing them.

By using VBA and objects exposed through automation, you can select the best set of features to use to perform the tasks you want to accomplish, you can provide the data users must have to accomplish their jobs, and you can manage workflow to provide an effective and productive solution.

Objects, Collections, and Object Models: Technology Backgrounder

Microsoft® Office XP applications expose their functionality to the Microsoft® Visual Basic® for Applications (VBA) language through a hierarchical system of objects and collections of objects called an object model. When you understand how to reference objects in an application's object model, you can use the objects and features available to build your solution.

An application fundamentally consists of two things: content and functionality. Content refers to the information within an application, that is, the documents, worksheets, tables, or slides and the information they contain. Content also refers to information about the attributes of individual elements in that application, such as the size of a window, the color of a graphic, or the font size of a word. Functionality refers to all the ways you can work with the content in the application, for example, opening, closing, adding, deleting, sending, copying, pasting, editing, or formatting the content in the application.

The content and functionality that make up an application are represented to the Visual Basic language as discrete units called objects. For the most part, the set of objects exposed by an application to VBA corresponds to all the objects that you can work with by using the application's user interface. You probably are familiar with many of these objects, such as Microsoft® Access databases, tables, queries, forms, and reports; Microsoft® Excel workbooks, worksheets, and cell ranges; Word documents, sections, paragraphs, sentences, and words; Microsoft® Outlook® messages, appointments, and contacts; Microsoft® PowerPoint®presentations and slides; and Microsoft® FrontPage®-based webs and web pages.

The objects exposed by an application are arranged relative to each other in hierarchical relationships. The top-level object in a Microsoft® Office XP application is the Application object, which represents the application itself. The Application object contains other objects that you have access to only when the Application object exists (that is, when an instance of the application itself is running). For example, the Excel Application object contains Workbook objects, and the Word Application

object contains Document objects. Because the Document object depends on the existence of the Word Application object for its own existence, the Document object is said to be the child of the Application object; conversely, the Application object is said to be the parent of the Document object.

Many child objects have children of their own. For example, the Excel Workbook object contains, or is parent to, the Worksheets object. The Worksheets object is a special kind of object called a collection that represents a set of objects - in this case, all the worksheets in the workbook, which in turn are represented as individual Worksheet objects within that collection. A parent object can have multiple children; for instance, the Word Window object has as children the Document, Panes, Selection, and View objects. Additionally, identically named child objects might belong to more than one parent object; for instance, in Word, both the Application object and the Document object have a Windows collection as a child object. However, even though the child objects have the same name, typically, their functionality is determined by the parent object; for example, the Microsoft® Windows® collection for the Application object contains all the current document windows in the application, whereas the Windows collection for the Document object contains only the windows that display the specified document.

In addition to containing child objects, each object in the hierarchy contains content and functionality that apply both to the object itself and to all its child objects. The higher an object is in a hierarchy of nested objects (that is, the more child objects an object has), the wider the scope of its content and functionality. For example, in Excel, the Application object contains the size of the application window and the ability to quit the application; the Workbook object contains the file name and format of the workbook and the ability to save the workbook; the Worksheets collection contains Worksheet object names and the ability to add and delete worksheets.

You often do not get to the actual contents of a file, such as the values on an Excel worksheet or the text in a Word document, until you have navigated through several levels in the object hierarchy. This is because the scope of this specific content belongs to a particular functionality of the application. For example, the value in a cell on a worksheet applies only to that cell, not to all cells on the worksheet, so you cannot store the value directly in a Worksheet object.

To work with the content and functionality exposed by an object, you use properties and methods of that object. You use properties to determine or change some characteristic of an object, such as its color, dimensions, or state. For example, you can set the Visible property of an Excel Worksheet object to specify whether a worksheet is visible to the user. You use methods to perform a particular action on an object. For example, you use the PrintOut method of the Word Document object to print the document.

Some objects also respond to events. An event is an action that typically is performed by a user such as clicking a mouse, pressing a key, changing data, or opening a document or form but also can be performed by program code or by the system itself. You can write code, called an event procedure, which will run whenever an event occurs. For example, you can write code in a form's Open event to size or position the form whenever it is opened.

In summary, the representation of content and functionality in an application is divided among the objects in the application's object model. Together, the objects in the object model's hierarchy represent all the content and functionality in the application that is exposed to Visual Basic. Separately, the objects provide access to very specific areas of content and functionality. To determine or set a characteristic of an object, you read or set one of the object's properties. To perform an action on or with an object, you use one of the object's methods. Additionally, some objects provide events that are typically triggered by a user's action, so you can write code that will run in response to that action.

Objects Exposed by an Object Model

To work with the objects exposed by an object model, you first must declare an object variable and set a reference to the object you want to work with. When you have established a reference to an object, you can work with its properties, methods, or events.

To set a reference, you must build an expression that gains access to one object in the object model and then use properties or methods to move up or down the object hierarchy until you get to the object you want to work with. The properties and methods you use to return the object you start from and to move from one object to another are called "object accessors" or just "accessors."

Accessors typically have the same name as the object they are used to access; for example, the Word Documents property is used to access the Documents collection. Accessors are typically properties, but in some object models, accessors are methods.

This topic includes the following sections:

- The Application Object
- Navigating the Object Hierarchy
- Shortcut Accessors
- Referencing the Document or Workbook in Which Code Is Running
- The Parent Property
- Accessing an Embedded OLE Object's Application
- Creating Your Own Objects and Object Models

The Application Object

A common place to gain access to the object model is the top-level object. In all Microsoft® Office XP applications and in most applications that support Microsoft® Visual Basic® for Applications (VBA), the top-level object is the Application

object. However, some applications and components might have a different top-level object. For example, when you are programming the Visual Basic Editor (by using a reference to the Visual Basic for Applications Extensibility 5.3 library), the top-level object is the VBE object.

Note
The following code assumes you are running Microsoft® Word.

You use the Application property to return a reference to the Application object. The following code fragment returns a reference to the Application object and then sets properties to display scroll bars, ScreenTips, and the status bar:

```
Dim wdApp As Application
Set wdApp = Application
With wdApp
  .DisplayScrollBars = True
  .DisplayScreenTips = True
  .DisplayStatusBar = True
End With
```

If you have established references to more than one type library that contains an Application object, the Application property will always return the Application object for the host application. In addition, for any other object that has the same name in two or more referenced type libraries, the accessor property or method will return the object from the first type library referenced in the Available References list of the References dialog box (Tools menu).

For example, the Microsoft® ActiveX® Data Objects (ADO) and Data Access Objects (DAO) type libraries both have Recordset objects. If you have a reference to the ADO type library followed by the DAO type library, a declaration such as the following will always return the ADO Recordset object:

```
Dim rstNew As Recordset
```

While you might be able to adjust the priority of references in the References dialog box to correct this, a better solution, which eliminates any ambiguity and prevents errors, is to declare an object variable by using the fully qualified class name, also called the programmatic identifier or ProgID, of the object. To do this, combine the name of the application or component that contains the object (as it appears in the Object Browser's Project/Library box) with the name of the object separated by a period.

For example, to declare an object variable that will be used to work with the Word Application object from another application, you must declare the object variable this way:

```
Dim wdApp As Word.Application
```

Similarly, if you have both the ADO and the DAO type libraries referenced in your project, you should declare object variables to work with Recordset objects this way:

```
Dim rstADO As ADODB.Recordset
Dim rstDAO As DAO.Recordset
```

Note
You can view the ProgIDs of all installed applications and components on a computer by running the Registry Editor and looking under the \HKEY_CLASSES_ROOT\CLSID subkey.

Navigating the Object Hierarchy

To get to an object from the top-level object, you must step through all the objects above it in the hierarchy by using accessors to return one object from another. Many objects, such as workbooks, worksheets, documents, presentations, and slides, are members of collections. A collection is an object that contains a set of related objects. You can work with the objects in a collection as a single group rather than as separate entities. Because collections are always one level higher than individual objects in the hierarchy, you usually have to access a collection before you can access an object in that collection. The accessor that returns a collection object usually has the same name as the collection object itself. For example, the Documents property of the Word Application object returns the Documents collection object, which represents all open documents. The following expression returns a reference to the Word Documents collection object:

```
Application.Documents
```

You reference an item in a collection either by using a number that refers to its position in the collection or by using its name. For example, if a document named Report.doc is the first open document in the Documents collection, you can reference it in either of the following ways:

```
Application.Documents(1)
-or-
Application.Documents("Report.doc")
```

To get to an object further down the object hierarchy, simply add additional accessors and objects to your expression until you get to the desired object. For example, the following expression returns a reference to the second paragraph in the Paragraphs collection of the first open document:

```
Application.Documents(1).Paragraphs(2)
```

Shortcut Accessors

There are shortcut accessors you can use to gain direct access to objects in the model without having to navigate from the Application object. These shortcuts include accessors, such as the Documents, Workbooks, Items, and Presentations properties, that you can use to return a reference to the document collection for the corresponding application. For example, in Word, you can use either of the following statements to open MyDoc.doc:

Application.Documents.Open Filename:="c:\docs\mydoc.doc"
-or-
Documents.Open Filename:="c:\docs\mydoc.doc"

There are other shortcut accessors, such as the ActiveWindow, ActiveDocument, ActiveWorksheet, or ActiveCell properties that return a direct reference to an active part of an application. The following statement closes the active Word document. Note that the Application object and the Documents collection object are not explicitly specified.

ActiveDocument.Close

Tip
When <globals> is selected in the Classes list in the Object Browser, you can use any accessor that appears in the Members of list as a shortcut. That is, you do not have to return the object that the property or method applies to before you use the property or method, because VBA can determine that information from the context in which your code is running.

Referencing the Document or Workbook in Which Code Is Running

When you are using the ActiveDocument and ActiveWorkbook accessor properties, it is important to remember that the reference returned is to the document or workbook that is currently in use (the topmost window of all open documents or workbooks). In many circumstances, you can reference an active object implicitly, that is, without including the entire hierarchy above the object to which you are referring. For example, you can create a reference to the active workbook's Worksheets collection without preceding the collection with ActiveWorkbook. or an explicit reference to the workbook's name or number in the Workbooks collection:

Worksheets("MySheet")

However, using implicit references or references to the ActiveDocument or ActiveWorkbook accessor properties can create problems if you are developing a global template or add-in and need to make sure your code refers to the add-in or global template itself. Word and Excel provide two special accessor properties that return a reference to the document or workbook in which the VBA code is running: ThisDocument and ThisWorkbook. Use the ThisDocument or ThisWorkbook property whenever you need to make sure that your code refers to the document or workbook that contains the code that is running.

For example, both of the following Set statements reference the worksheet named Addin Definition. The first makes an explicit reference to the active workbook by using the ActiveWorkbook property. The second makes an implicit reference; because it doesn't explicitly refer to a specific workbook, the reference is assumed to be to the active workbook. In either case, the reference made in the Set statement will be to the worksheet in whatever workbook happens to be active when the code runs.

Set rngMenuDef = ActiveWorkbook.Worksheets("Addin Definition"). Range("MenuDefinition")
Set rngMenuDef = Worksheets("Addin Definition"). Range("MenuDefinition")

References such as these will work correctly while you are developing an add-in or template if you have no other documents or workbooks open while you are testing your code, or if the add-in or template is in the active window when the code is running. However, when your add-in or template is in use, these types of references can cause errors. To make sure that you are referencing the workbook in which code is running, use the ThisWorkbook property as shown in the following Set statement:

Set rngMenuDef = ThisWorkbook.Worksheets("Addin Definition"), Range("MenuDefinition")

The Parent Property

To access an object higher up in the object hierarchy from the current object, you can often use the Parent property of the object. Using an object's Parent property makes it possible for you to reference the higher object that contains the current object. For example, if you write a function to work with a control on a form (the function takes an argument of type Control), you can use the control's Parent property to reference the form that contains the control.

Note that the Parent property doesn't always return the object immediately above the current object in the hierarchy, it might return a higher object, especially if the object immediately above the current object is a collection. For example, the Parent property of a Word Document object returns the Application object, not the Documents collection. You can use the TypeName function to find out what kind of object to which the Parent property of an object refers. For example, in Word, the following statement displays the type of object that the Parent property of the Document object refers to:

MsgBox TypeName(Documents("Document1").Parent)

Tip
You can use the TypeName function to determine the type of object returned by any expression, not just expressions that use the Parent property. The TypeName function can also be used to determine the kind of data type returned by an expression, such as Byte, Integer, or Long.

Creating Your Own Objects and Object Models

You can create your own objects and object models by creating and using class modules. For example, you might need to work with complex sets of data that need to be managed in a consistent and reliable way. By creating your own objects, properties, and methods to work with this data in a class module, you can create an object model to make working with your data simpler and less error-prone. Similarly, you can create class modules to create wrapper functions around Windows application programming interface (API) calls or even complex parts of existing object models to make them easier to use.

Collections

Although collections and the objects they contain, such as the Workbooks collection and the Workbook object are distinct objects each with their own properties and methods, they're grouped as one unit in most object model graphics to reduce complexity.

To return a single member of a collection, you usually use the Item property or method and pass the name or index number of the member as the index argument. For example, in Excel, the following expression returns a reference to an open workbook by passing its name "Sales.xls" to the Item property and then invokes the Close method to close it:

Workbooks.Item("Sales.xls").Close

The Item property or method is the default for most collections, so you can usually omit it from your expression. For example, in Excel, the following two expressions are equivalent:

Workbooks.Item("Sales.xls")
-or-
Workbooks("Sales.xls")

To reference items in a collection by using an index number, simply pass the number of the item to the Item property or method of the collection. For example, if Sales.xls is the second workbook in the Workbooks collection, the following expression will return a reference to it:

Workbooks(2)

Note
Most collections used in Office applications (except Access) are one-based, that is, the index number of the first item in the collection is 1. However, the collections in Access and some components, such as ADO and DAO, are zero-based, which is, the index number of the first item is 0. For more information, refer to the Visual Basic Reference Help topic for the collection you want to work with.

Adding Objects to a Collection

You can also create new objects and add them to a collection, usually by using the Add method of that collection. The following code fragment creates a new document by using the Professional Memo.dot template and assigns it to the object variable docNew:

```
Const TEMPLATE_PATH   As String = "c:\program files\microsoft office\templates\1033\"
Dim docNew            As Word.Document

Set docNew = Documents.Add(Template:=TEMPLATE_PATH & "memos\professional memo.dot")
```

Working with Objects in a Collection

You can find out how many objects there are in a collection by using the Count property. The following Excel example displays a message box with the number of workbooks that are open:

```
MsgBox Workbooks.Count & " workbooks are open."
```

You can perform an operation on all the objects in a collection, or you can set or test a value for all the objects in a collection. To do this, you use a For Each...Next structure, or a For...Next structure in conjunction with the Count property to loop through all the objects in the collection.

Whenever possible, you should use a For Each...Next loop when you need to work with all the items in a collection. A For Each...Next loop generally performs faster and doesn't require you to use or test a loop counter, which can introduce errors. The following Excel example contains a For Each...Next structure that loops through the Worksheets collection of a workbook and appends " - By Automation" to the name of each worksheet:

```
Sub CreateExcelObjects()
  Dim xlApp          As Excel.Application
  Dim wkbNewBook     As Excel.Workbook
  Dim wksSheet       As Excel.Worksheet
  Dim strBookName    As String

  ' Create new hidden instance of Excel.
  Set xlApp = New Excel.Application
```

```
  ' Add new workbook to Workbooks collection.
  Set wkbNewBook = xlApp.Workbooks.Add
  ' Specify path to save workbook.
  strBookName = "c:\my documents\xlautomation.xls"
  ' Loop through each worksheet and append " - By Automation" to the name of each sheet. Close and save workbook to specified path.
  With wkbNewBook
    For Each wksSheet In .Worksheets
      wksSheet.Name = wksSheet.Name & " - By Automation"
    Next wksSheet
    .Close SaveChanges:=True, FileName:=strBookName
  End With
  Set wkbNewBook = Nothing
  Set xlApp = Nothing
End Sub
```

Under some circumstances, you must use a For...Next loop to work with items in a collection. For example, if you try to use a For Each...Next loop to delete all the objects in a collection, only every other object in the collection will be deleted. This is because after deleting the first item, all items in the collection are re-indexed so that what was the second item is now the first. When the Next statement runs at the end of the first execution of the loop, the pointer is advanced one, skipping that item for the next iteration of the loop. For this reason, to delete all items in a collection, you must use a For...Next loop that starts from the end of the collection and works backwards.

Another situation that requires you to use a For...Next loop to work with items in a collection is if you need to work with only a specific number of items, say the first ten, or every tenth item.

Properties and Methods

To work with the content and functionality exposed by an object, you use properties and methods of that object. The following Excel example uses the Value property of the Range object to set the contents of cell B3 on the Sales worksheet in the Current.xls workbook to 3:

```
Workbooks("Current.xls").Worksheets("Sales").Range("B3").Value = 3
```

The following example uses the Bold property of the Font object to apply bold formatting to cell B3 on the Sales worksheet:

```
Workbooks("Current.xls").Worksheets("Sales").Range("B3").Font.Bold = True
```

The following Word example uses the Close method of the Document object to close the file named Draft3.doc:

```
Documents("Draft3.doc").Close
```

In general, you use properties to set or read the content, which can include the text or value contained in an object, or other attributes of the object, and you use methods to work with an application's (or the Microsoft®Visual Basic®for Applications) built-in functionality to perform operations on the content. Be aware, however, that this distinction doesn't always hold true; there are a number of properties and methods in every object model that are exceptions to this rule.

Events

An event is an action that is typically performed by a user, such as clicking a mouse button, pressing a key, changing data, or opening a document or form, but the action can also be performed by program code, or by the system itself. You can write event procedure code to respond to such actions at either of two levels:

- **Document-level or subdocument-level events** These events occur for open documents and in some cases, for objects within them. For example, the Word Document object can respond to the Open, New, and Close events; the Excel Workbook object can respond to events such as the Open, BeforeClose, and BeforeSave events; and the Excel Worksheet object can respond to events, such as the Activate and Calculate events. Microsoft® PowerPoint® supports only application-level events.
- **Application-level events** These events occur at the level of the application itself, for example, when a new Microsoft® Word document, Microsoft® Excel workbook, or PowerPoint presentation is created, for which the corresponding events are the NewDocument, NewWorkbook, and NewPresentation events.

Microsoft® Access provides a different model that responds to events on Form and Report objects, and most of the controls on them, such as ListBox and TextBox objects. UserForms, which can be used from Excel, Word, and PowerPoint, provide a similar event model to Access forms.

The Microsoft® Outlook® Application object provides events that can be used from the ThisOutlookSession module or a COM add-in running from an installation of the Outlook application, such as ItemSend, NewMail, OptionsPagesAdd, Quit, Reminder, and Startup. To create code that responds to a user's actions in the Outlook user interface, you can use the WithEvents keyword to declare object variables that can respond to Outlook Explorer, Inspector, and MAPIFolder object events. All Outlook item objects, except the NoteItem object, can respond to events, such as the Open, Read, and Reply events.

The Microsoft® FrontPage® Application object provides events that make it possible for your solution to respond to the creation and editing of pages and FrontPage-based webs, such as OnPageNew, OnPageOpen, OnBeforePageSave, OnAfterPageSave, and OnPageClose, and OnWebNew, OnWebOpen, OnBeforeWebPublish, OnAfterWebPublish, and OnWebClose.

In addition to the events supported by each Office application, the CommandBarButton object, CommandBarComboBox object, and CommandBars collection support events.

Responding to Document-Level Events

To create event procedures for events in Excel workbooks and Word documents, you need to work with the ThisWorkbook or ThisDocument modules. For example, to write an event procedure that will run when a Word document is opened, open the document and then open the Visual Basic Editor. In the Project Explorer, double-click ThisDocument to open the ThisDocument module. In the Object box in the Code window, click Document, and then click Open in the Procedure box. The Microsoft® Visual Basic® Editor will create an event procedure template for the document's Open event. You can then enter any code you want to run whenever the document is opened. For example, the following event procedure sets certain features of the active window and view of a Word document when it is opened:

```
Private Sub Document_Open()
' Set Window and View properties to display document with document map  in page layout view.
  With ActiveWindow
    .DisplayVerticalScrollBar = True
    .DisplayRulers = False
    .DisplayScreenTips = True
    .DocumentMap = True
    .DocumentMapPercentWidth = 25
    With .View
      .Type = wdPageView
      .WrapToWindow = True
      .EnlargeFontsLessThan = 11
      .ShowAll = False
      .ShowPicturePlaceHolders = False
      .ShowFieldCodes = False
      .ShowBookmarks = False
    End With
  End With
End Sub
```

If you want to prevent code written in a document's Open event from running when the document is opened programmatically from another application, you can check the Application object's UserControl property to determine if a user opened the application.

Responding to Application-Level Events

Microsoft® Office XP includes a comparable set of events for Word and PowerPoint with similar names across each application. For example, where Excel provides NewWorkbook and WorkbookOpen events, Word provides NewDocument and DocumentOpen events, and PowerPoint provides NewPresentation and PresentationOpen events. Providing consistent event handling and similar names across Word, Excel, and PowerPoint makes it easier to create a COM add-in that works across these applications. FrontPage doesn't supply as extensive a set of application-level events as the other Office applications, but FrontPage events also have similar names; for example, OnPageNew, OnWebNew, OnPageOpen, and OnWebOpen.

The NewDocument, NewWorkbook, NewPresentation, and OnPageNew events are useful for tasks such as automatically formatting new documents and inserting content such as the date, time, author, or latest company logo off the intranet. Similarly, the OnWebNew event can be used to automatically apply themes or to add pages and content to new FrontPage-based webs. The DocumentOpen, WorkbookOpen, PresentationOpen, and OnPageOpen events can be used to retrieve information from the document and update command bar customizations. The DocumentClose, DocumentSave, and DocumentPrint events in Word (and comparable events in Excel and PowerPoint) can be used to ensure that document properties, such as the author or subject, are entered in the document before the document can be closed, saved, or printed. Similarly, the FrontPage OnBeforePageSave, OnBeforeWebPublish, OnPageClose, and OnWebClose events can be used to check page properties or to check the sizes of image files on the page, and to verify hyperlinks before publishing a FrontPage-based web.

To write event procedures for the Application object, you must create a new class module and declare an object variable as type Application by using the WithEvents keyword. For example, you can create a class module named XLEvents and add the following declaration to create a private Excel Application object variable to respond to events:

```
Private WithEvents xlApp As Excel.Application
```

When you have done this, you can click xlApp in the Object box of the class module's Code window, and then click any of the events in the Procedure box to write event procedures to respond to Excel Application object events. However, because you

can't use the New keyword to create an instance of the Application object variable when you are declaring it by using the WithEvents keyword, you'll need to write a Set statement to do so in the class module's Initialize event this way:

```
Private Sub Class_Initialize
  Set xlApp = Excel.Application
End Sub
```

This process is called creating an event sink. To activate the event sink, you declare in another module a public (or private) object variable for your event sink class, and then run a procedure that will create an instance of your class before the events you want to handle occur. For example:

```
Public evtEvents As XLEvents
Public Sub InitXLEvents()
  Set evtEvents = New XLEvents
End Sub
```

Creating an event sink in a class module provides a way for you to create an independent object that will respond to application-level events. The VBA project that contains the class module and procedure used to initialize your event sink must be running before any of the events you want to trap occur. Because application-level events are triggered by events that occur while the application itself is being used to open and work with documents, you will most typically implement an event sink in an add-in to trap an application's application-level events, or in automation code running from another application.

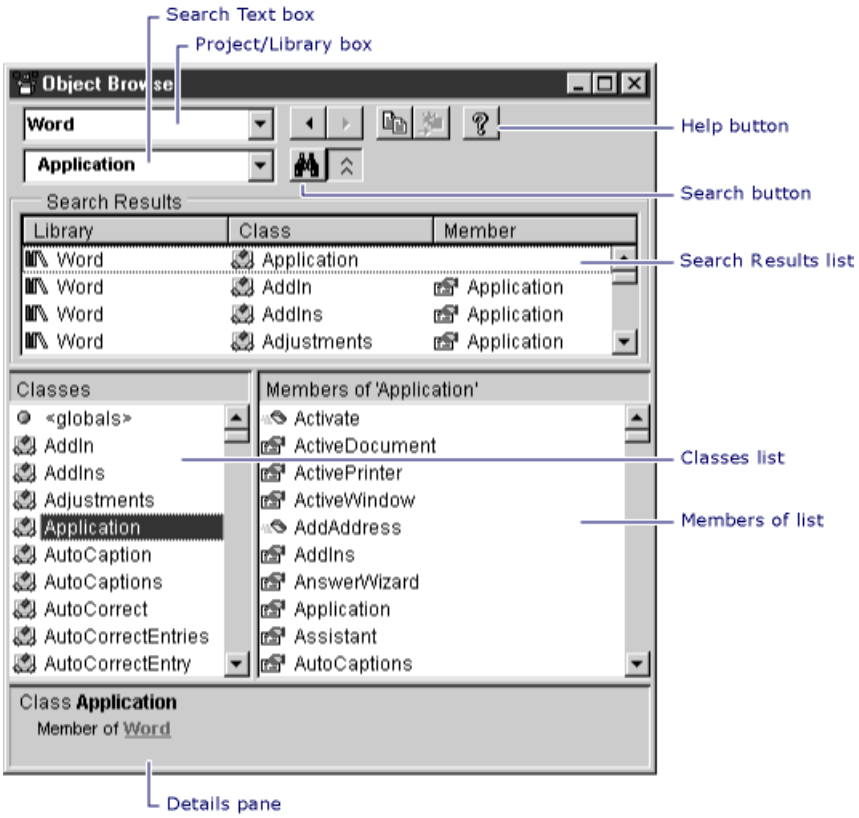
Using the Object Browser

The Object Browser is available in all Microsoft products that contain the Microsoft® Visual Basic® for Applications (VBA) programming environment. The Object Browser makes it possible for you to view all objects, methods, properties, events, and constants of all COM components whose type libraries are referenced by the application you are working with. By default, each Microsoft® Office XP application references a set of type libraries. For example, Word references by default Visual Basic for Applications, Microsoft® Word, OLE automation, and Microsoft® Office XP type libraries.

To manually reference any additional type libraries available on your system

- From the **Tools** menu in the **Visual Basic Editor** , click **References** .

The Object Browser



The Project/Library box shows all the available referenced type libraries. These libraries make it possible for you to use early binding with the corresponding applications.

The Search Text box shows any keywords you have searched for by using the drop-down list. You can also type a word in this box, and then click the Search button to search the available libraries for that word. The Search Results list displays any classes, properties, methods, events, or constants that contain the word you searched for.

The Classes list displays all objects and collections in the library, and the Members of list displays all methods, properties, events, and constants associated with the selected object in the Classes list. The bottom pane of the dialog box (the Details pane) displays other information about the currently selected item, such as what kind of object it is, its data type, what arguments it might take, and what library or collection of which the item is a member.

If a Help file has been associated with the objects in the type library, you can display a Help topic by clicking the item in either the Classes or Members of list, and then pressing F1 or clicking the Help button in the upper-right corner of the dialog box.

Working with the Outlook Object Model

The programming model of Microsoft® Outlook® differs somewhat from the other Microsoft® Office applications. You can work with Outlook's object model in three ways:

- You can write Microsoft® Visual Basic® for Applications (VBA) code that runs from a local project file or a COM add-in that is associated with the local installation of Outlook.
- You can use the native scripting environment available within the Outlook forms that are used to display items such as messages and appointments.
- You can use automation to work with Outlook from other Office applications and applications that support VBA.

To write VBA code that runs from a local project file (VBAProject.OTM), open Outlook, point to Macro on the Tools menu, and then click Visual Basic Editor. In Outlook, the Visual Basic Editor makes it possible for you to write code that can be run from this installation of Outlook only. For example, by adding code to the ThisOutlookSession module, you can write code against the following Application object events: ItemSend, NewMail, OptionsPagesAdd, Quit, Reminder, and Startup. Just as with other Office applications, you can insert code modules, class modules, and UserForms to further customize your solution, and you can run procedures from menu items or toolbar buttons.

To distribute a solution created by using a local Outlook VBA project, you can export your modules and objects to files and then import them on other machines where Outlook is installed. However, a much better way to distribute your solution is to compile and install your solution as a COM add-in by using the COM add-in designer available in Microsoft® Office XP Developer or in Microsoft® Visual Basic® 6.0.

To write script that runs within Outlook items, you use the Outlook Script Editor. Outlook forms (that is, all items you can open in Outlook, such as messages, appointments, and contacts) support scripting in Microsoft® Visual Basic® Scripting Edition (VBScript) by using the Outlook Script Editor. Because VBScript is a subset of VBA, there are limitations to what you can do; for example, VBScript supports only one data type, the Variant data type, and a number of VBA keywords and features aren't supported. To access the Outlook Script Editor, you must be in form design mode.

To open the Outlook Script Editor

1. Open the Outlook item you want to base your form on.
2. On the **Tools** menu, point to **Forms** , and then click **Design This Form.** To start from an existing custom form, point to **Forms** on the **Tools** menu, click **Design a Form** , and then select a form.
3. Make any changes you want to the design of the form.
4. On the **Form** menu, click **View Code.**

When working with scripting in Outlook forms, you will most typically be writing event procedures. For example, you might want to write an event procedure for your form's Open event to initialize the form to display a particular tab page and enter default values in certain fields.

To add an event handler stub to the Outlook Script Editor

1. From the **Script** menu, click **Event Handler.**
2. Select the event you want to work with, and then click **Add.**

For more information about the Outlook object model, click Microsoft® Outlook® Object Library Help on the Help menu in the Outlook Script Editor.

To work with Outlook from another application, you can use automation code with either early binding or late binding. To use early binding, establish a reference to the Microsoft® Outlook® object library and then declare and initialize an object variable that references the Outlook Application object. For example:

```
Dim olApp As Outlook.Application
Set olApp = New Outlook.Application
```

Similarly, you can use the CreateObject function with an object variable declared as type Object to initialize a late-bound object variable. For example:

```
Dim olApp As Object
Set olApp = CreateObject("Outlook.Application")
```

Either way, you can then use this object variable to work with the other objects, properties, methods, and events of the Outlook object model.

Office Application Automation

Automating one application from another is often referred to as running code from a host application to automate another application. Although automating one Office application from another is generally accomplished in the same way, there are some important differences regarding how you work with each application.

When Microsoft® Visual Basic® for Applications (VBA) code references an object that is not installed, the Windows installer technology will attempt to install the required feature. In all Microsoft® Office applications except Microsoft® Outlook® and Microsoft® FrontPage®, you can use the FeatureInstall property to control what happens when an uninstalled object is referenced. When this property is set to the default (msoFeatureInstallOnDemand), any attempt to use an uninstalled object causes the Microsoft® Windows® installer to try to install the requested feature.

In some circumstances, this might take some time, and the user might believe that the machine has stopped responding to additional commands. To address this, you can set the FeatureInstall property to msoFeatureInstallOnDemandWithUI to display a progress meter so that users can see that something is happening as the feature is being installed.

If you want to trap the error that is returned and display your own dialog box to the user or take some other custom action, you can set the FeatureInstall property to msoFeatureInstallNone. For more information and details about application-specific behavior, search the Office application's Help index for "FeatureInstall property."

Setting References

The first step in automating one Microsoft® Office application from another is referencing the application you want to automate. This reference lets your application "see" the objects exposed by the other application. In most cases, this means setting a reference to the application's type library by using the References dialog box.

Before you work with objects exposed by an Office application, you should set a reference to that application by using the References dialog box.

To open the References dialog box

- Click **References** on the **Tools** menu in the **Visual Basic Editor** .

In your custom solutions, you must reference only the application that contains the objects you want to manipulate by using automation. Including unnecessary references will increase the time it takes for your solution to load and will consume some additional memory resources.

You can use the objects of another Office application (or the objects exposed by any other application or component that supports automation) without setting a reference in the References dialog box by using the CreateObject or GetObject function and declaring object variables as the generic Object type.

If you use this technique, the objects in your code will be late-bound, and as a result you will not be able to use design-time tools such as automatic statement completion or the Object Browser, and your code will not run as fast.

Tip
Because the Application object of every Office XP application includes accessor properties to work with some of the shared Office components such as the Assistant and FileSearch objects, you can work with these objects without having a reference to the Microsoft Office XP object library. You might want to do this if your application must load quickly. However, when you are using a shared Office component without a reference to the Microsoft® Office XP object library, your code can't use enumerated constants; if it does, an error will be displayed. For example, when you are using the Assistant object with a reference to the Microsoft® Office XP object library, you can use a line of code such as the following to animate the Office Assistant:

Application.Assistant.Animation = msoAnimationGreeting

Tip
To use the same line of code without a reference to the Microsoft® Office XP office library, you must use the actual value of the msoAnimationGreeting constant, which is 2, as in the following line of code:

Application.Assistant.Animation = 2

Tip
However, this is not a recommended coding procedure, because these numbers could change during the next revision. Using the constant makes sure your code will not break because of a change in the number. To determine the values for constants such as msoAnimationGreeting, you must temporarily establish a reference to the Microsoft® Office XP office library and use the Object Browser to look up the numeric values of the constants you want work with. Using the numeric values will make your code less readable, and Microsoft doesn't guarantee that the same value will be used in future versions of Microsoft® Office, so code written in this manner might not work correctly in future versions of Office. The VBA projects for all Office applications except Access include a reference to the Microsoft® Office XP office library by default. Therefore, if you want to prevent a reference to the Microsoft® Office XP office library from being loaded when your solution is opened, you must remove the reference in your solution's VBA project.

When you refer to an object in code, VBA determines what type of object it is by searching the type libraries selected in the References dialog box in the order in which they are displayed. If an object has the same name in two or more referenced type libraries, VBA uses the definition provided by the type library listed higher in the Available References list.

To change the order in which the libraries are searched, you can use the Priority buttons to move the type libraries (except for the Visual Basic for Applications and the host application's type library) up or down the list. However, a better way to eliminate ambiguous object references is to fully qualify type declarations by including the programmatic identifier in front of the object name; for example, Dim docNew As Word.Document. Qualifying type declarations by using the programmatic identifier eliminates a potential source of errors and also makes your code more self-documenting.

If you have established a reference to an application or component's type library, you can learn about the exposed objects by using the Object Browser and the Help system.

Object Variable Declaration

Before one application can work with the objects exposed by another application's type library, it must first determine what information is contained in that type library. The process of querying the objects, methods, and properties exposed by another application is called binding. Microsoft® Visual Basic® for Applications (VBA) programming in Microsoft® Office applications supports two kinds of binding: early binding and late binding. How and when binding occurs can have a great impact on how your solution performs.

If you establish a reference to the application's or component's type library, you can use early binding. When early binding is used, VBA retrieves information at design time about the application's objects directly from the type library, thus making it possible for you to declare object variables as specific types. For example, if you establish a reference to the Microsoft®Word object library when you are working with Word documents, you can declare object variables by using data types that are specific to Word, such as the Documents or Document types. Early binding reduces the amount of communication that needs to occur when your solution is running, thereby enhancing your solution's performance.

Late binding queries the application you are automating at run time, which makes it possible for you to declare object variables by using the generic Object or Variant data type. In general, late binding is useful if you are writing generic code to run against any of several applications and won't know the type of object you are working with until run time. Note that the additional overhead of querying an application at run time can slow down the performance of your solution.

Note
Some applications and components that support automation support only late binding. All Office XP applications and most contemporary applications that support automation support both early and late binding. However, scripting languages such as VBScript and Microsoft® JScript® don't support early binding because they don't support references or specific object data types (for example, in VBScript only the Variant data type is supported).

Early-Bound Declarations

Early binding makes it possible for you to declare an object variable as a programmatic identifier, or class name, rather than as an Object or a Variant data type. The programmatic identifier of an application is stored in the Microsoft® Windows® registry as a subkey below the \HKEY_CLASSES_ROOT subtree. For example, the programmatic identifier for Microsoft® Access is "Access.Application"; for Microsoft® Excel it is "Excel.Application."

When you are using early binding, you can initialize the object variable by using the CreateObject or GetObject function or by using the New keyword if the application supports it. All Office XP applications can be initialized by using the New keyword. Because the Microsoft® Outlook® programming environment for Outlook items supports only scripting, you can't use early binding declarations of any sort in its VBScript programming environment; however, you can use early binding in VBA code in a local Outlook VBA project or COM add-in, or in automation code that works with Outlook from another host application.

Early binding is the friendly name for what C programmers call virtual function table binding, or vtable binding. To use early binding, the host application must establish a reference to a type library (.tlb) or an object library (.olb), or an .exe, .dll, or .ocx file that contains type information about the objects, methods, properties, and events of the application or service you want to automate.

In the following code fragment, an Application variable is declared by using the programmatic identifier for Word (Word.Application) and a new instance of Word is created by using the Set statement with the New keyword:

Dim wdApp As Word.Application

Set wdApp = New Word.Application

If the code following these lines doesn't set the Application object's Visible property to True, the new instance of Word will be hidden. All Office applications are hidden by default when they are automated from another application.

Use early binding whenever possible. Early binding has the following advantages:

Syntax checking When you use early binding, VBA checks the syntax of your statements against the syntax stored in the object library during compilation rather than checking it at run time, so that you can catch and address errors at design time. For example, VBA can determine if you are using valid properties or methods of an object, and if you are passing valid arguments to those properties and methods.

Support for statement-building tools When you use early binding, the Visual Basic Editor supports features that make writing code much easier and less prone to errors, such as automatic listing of an object's properties and methods, and pop-up tips for named arguments.

Support for built-in constants When you use early binding, your code can refer to the built-in constants for method arguments and property settings because this information is available from the type library at design time. If you use late binding, you must define these constants in your code by looking up the values in the application's documentation.

Better performance Performance is significantly faster with early binding than with late binding.

Late-Bound Declarations

Late binding makes it possible for you to declare a variable as an Object or a Variant data type. The variable is initialized by calling the GetObject or CreateObject function and specifying the application's programmatic identifier. For example, in the following code fragment, an Object variable is declared and then set to an instance of Microsoft® Access by using the CreateObject function:

Dim objApp As Object

Set objApp = CreateObject("Access.Application")

Late binding is the friendly name for what C programmers used to call IDispatch binding, and was the first method of binding implemented in applications that can control other applications through automation. For this reason, you can use late binding to maintain backward compatibility with older applications. However, late binding uses a lot of overhead; it is faster than dynamic data exchange (DDE), but slower than early binding.

Tip
DDE is a protocol that was established before OLE for exchanging data between Windows applications. There is no need to use DDE to exchange data between Office applications because of their support for automation. However, you might have to use DDE from some other application that doesn't support automation code to work with data from an Office application. For more information about using DDE, search the Visual Basic Reference Help for the Office application you want to work with.

The CreateObject function must also be used to work with objects from any automation component from script. This is because scripting has no method of establishing references to type libraries to support early binding.

Creation of Object Variables to Automate Another Office Application

Working with the objects from one Microsoft® Office application to another Office application through Microsoft® Visual Basic® for Applications (VBA) code is very similar to using code to work with the objects within the code's host application. In most cases, you begin by creating an object variable that points to the Application object representing the Office application that contains the objects you want to work with. In general, you create an early-bound object variable by using the New keyword. However, there are limited circumstances where you might choose to use the CreateObject or GetObject function to create an object variable.

When you write VBA code in an application that manipulates objects within that same application, the reference to the Application object is implicit. When you are automating another application, the reference to the Application object generally must be explicit. The following two examples illustrate this difference. The first example contains VBA code intended to be run in Microsoft® Word. The second example contains VBA code intended to be run from another Office application (or any application that supports automation through VBA). For the second example to work, a reference must be set to the Microsoft® Word object library in the application the code is run from.

```
Sub CodeRunningInsideWord()
  Dim docNew As Word.Document
  ' Add new document to Documents collection.
  Set docNew = Documents.Add
  ' Type text into document.
  Selection.TypeText "Four score and seven years ago"
  ' Display document name and count of words, and then close document without  saving changes.
  With docNew
    MsgBox """ & .Name & "' contains " & .Words.Count & " words."
    .Close wdDoNotSaveChanges
  End With
  Set docNew = Nothing
End Sub

Sub CodeRunningOutsideWord()
  Dim wdApp      As Word.Application
  Dim docNew   As Word.Document

  ' Create new hidden instance of Word.
  Set wdApp = New Word.Application
  ' Create a new document.
```

```
  Set docNew = wdApp.Documents.Add
  ' Add text to document.
  wdApp.Selection.TypeText "Four score and seven years ago"
  ' Display document name and count of words, and then close  document without saving changes.
  With docNew
    MsgBox """ & .Name & "' contains " & .Words.Count & " words."
    .Close wdDoNotSaveChanges
  End With
  wdApp.Quit
  Set wdApp = Nothing
End Sub
```

In most cases, you will create an object variable that refers to the top-level object representing the application you want to access through automation, the Application object. When you have the reference to the Application object, you use additional references to that object's child objects to navigate to the object or method you want to manipulate. You assign object variables to child objects by using a method of a higher-level object with the Set statement.

However, Microsoft®Excel and Word also make it possible for you to create a top-level reference to certain child objects of the Application object. For this reason, it is possible to rewrite the previous CodeRunningOutsideWord procedure to start from a reference to a Word Document object, this way:

```
Sub CodeRunningOutsideWord()
  Dim docNew As Word.Document

  Set docNew = New Word.Document
  Set docNew = Documents.Add
  ' The following line uses the Application property to access the  implicit instance of the Word Application object.
  docNew.Application.Selection.TypeText "Four score and seven years ago"
  With docNew
    MsgBox """ & .Name & "' contains " & .Words.Count & " words."
    .Close wdDoNotSaveChanges
  End With
  docNew.Application.Quit
  Set docNew = Nothing
End Sub
```

Similarly, Excel makes it possible for you to create a top-level reference starting from the Workbook object. You can do this in either of two ways:

- By using the Excel.Sheet class name to create a workbook that contains a single worksheet.

or-

- By using the Excel.Chart class name to create a workbook that contains a worksheet with an embedded Chart object and another worksheet that contains a default data set for the chart.

To create a Workbook object either way, you must use the CreateObject function, because the Excel.Sheet and Excel.Chart class names don't support the New keyword. For example, to automate Excel starting with a top-level reference to a Workbook object that contains a single worksheet, use code such as this:

```
Dim wbkSheet As Excel.Workbook

Set wbkSheet = CreateObject("Excel.Sheet")
```

To automate Excel starting with a top-level reference to a Workbook object that contains a worksheet with a chart and another worksheet containing a default data set for the chart, use code such as this:

```
Dim wbkChart As Excel.Workbook

Set wbkChart = CreateObject("Excel.Chart")
```

When you are automating Word starting from a Document object or automating Excel starting from a Workbook object, an implicit reference is created to the Application object. If you must access properties and methods of the Application object, you can use the Application accessor property of the Document or Workbook objects. While using the Document or Workbook objects as top-level objects might reduce the amount of code you have to write somewhat, in most cases your code will be easier to understand and more consistent if you start from a reference to the Application object.

The following table shows all the top-level Office objects you can reference and their class names.

Object type	**Class name**
Access application	Access.Application
Excel application	Excel.Application

Excel workbook	Excel.Sheet Excel.Chart
FrontPage application	FrontPage.Application
Outlook application	Outlook.Application
PowerPoint application	PowerPoint.Application
Word application	Word.Application
Word document	Word.Document

Automating the Visual Basic Editor

In addition to using code to work with other Microsoft®Office applications, you can also use automation code to work with the objects exposed by the Microsoft® Visual Basic® Editor object model. You can use the Visual Basic Editor's object model to work with the objects in its user interface, such as its windows and command bars, which makes it possible for you to develop add-ins to customize and extend the Visual Basic Editor's user interface. Additionally, you can use the Visual Basic Editor's object model to work with your Microsoft® Visual Basic® for Applications (VBA) project itself to add and delete references, to set and read project properties, and to work with the components that make up your project, such as standard modules, class modules, and UserForms. This feature makes it possible for you to write code to maintain references, to document and set properties for projects, and to work with existing components and add new ones.

To work with the Visual Basic Editor's objects, first you must establish a reference to its type library, which is named Microsoft® Visual Basic® for Applications Extensibility 5.3. To write code to work with the Visual Basic Editor, you must initialize a variable to work with the Visual Basic Editor's top-level object, the VBE object. However, you can't reference the VBE object directly. This is because the Visual Basic Editor isn't an independent application or service; it's running as part of the host application's process. To initialize an object variable to work with the Visual Basic Editor, you must use the VBE accessor property of the host application's Application object. The VBE property is available in all Office applications except Outlook. The following example shows how to initialize an object variable to work with the Visual Basic Editor:

```
Dim objVBE As VBIDE.VBE

Set objVBE = Application.VBE
```

For an overview of working with the Visual Basic Editor's object model, see the Visual Basic Language Developer's Handbook by Ken Getz and Mike Gilbert (Sybex, 1999).

Note
The Microsoft® Access Application object provides a References collection and Reference object that make it possible for you to work with references in an Access VBA project without requiring you to establish a reference to the Microsoft Visual Basic for Applications Extensibility 5.3 type library. For more information about the Access References collection, search the Microsoft Access Visual Basic Reference Help index for "References collection."

The Set Statement and the New Keyword in Automation

You start automation code by declaring object variables with a specific object type that represents the top-level object and then declaring any child objects you want to reference. You then create an instance of the top-level object by using the Set statement and the New keyword. However, the New keyword can't be used to create a new instance of a child object. To create an instance of a child object, use the appropriate method of the parent object along with the Set statement.

In the following example, the top-level Microsoft® Excel Application object variable is assigned by using the Set statement and the New keyword. The object variable representing the Workbook child object is assigned by using the parent object's Add method and the Set statement.

```
Sub CreateExcelObjects()
  Dim xlApp          As Excel.Application
  Dim wkbNewBook     As Excel.Workbook
  Dim wksSheet       As Excel.Worksheet
  Dim strBookName    As String

  ' Create new hidden instance of Excel.
  Set xlApp = New Excel.Application
  ' Add new workbook to Workbooks collection.
  Set wkbNewBook = xlApp.Workbooks.Add
  ' Specify path to save workbook.
  strBookName = "c:\my documents\xlautomation.xls"
```

```
  ' Loop through each worksheet and append " - By Automation" to the name of each sheet. Close and save workbook_
  ‘ to specified path.
  With wkbNewBook
    For Each wksSheet In .Worksheets
      wksSheet.Name = wksSheet.Name & " - By Automation"
    Next wksSheet
    .Close SaveChanges:=True, FileName:=strBookName
  End With
  Set wkbNewBook = Nothing
  XlApp.Quit
  Set xlApp = Nothing
End Sub
```

Note
The CreateExcelObjects procedure uses three Excel object variables, b ut only the first two are instantiated by using the Set statement. You do not need to use the Set statement to create an object variable that will be used only inside a For...Each loop.

In the next example, the top-level Microsoft®Outlook® Application object is created by using the Set statement and the New keyword. The MailItem child object variable is created by using the Application object's CreateItem method. The Recipient child object is created by using the Add method of the MailItem object's Recipients collection.

```
Sub CreateOutlookMail()
  Dim olApp              As Outlook.Application
  Dim olMailMessage      As Outlook.MailItem
  Dim olRecipient        As Outlook.Recipient
  Dim blnKnownRecipient  As Boolean

  ' Create new instance of Outlook or open current instance.
  Set olApp = New Outlook.Application
  ' Create new message.
  Set olMailMessage = olApp.CreateItem(olMailItem)
  ' Prompt for message recipient, attempt to resolve address, and  then send or display.
  With olMailMessage
    Set olRecipient = .Recipients.Add(InputBox("Enter name of message recipient", "Recipient Name"))
    blnKnownRecipient = olRecipient.Resolve
    .Subject = "Testing mail by Automation"
    .Body = "This message was created by VBA code running "  & "Outlook through Automation."
    If blnKnownRecipient = True Then
      .Send
    Else
      .Display
    End If
  End With
  Set olMailMessage = Nothing
  olApp.Quit
  Set olApp = Nothing
End Sub
```

Note
At the end of this procedure, each object variable is destroyed by explicitly setting it equal to the Nothing keyword.

You can also use the New keyword to create a new instance of the object at the same time you declare its object variable. For example:

```
Dim olApp As New Outlook.Application
```

If you do this, there is no need to use a Set statement to instantiate the object. However, this technique is not recommended because you have no control over when the object variable is created. For example, if your code must test to see if an object exists by using a statement such as If olApp Is Nothing Then, this test will return True if you have created an instance of the object in the Dim statement. Additionally, you might not need to use an object except at the user's request. If you create an instance of the object by using New in the Dim statement, the object will be created even if it isn't used. To maintain control over when an object is created, don't use the New keyword in the Dim statement, and instantiate the object by using a Set statement at the point in your code where you must use the object.

Single-Use vs. Multi-Use Applications

Whether you return a reference to a new instance of the Application object or an existing instance depends on whether the application's default behavior is as a single-use or a multi-use application. A single-use application causes a new instance of that application to be created whenever an object variable is instantiated in any host application. For example, Microsoft®

Word is a single-use application, so the following code creates a new instance of Microsoft® Word regardless of how many instances of Word might be running already:

Dim wdApp aAs Word.Application

Set wdApp = New Word.Application

A multi-use application makes it possible for host applications to share the same instance of the application. The next example creates a new instance of Microsoft®Outlook®only if Outlook is not running when the code is executed. Because Outlook is a multi-use application, if Outlook is running already when this code is run, the object variable points to the currently running instance.

Dim olApp As Outlook.Application

Set olApp = New Outlook.Application

The following table shows the default behavior for each Office application.

Application	Application type
Access	Single-use
Excel	Single-use
FrontPage	Single-use
Outlook	Multi-use
PowerPoint	Multi-use
Word	Single-use

You can use the GetObject function to create an object variable that references a currently running instance of a single-use application.

If you create an object variable that points to a multi-use application (Outlook or Microsoft® PowerPoint®) and an instance of the application is running already, any method you use to create the object variable will return a reference to the running instance. For example, if Outlook is running already, the following lines of code all return a reference to the same instance of Outlook:

Dim olApp1 As Outlook.Application
Dim olApp2 As Outlook.Application
Dim olApp3 As Outlook.Application

Set olApp1 = New Outlook.Application
Set olApp2 = CreateObject("Outlook.Application")
Set olApp3 = GetObject(, "Outlook.Application")

Using the CreateObject and GetObject Functions

You can use the Set statement with the CreateObject and GetObject functions to create a top-level object variable that represents a Microsoft® Office application. These functions should be used only in those situations where the New keyword does not provide the functionality you require.

You use the CreateObject function to create a top-level object variable that represents an Office application in the following two situations:

The Office application for which you want to create an Application object is not available on the local computer but is available on some other computer on your network. For example, you can run Microsoft® Visual Basic® for Applications (VBA) code that prints reports from a Microsoft® Access database that is located on a network server even though Access is not installed on the computer from which the code is run. If Access is installed on the network server, you can create an Access Application object that runs on the server by specifying the name of the server in the CreateObject function's optional servername argument. For example:

Dim objAcApp As Object

Set objAcApp = CreateObject("Access.Application", "MyServer1")

The servername argument of the CreateObject function is the same as the machine name portion of a share name. Therefore, for a share named \\MyServer1\Public, the servername argument is "MyServer1".

To successfully run an Office application as a remote server, you must configure Distributed Component Object Model (DCOM) settings on the computer that is acting as a server, and also possibly on the client computers. To configure DCOM,

run the Distributed COM Configuration utility (Dcomcnfg.exe) from the Run box on the Startup menu. For more information about configuring DCOM, search the Microsoft Technical Support Web site (http://support.microsoft.com55) for "Configure DCOM."

The CreateObject function is also useful when you are not sure if the Office application you want to automate will be installed on the computer that runs your code. The following example illustrates how to use the CreateObject function to make sure an application is available for automation:

```
Sub CreateObjectExample()
  Dim objApp As Object
  Const ERR_APP_NOTFOUND As Long = 429

  On Error Resume Next
  ' Attempt to create late-bound instance of Access application.
  Set objApp = CreateObject("Access.Application")
  If Err = ERR_APP_NOTFOUND Then
    MsgBox "Access isn't installed on this computer. "  & "Could not automate Access."
    Exit Sub
  End If
  With objApp
    ' Code to automate Access here.
    .Quit
  End With
  Set objApp = Nothing
End Sub
```

Note
The Application object variable in this procedure is declared by using the Object data type and is late-bound to the application by using the CreateObject function. The code must be written this way, because, if an object variable is declared as a specific Application object type and that application is not present, the code will break.

Note
The CreateObject function also must be used to work with objects from any automation component from script. This is because scripting has no method of establishing references to type libraries to support early binding. However, for security reasons, you wouldn't typically use the CreateObject function from script to create an instance of an Office application.

You can use the GetObject function in these situations:

1. You must create a reference to a running instance of an application. For example, the following code creates a reference to the running instance of Access. If Access is not running when the code executes, a Set statement is used to create an object variable for the Access Application object.

```
Sub GetObjectExample()
  Dim acApp As Access.Application
  Const ERR_APP_NOTRUNNING As Long = 429

  On Error Resume Next
  ' Attempt to reference running instance of Access.
  Set acApp = GetObject(, "Access.Application")
  ' If Access isn't running, create a new instance.
  If Err = ERR_APP_NOTRUNNING Then
    Set acApp = New Access.Application
  End If
  With acApp
    ' Code to automate Access here.
  End With
  ' If instance of Access was started by this code,  shut down application.
  If Not acApp.UserControl Then
    acApp.Quit
    Set acApp = Nothing
  End If
End Sub
```

   If multiple instances of the application you want to automate are running, there is no way to guarantee which instance the GetObject function will return. For example, if two sessions of Access are running and you use the GetObject function to retrieve an instance of Access from code running in Excel, there's no way to guarantee which instance of Access will be used.

   There are few circumstances where it makes sense to use the GetObject function to return a reference to a running instance of an Office application. If a user opened the running instance, you would rarely want your code to be manipulating the objects in that instance of the application. However, when you use the Shell function to start an

Access application (so that you can supply a password and workgroup information file to open a secured database), it does make sense to work with the running instance of Access by using the GetObject function to return a reference to the instance of Access that you started.

2. You also use the GetObject function when you must open an Office file and return a reference to the host application object at the same time. The following example shows how to use the GetObject function to open an Access database from disk and return a reference to the Access application. When HTML is passed as the value for the lngRptType argument, the procedure creates a Web page from a report and displays that page in a Web browser.

```
Function GetReport(Optional lngRptType As opgRptType) As Boolean
  ' This function outputs a report in the format specified by  the optional lngRptType argument. If lngRptType is specified,
  ' the report is automatically opened in the corresponding  application.
  ' lngRptType can be any of the following constants defined  by Enum opgRptType in the Declarations section of this  module:
  ' XLS = output to Excel, RTF = output to Rich Text Format,  SNAPSHOT = output to Access snapshot report format
  ' HTML = output to HTML. If lngRptType is not specified, the report is opened in Access and displayed in Print Preview.
  Dim acApp            As Access.Application
  Dim strReportName    As String
  Dim strReportPath    As String
  Const SAMPLE_DB_PATH As String = "c:\program files\"  & "microsoft office\office\samples\northwind.mdb"

  strReportName = "Alphabetical List of Products"
  strReportPath = "c:\my documents\"
  ' Start Access and open Northwind Traders database.
    Set acApp = GetObject(SAMPLE_DB_PATH, "Access.Application")
  With acApp
    ' Output or display in specified format.
    With .DoCmd
      Select Case lngRptType
        Case XLS
        .OutputTo acOutputReport, strReportName,  acFormatXLS, strReportPath & "autoxls.xls", True
        Case RTF
        .OutputTo acOutputReport, strReportName, acFormatRTF, strReportPath & "autortf.rtf", True
        ' Snapshot Viewer must be installed to view snapshot  output.
        Case SNAPSHOT
        .OutputTo acOutputReport, strReportName,  acFormatSNP, strReportPath & "autosnap.snp", True
        Case HTML
        .OutputTo acOutputReport, strReportName,  acFormatHTML, strReportPath & "autohtml.htm", _
                  True, "NWINDTEM.HTM"
        Case Else
        acApp.Visible = True
        .OpenReport strReportName, acViewPreview
      End Select
    End With
    ' Close Access if this code created current instance.
    If Not .UserControl Then
      acApp.Quit
      Set acApp = Nothing
    End If
  End With
End Function
```

Working with Documents That Contain Startup Code

Using automation to open a document does not prevent a document's startup code from running. Startup code can be defined in various ways in Microsoft® Office applications, as explained in the following table.

Application	Startup code location
Word	Startup code is contained in the event procedures for the Open or New events in the ThisDocument module of a document or template.
Excel	Startup code is contained in the event procedure for the Open event in the ThisWorkbook module of a workbook or template.
Outlook	Startup code is contained in the event procedure for the Startup event in the ThisOutlookSession of the local Outlook VBA project.

Access	If you create an Access macro named AutoExec, this macro's actions will run on startup. You can also place startup code in the event procedure for the startup form's Open event. To specify a form to be opened on startup, use the Startup command on the Tools menu.

Note

Microsoft® PowerPoint® and Microsoft® FrontPage® documents don't have a way to define startup code.

Because startup code might display message boxes or modal forms that act as dialog boxes, these message or dialog boxes might prevent your code from proceeding until a user closes or responds to them. If you have startup code in a Microsoft® Excel workbook or a Microsoft® Access database that you don't want to run if the document is opened programmatically from another application, you can use the UserControl property of the Application object to determine how a document is being opened and then act accordingly. If you can't use the UserControl property, you might need to use a SendKeys statement to send keystrokes to close the message or dialog box.

In Excel, the UserControl property will return False only when the document or workbook is opened from automation by using a hidden instance of the Excel Application object (Application.Visible = False). For example, the following code defined in an Excel workbook's Open event procedure will run only if the workbook is opened by a user or a visible instance of the Excel Application object. If you open the workbook by using a hidden instance of the Excel Application object from code running in another application, the message box won't be displayed.

```
Private Sub Workbook_Open()
  Dim strMsg As String

  strMsg = "This message was triggered by this workbook's " &  "Open event." & vbCrLf & _
       "It won't be displayed if this workbook is opened by using a hidden" & vbCrLf & _
       "instance of the Excel Application object from Automation code."
  ' If opened through Automation by using a hidden instance,  the UserControl property will be False.
  If Application.UserControl = True Then
    MsgBox strMsg
  End If
End Sub
```

Note

In Microsoft® Word 97 and later, there is no way to prevent Open event code from running with the UserControl property. If Word is visible to the user, or if you call the UserControl property of a Word Application or Document object from within a Word code module, this property will always return True. However, you can still use the Word UserControl property from automation code (that creates a hidden instance of Word) running from another application to determine if a document was opened programmatically or by the user.

In Access, you don't have to check or keep track of whether the instance of the Application object is hidden or visible because the UserControl property is False whenever the application is started from code. To control whether code in the startup form's Open event is executed, Access provides a Cancel argument for the Open event. As shown in the following example, you can set the Cancel argument to True to keep a startup form from opening if you open the database by using automation code:

```
Private Sub Form_Open (Cancel As Integer)
  ' If database is opened from Automation,  cancel the Open event of the form.
  If Application.UserControl = False Then
    Cancel = True
  Else
  ' Any startup code that needs to run when the  database is opened by a user goes here.
  End If
End Sub
```

You can also use the UserControl property of the Access Application object to control whether actions in a database's AutoExec macro will run when the database is opened from another program by using automation. To do this, you must enter Application.UserControl = True in the Condition column for each action you want to cancel. (To display the Condition column, click Conditions on the View menu.)

Tip

You can also use COM add-ins to implement a startup form or code. COM add-ins support events that you can use to determine how an application was loaded before connecting the add-in.

Shutting Down Objects Created by Using Automation

A local variable is normally destroyed when the procedure in which it is declared is finished executing. However, it is good programming practice to explicitly destroy an application-level object variable used to automate another application by setting it equal to the Nothing keyword. Doing this frees any remaining memory used by the variable. For some Application objects,

you might also have to use the object's Quit method to completely destroy an object variable and free up the memory it is using. As a general rule, it's safest to do both: Use the Quit method and then set the object variable equal to the Nothing keyword.

There might be situations where you must determine if the instance of an application you are working with was created by your code before shutting it down. Generally, you can inspect the UserControl property of the Application object to determine if your code opened the current instance. However, there are cases where the value of the UserControl property can change from False to True as your code executes. For example, if you start Microsoft® Excel through automation, make it visible, and make it possible for the user to interact with this instance, such as by typing something in a cell, the UserControl property will return True even though your code started the instance. To handle this situation, assign the value of the UserControl property to a variable right after you create the instance of the Application object, and use this variable to test the value of the UserControl property before closing the application, as shown in the following example:

```
Sub GetObjectXL()
  Dim xlApp             As Excel.Application
  Dim blnUserControl    As Boolean

  Const ERR_APP_NOTRUNNING As Long = 429
  ' Set blnUserControl to True as default.
  blnUserControl = True
  On Error Resume Next
  ' Attempt to open current instance of Excel.
  Set xlApp = GetObject(, "Excel.Application")
  ' If no instance, create new instance.
  If Err = ERR_APP_NOTRUNNING Then
    Set xlApp = New Excel.Application
    ' Store current state of UserControl property.
    blnUserControl = xlApp.UserControl
  End If
  With xlApp
    ' Code to automate Excel here. Check original value of UserControl property.
    If blnUserControl = False Then
      xlApp.Quit
      Set xlApp = Nothing
    End If
  End With
End Sub
```

Note

Microsoft® PowerPoint®, Microsoft® Outlook®, and Microsoft® FrontPage® have no method of determining if an instance of the Application object has been started by a user or program.

2. WORKING WITH OFFICE APPLICATIONS

Each Microsoft® Office XP application exposes an object model with hundreds of different objects, collections of objects, properties, methods, and events that you can take advantage of to build your application.

This section introduces the objects that you will use most often in each of the Office applications. This introduction helps you become immediately productive when you are working with Microsoft® Visual Basic® for Applications (VBA) in any Office application or when you are driving another application through Automation (formerly called OLE Automation).

In This Section

Working with Microsoft Access Objects
Use Form, Report, and DataAccessPage objects and the controls they contain to format and display data and make it possible to add or edit data in a database.

Working with Microsoft Excel Objects
Use Microsoft® Visual Basic®for Applications (VBA) to work with Microsoft® Excel objects, from within either Excel itself or another Microsoft® Office XP application to gain access to every part of Excel.

Working with Microsoft FrontPage Objects
Create, deploy, modify, and manage Web sites using Microsoft® FrontPag®.

Working with Microsoft Outlook Objects
Create custom Microsoft® Outlook® objects and manipulate those objects from within Outlook or from another application using VBA code from within Outlook or another Microsoft® Office XP application by using Automation.

Working with Microsoft PowerPoint Objects
Automate Microsoft® PowerPoint® by using the Application object, from which you can open an existing Presentation object or create a new presentation.

Working with Microsoft Project Objects
Build powerful custom applications easily with the Microsoft® Project object model.

Working with Microsoft Publisher Objects
Use Microsoft®Visual Basic® for Applications (VBA) to work with t he Microsoft® Publisher object model.

Working with Microsoft Word Objects
Use Microsoft® Visual Basic® for Applications(VBA) to work with the Microsoft® Word Document object, Application object, and Documents collection.

Working with Microsoft Visio Objects
Design, model, and manage complex enterprise-level systems with the sophisticated tool set provided by Microsoft® Visio®products.

Working with Microsoft Access Objects

Working with Microsoft® Access objects primarily means working with Form, Report, and DataAccessPage objects and the controls they contain. You can use these powerful Access objects to format and display data and make it possible for the user to add or edit data in a database. In addition, Access exposes many other objects you can use to work with your Access application; among the most important are the CurrentProject, CurrentData, CodeProject, CodeData, Screen, and DoCmd objects and the Modules and References collections. This section presents an overview of how to work with Access objects by using Microsoft® Visual Basic® for Applications (VBA).

Note
You can use the Object Browser and Access Visual Basic Reference Help to learn more about individual objects, properties, methods, and events.

Tables and relationships, the data in tables, and queries are managed and maintained by a database engine. For .mdb-type databases, Access uses the Microsoft® Jet database engine. For .adp-type databases, Access uses the Microsoft® SQL Server database engine or any other ActiveX Data Objects (ADO) data source. You programmatically work with tables, data in tables, or queries by using ADO or Data Access Objects (DAO).

In This Section

Understanding the Access Application Object
Use the properties and methods provided by the Application objects to create and work with other Access objects.

Built-in Access Functions and Methods
Learn about functions and methods that appear in the Object Browser as methods of the Application object.

Working with Reports, Forms, and Data Access Pages
Use reports, forms, and data access pages provided by Access to display data to the user.

Understanding the Access Application Object

The Application object is the top-level object in the Microsoft® Access® object model. It provides properties and methods you can use to create and work with other Access objects. It also provides several built-in functions you can use to work with the objects in your database. In essence, the Application object serves as the gateway to all other Access objects.

Application-wide options are available through the Options dialog box and the Startup dialog box. The commands to open these dialog boxes are located on the Tools menu. You can use the Options dialog box to specify or determine application-wide settings, such as whether the status bar is displayed, the new database sort order, and the default record-locking settings. You use the Startup dialog box to specify or determine settings such as which form opens automatically when your database opens and your database application's title and icon. The following sections discuss how you can use Microsoft®Visual Basic®for Applications (VBA) to access all of these settings.

Figure 1. Microsoft Access Object Model

Working with the Options Dialog Box Settings

Use the Application object's SetOption and GetOption methods to specify or determine the settings in the Options dialog box. Both methods use a string argument that identifies the option you want to access. The SetOption method takes an additional argument representing the value you want to set. For example, the following code displays a message box that indicates whether datasheet gridlines are turned on:

```
MsgBox "Horizontal Gridlines On = " &  CBool(GetOption("Default Gridlines Horizontal")) & vbCrLf _
   & "Vertical Gridlines On = " & CBool(GetOption("Default Gridlines Vertical"))
```

The next example illustrates how you can use the SetOption method to specify a new default database folder:

```
SetOption "Default Database Directory", "C:\NewMDBs"
```

To see a list of all the string arguments used to access settings in the Options dialog box, search the Microsoft® Access Visual Basic Reference Help index for "options, setting," open the topic "Set Startup Properties and Options in Code," and then jump to the topic "Set Options from Visual Basic."

The value returned by the GetOption method and the value you pass to the SetOption method as the setting argument depend on the type of option you are using. The following table establishes some guidelines for Options dialog box settings.

If the option is	**Then the value of the option is**
A text box	A string or numeric value
A check box	An integer that will be True (-1) (selected) or False (0) (not selected)
An option button in an option group, or an item in a combo box or a list box	An integer corresponding to the item's position in the option group or list (starting with 0 for the first item, 1 for the second item, and so on)

Note

If you use the SetOption method to change a user's Options dialog box settings, be sure to restore those settings when your code is finished executing or when your application ends. Otherwise, the settings you specify will be applied to any database the user opens. Note that the settings in the Options dialog box are stored in the Microsoft®Windows® registry in the \HKEY_CURRENT_USER\Software\Microsoft\Office\10.0\Access\Settings subkey. As a result, changes to these settings will not persist if the database is run on a different machine.

Understanding Startup Properties

You use startup properties to customize how a database application appears when it is opened. You work with startup properties differently than you do the settings in the Options dialog box. Each option in the Startup dialog box has a corresponding Access property, but you won't find these properties in the Object Browser. In a new database, the startup properties do not exist until a user makes a change to the default settings in the Startup dialog box.

To set these properties programmatically for an .mdb-type database, you must first add each property to the Properties collection of the Database object. This is true whether you are using DAO or ADO. In other words, even without a reference to DAO, you still use the Properties collection of the Database object to work with these properties. In an .adp-type database, startup properties are stored in the Properties collection of the CurrentProject object.

In the following sample, the AddCustomProperty sample procedure is used to set the AppTitle property in an .mdb-type database. Note that if the property does not exist when the AddCustomProperty procedure is called, the property is created and appended to the Properties collection of the Database object.

```
Const TEXT_VALUE As Integer = 10
If AddCustomProperty("AppTitle", TEXT_VALUE, "MyDatabase") Then    ' Property added to collection.
End If

Function AddCustomProperty(strName As String,  varType As Variant,  varValue As Variant) As Boolean
  ' The following generic object variables are required  when there is no reference to the DAO 3.6 object library.
  Dim objDatabase As Object
  Dim objProperty As Object
  Const PROP_NOT_FOUND_ERROR = 3270
  Set objDatabase = CurrentDb
  On Error GoTo AddProp_Err
  objDatabase.Properties(strName) = varValue
  AddCustomProperty = True
AddProp_End:
  Exit Function
AddProp_Err:
  If Err = PROP_NOT_FOUND_ERROR Then
    Set prpProperty = objDatabase.CreateProperty(strName, varType, varValue)
    objDatabase.Properties.Append objProperty
    Resume
  Else
    AddCustomProperty = False
    Resume AddProp_End
  End If
End Function
```

Note

Changes you make to any of the startup properties by using VBA will be available programmatically but will not take effect until the next time the database is opened.

Built-in Access Functions and Methods

The Microsoft® Access Application object contains several functions and methods you can use to work with data, Access objects, or the application itself. These functions and methods appear in the Object Browser as methods of the Application object, although they might be referred to as "functions." These functions and methods can be used within Access or from another application by using Automation.

Calling Built-in Access Functions and Methods Without Using an Application Object Variable

To use Automation, you usually have to create an instance of the Application object, but you can call built-in Access functions and methods of the Application object from other Microsoft® Office applications without first creating an Access Application object variable. The only requirements are that you set a reference to the Microsoft® Access object library in the calling application's Microsoft® Visual Basic® for Applications (VBA) project, and that you call the function or method by using the Access qualifier, as illustrated in the following example. For example, you could use the following VBA code to call the built-in Access Eval function to evaluate a string expression contained in a Microsoft® Word bookmark:

```
Dim rngResults As Word.Range
Set rngResults = ActiveDocument.Bookmarks("MathMark").Range
rngResults.Text = Access.Eval(rngResults.Text)
```

Note
Direct calls to built-in Access functions and methods, such as the one illustrated in the preceding example, automatically create a new instance of Access that remains in memory until the document containing the code that called the function or method is closed. If you want more control over when the instance of Access is created and destroyed, create it by using the New keyword or the CreateObject or GetObject function, and close it by setting the Application object variable equal to Nothing.

The following table summarizes some of the Access functions and methods available to you from the Application object and descriptions of how they might be used.

Function or method	Description
Domain aggregate functions	A domain is simply a set of records defined by a table or query. You use domain aggregate functions to get statistical information about a set of records, for example, to count the number of records or to determine the sum of values in a particular field. These functions use a naming convention that begins with a capital "D", for example, DAvg, DCount, DLookup, DSum, and so on. You can use these functions in VBA code, in a query expression, or in a calculated control on a form or report.
Eval function	You use this function to evaluate a string expression that results in a text string or numeric value. The Eval function uses a single argument that either is a string expression that returns a value or is the name of a built-in or user-defined function that returns a string. You can use the Eval function in a calculated control, a query expression, a macro, or VBA code.
GUIDFromString and StringFromGUID functions	You use these functions to convert a globally unique identifier (GUID) to a String value or a String value to a GUID. A GUID is a 16-byte value used to uniquely identify an object.
hWndAccessApp method	You can use this method to determine the handle (a unique Long Integer value) assigned by Microsoft® Window® to the main Access window. You can use the hWnd property to determine the handle assigned by Microsoft Windows to an Access Form or Report window.
HyperlinkPart function	The HyperlinkPart function returns information about data stored in a field that has the Hyperlink data type. This information is similar to the information contained in the properties of a Hyperlink object. You can use this function in VBA code, a query expression, or a calculated control.
LoadPicture method	This method loads a graphic file stored on disk into the Picture property of a control. You use this method to set or change the Picture property of a control at run time.
Nz function	You use the Nz function to evaluate a value and return a specified value if the evaluated value is Null. This function is useful when you are assigning values from a field in a recordset to a control that cannot use Null values.
SysCmd method	This is the Swiss army knife of Access methods. It can perform a variety of tasks depending on the value of the acSysCmdAction constant supplied in its action argument. For example, you can use this method to display a progress meter or text in the status bar, return information about Access (such as the directory where Msaccess.exe is located), or to get information about an Access object (such as whether a form is open).

Note
In addition to working with built-in Access methods and functions, you can use the Application object's Run method to call custom procedures stored in an Access database.

Creating, Opening, and Closing an Access Application

You can create a new database, or open and close an existing database, from within Microsoft® Access® or by using Automation from another application. The methods discussed in this section are typically used in Automation from another application. If your code is running inside Access, the code typically works with the currently open database, and using these methods is not necessary.

Note
If you are working in another application and you must access only the data in a database (tables or queries), and not objects such as forms or reports, you use ADO to access the data you require.

You use the NewCurrentDatabase method to create a new .mdb-type database. You use the OpenCurrentDatabase and CloseCurrentDatabase methods to open and close an existing .mdb-type database. The following sample is designed to be run from any Microsoft® Office application. It opens the Northwind Traders sample database and prints the portion of the Product Catalog report specified in the OpenReport method:

```
Sub PrintReport(strCategoryName As String)
  Dim acApp        As Access.Application
  Dim strDBPath    As String

  Const DB_PATH As String = _
    "c:\program files\microsoft office\office\samples\northwind.mdb"
  Set acApp = New Access.Application
  With acApp
    .OpenCurrentDatabase DB_PATH
    ' Print the Product Catalog report.
    .DoCmd.OpenReport "Catalog", acViewNormal, , "CategoryName = '" & strCategoryName & "'"
  End With
  acApp.Quit
  Set acApp = Nothing
End Sub
```

You use the NewAccessProject, OpenAccessProject, or CreateAccessProject method to open or create an .adp-type database. The NewAccessProject method creates a new .adp-type database and causes it to become active, whereas the CreateAccessProject method only creates an .adp file on disk. You use the OpenAccessProject method to open an existing .adp-type database and the CloseCurrentDatabase method to close an .adp-type database.

When you create a new database or have a database open, you can use other methods of the Application object to create new Access objects. For example, you use the CreateForm and CreateControl methods to create forms and controls on forms. You use the CreateReport and CreateReportControl methods to create reports and controls on reports. You use the CreateDataAccessPage method to create data access pages. To programmatically add controls to a data access page, you must use script or the Dynamic HTML (DHTML) object model to work with HTML directly.

Note
Although the methods discussed above let you programmatically create a database and the objects it contains, these methods typically are used only in wizards or add-ins. Generally, you create the database and its objects through the Access user interface and then work with these objects programmatically by using Microsoft® Visual Basic®for Applications (VBA) code run from Access or another Office application.

The CurrentData and CurrentProject Objects

In previous versions of Microsoft® Access, you can use Data Access Objects (DAO's) and their methods and properties to get information about forms, reports, macros, tables, fields, relationships, and queries. For example, you can use Document objects to get information about the tables and queries in a database. There are separate Container objects representing forms, reports, scripts (Access macros), tables (tables and queries), and modules. Each of these Container objects contains a collection of Document objects representing all the objects of the specified type in the current database. Each Document object contains only summary information about each object and does not provide access to the properties of the object or the data it contains. You use DAO Recordset objects to work with the data in a table or query, and you use members of the Forms or Reports collection to work with forms and reports themselves.

However, in Access, DAO is no longer the default programmatic way to interact with data and objects that contain data; therefore, Access has two new objects-CurrentData and CurrentProject-that contain collections of AccessObject objects, which are used in place of the Container and Document objects available through DAO in previous versions.

Access uses the CurrentData object to store collections of AccessObject objects that are administered by the database engine, for example, tables and queries in .mdb-type databases, and database diagrams, stored procedures, tables, and views in .adp-type databases. Information about each collection of objects is stored in a collection where each object is represented as an AccessObject object. For example, information about tables is contained in the AllTables collection, and information about views is stored in the AllViews collection. To access the CurrentData object, you use the CurrentData property of the Application object. When code is running in an add-in or library database, you would use the CodeData object to refer to the objects managed by the add-in or library database. The CodeData property of the Application object returns the CodeData object.

Note
AccessObject objects contain information about the objects that contain data, but do not provide access to the data itself.

You use the CurrentProject property of the Application object to get information about the Access objects in a database, such as data access pages, forms, macros, modules, and reports. The CurrentProject property of the Application object returns the CurrentProject object, which contains collections of AccessObject objects as well as information about the name, path, and connection of the database itself. For example, the AllForms collection contains information about all the forms in a database, and the AllReports collection contains information about all the reports in the database. When code is running in an add-in or library database, the CodeProject object contains the collections of AccessObject objects in the add-in or library database. The CodeProject property of the Application object returns the CodeProject object.

An AccessObject object exposes the following properties you can use to get information about an object: IsLoaded, Name, Parent, Properties, and Type. These properties are described in the following table.

AccessObject property	**Description**
IsLoaded	A Boolean value indicating whether the object is currently loaded. This property is True when an object is open in any view.
Name	A String value representing the name of the object.
Parent	Returns the parent object for the specified object. For example, the parent of an item in the AllForms collection is the AllForms collection object. The parent of the CurrentProject object is the Application object.
Properties	Returns an AccessObjectProperties collection, which contains all the custom properties associated with a particular AccessObject object. The Properties collection can store String or Long Integer values only.
Type	A Long Integer value representing one of the objects specified by the acObjectType intrinsic constants.

Note
Collections of AccessObject items are indexed beginning with a value of 0 for the first item in the collection, 1 for the second item, and so on.

The following sample shows how you can use the IsLoaded property to determine if a form, report, or data access page is currently loaded:

```
With CurrentProject
  Select Case intObjectType
    Case acForm
      IsObjectOpen = .AllForms(strObjName).IsLoaded
    Case acReport
      IsObjectOpen = .AllReports(strObjName).IsLoaded
    Case acDataAccessPage
      IsObjectOpen = .AllDataAccessPages(strObjName).IsLoaded
    Case Else
      Err.Raise ERR_INVALIDOBJTYPE
  End Select
End With
```

The intObjectType variable would be passed to a procedure as an argument of type acObjectType. The next sample illustrates how to add custom properties to a form:

```
Sub AddCustomFormProperty(strFormName As String, strPropName As String, varPropValue As Variant)
  ' This procedure illustrates how to add custom  properties to the Properties collection that
  ' is associated with an AccessObject object.
    With CurrentProject.AllForms(strFormName).Properties
    .Add strPropName, varPropValue
  End With
End Sub
```

The Printer Object and Printer Collection

Microsoft® Access contains a Printer object and a Printers collection that make it possible for you to control printer configuration without using Microsoft® Windows® api calls and the complex structure that PrtDevMode requires. This object makes it possible for you to change printer settings without opening the object in a design model, making custom printer settings in MDE/ADE files possible. You can make these changes even during print preview.

For example, to set printer properties for a Catalog report, you could use the following code:

```
Reports("Catalog").Printer.LeftMargin = 1440  'Set left margin to 1 inch
Reports("Catalog").Printer.Orientation =  acPRORLandscape  'Set orientation to landscape
Reports("Catalog").Printer.PaperSize = acPRPSLegal 'Set paper size to legal
```

To cycle through the installed printers on the system, you can use the following:

```
Dim prt As Access.Printer

For Each prt In Application.Printers
   Debug.Print prt.DeviceName
Next
```

The AddItem and RemoveItem Objects

Microsoft® Access adds the methods AddItem and RemoveItem to the combo box and list box objects. These new methods make it possible for you to add and remove values from a combo box or list box programmatically. For multicolumn combo boxes, you use a semicolon-delimited string to add values to separate columns.

For example, to add values to a two-column combo box, use the format:

```
Combo1.AddItem "ALFKI;Alfred's Futterkiste"
```

The following string removes the first item in a combo box:

```
Combo1.RemoveItem 0
```

To remove an item where the bound column is ALFKI, you can use this format:

```
Combo1.RemoveItem "ALFKI"
```

The Save Model

Microsoft® Access speeds up the process of regularly saving your work. Until you compile your project, Access saves only the modules that you have modified since your last save. This makes it less of a chore to save often during active development. After you compile the project, Access will save the complete project each time. Until then, it is more efficient to save only the parts you have changed.

Working with the Screen Object

Other Microsoft®Office applications have properties that return a reference to active objects. For example, Microsoft® Word has the ActiveDocument property to determine which document currently has the focus. Microsoft® Excel has properties to return the active Workbook, Worksheet, Cell, Chart, and Window objects. Similarly, Microsoft® PowerPoint® has the ActivePresentation property to determine the active presentation.

In Microsoft® Access, you use the Screen object to work with the object or control that currently has the focus. The Screen object has properties that return a reference to the currently active control (on a form or report), data access page, datasheet, form, or report. These properties are useful in code that operates against an object and must know only the type of object. For example, the following line of code hides the currently active form:

```
Screen.ActiveForm.Visible = False
```

The next example shows how you can use the Screen object to determine which cell in a datasheet is selected:

```
MsgBox "The selected item is located at: Row " & Screen.ActiveDatasheet.SelTop & ", Column " _
   & Screen.ActiveDatasheet.SelLeft
```

The Screen object also has properties you can use to work with the previously active control and the mouse pointer.

Note
If you try to refer to an object by using properties of the Screen object and there is no object of that type currently active, an error occurs.

Working with the DoCmd Object

The DoCmd object makes it possible for you to carry out various Microsoft® Access commands by using Microsoft® Visual Basic® for Applications (VBA). These commands are called actions when they are used in Access macros and are called methods of the DoCmd object when they are carried out in code.

Note
In other Microsoft® Office applications, the term "macro" is synonymous with a VBA procedure. In Access, macros are completely different from the VBA code you write in a procedure. For more information about Access macros, search the Microsoft® Access Help index for "macros, overview," and then open the topic "Macros: What they are and how they work."

Two of the most common tasks that require methods of the DoCmd object are opening and closing Access objects. To open an Access object, you use the DoCmd object's OpenObject method, where Object represents the name of the object you want to open. For example, you use the OpenForm method to open a form, the OpenReport method to open a report, and the OpenQuery method to open a query. All of the OpenObject methods take arguments that specify the object to open and how to display the object. For example, the following code opens the Customers form as read-only in Form view (acNormal) and specifies that only customers in the USA be shown:

DoCmd.OpenForm FormName:="Customers", View:=acNormal,
WhereCondition:="Country = 'USA'", DataMode:=acFormReadOnly

You can use the OpenReport method to open a report in Design view or Print Preview, or you can specify that the report be printed, as in the following example:

DoCmd.OpenReport ReportName:="CustomerPhoneList", View:=acViewNormal, WhereCondition:="Country = 'USA'"

Note
When you use the acViewNormal constant in the view argument of the OpenReport method, the report is not displayed but is printed to the default printer.

You use the DoCmd object's Close method to close an Access object. You can use the optional arguments of the Close method to specify the object to close and whether to save any changes. The following example closes the Customers form without saving changes:

DoCmd.Close acForm, "Customers", acSaveNo

Note
All the arguments of the Close method are optional. If you use the method without specifying arguments, the method closes the currently active object.

You can use the DoCmd object's RunCommand method to run commands that appear on an Access menu or toolbar that do not have separate methods exposed in the Access object model. The RunCommand method uses a collection of enumerated constants to represent available menu and toolbar commands. For more information about the RunCommand method, search the Microsoft® Access Visual Basic® Reference Help index for "RunCommand method."

Working with the Modules Collection

The Modules collection contains a Module object representing each module that is currently opened for editing. The Module object might represent a standard or class module that is currently open in the Microsoft® Visual Basic® Editor or a module associated with a form or report that is open in Design view. You can use the methods and properties of a Module object to get information about the code contained in the module or to insert procedures or lines of code. The objects in this collection are typically used by code running in an add-in or wizard.

For more information about the Modules collection and Module objects, search the Microsoft® Access Visual Basic® Reference Help index for "Modules collection" or "Module object."

Working with the References Collection

The References collection contains Reference objects representing each reference in the References dialog box (Tools menu in the Microsoft® Visual Basic® Editor) to another project or object library. A new Microsoft®Access database contains four references by default. You can add or remove references by using the References dialog box or by using methods of the References collection in Microsoft® Visual Basic® for Applications (VBA) code.

For more information about the References collection and Reference objects, search the Microsoft® Access Visual Basic® Reference Help index for "References collection" or "Reference object."

Working with Reports, Forms, and Data Access Pages

Microsoft® Access provides three objects you can use to display data to the user: reports, forms, and data access pages. Although these objects have many similar features, they are used in different ways.

You use reports to display formatted data. The user cannot edit or add data to a report. Reports can be viewed in the database where they were created or printed. You can also save reports as snapshot files so they can be viewed outside an Access application. For more information about working with snapshot files, search the Microsoft® Access Help index for "report snapshots."

You can also use forms to display data to users. However, the real power of forms comes from their ability to collect data from users or let users add new records or edit existing records. Forms can also be printed or saved as reports or data access pages.

Note
Although Microsoft® Access hosts Microsof®Visual Basic®for Applications (VBA) as it does with the other Microsoft® Office applications, it uses its own built-in forms package. UserForms are not available in Access.

Data access pages combine the features of forms and reports so that you can display data to users and let users interact with data through Microsoft® Internet Explorer version 5 or later. (You can also use other Web browsers to display data access pages, but users will not be able to work with the data directly.) Although you design data access pages by using Access, you save them to disk as separate files designed to be used in a Web browser, which means users can work with Access data from within an Access database or over an intranet or the Internet. Data access pages can contain data in an Access database (.mdb file) or Access project (.adp file).

Access forms, reports, and data access pages have numerous properties, methods, and events you can use to specify how the object will look and behave. A complete discussion of all properties, methods, and events is beyond the scope of this section.

For information about a specific property, method, or event, search the Microsoft® Access Visual Basic® Reference Help index for the name of the item about which you want information.

You can use the Application object's CreateForm, CreateReport, and CreateDataAccessPage methods to programmatically create forms, reports, and data access pages. You can also add controls to these objects through VBA code, but unless you are building an add-in or a wizard, you typically create these objects by using the Access user interface and then display them from code. When you display an object, you can use various properties of the object to specify the records it will contain.

Working with PivotViews

Microsoft® Access adds two new views to the set of views that currently exist for tables, queries, stored procedures, and functions. The views are called PivotTable View and PivotChart View, and they make it easy to create flexible reports and to quickly publish your work to the Web using data access pages.

PivotView reports are powerful tools for presenting and analyzing data. They provide a means to view a single set of data in a variety of configurations in a manner similar to the capabilities provided by Microsoft® Excel. As with a query, a PivotView report can answer a question about a data set: Which customers provided the most sales for the first quarter of this year? In which country was a particular product most popular last year? How well did a particular sales representative do in Europe for the past two years?

If you are skilled at building queries, you can answer each of these questions with a separate query. The advantage of the PivotView report, however, is when you have defined the data set, you can "pivot" the data to answer all of these questions with a single data set. For most users, this is easier and more intuitive than building a query, especially when the PivotView report combines data from multiple tables and queries. In addition, because PivotTable data is cached in memory, PivotView reports provide extremely fast querying.

Referring to Open Objects

The Application object has properties that return collections of open Microsoft® Access objects. The Reports property returns a reference to the Reports collection that contains all currently open reports. The Forms property returns a reference to the Forms collection that contains all currently open forms. The DataAccessPages property returns a reference to the DataAccessPages collection that contains all currently open data access pages. You specify a member of a collection by using its name or its index value in the collection. You typically use the index value only when iterating through all the members of a collection because the index value of an item can change as items are added to or removed from the collection. For example, the following sample uses the form's name to reference the Open Customers form:

```
Dim rstCustomers As ADODB.Recordset
Set rstCustomers = Forms("Customers").Recordset
```

The next example closes and saves all open data access pages by looping through the DataAccessPages collection:

```
For intPageCount = DataAccessPages.Count - 1 To 0 Step -1
  DoCmd.Close acDataAccessPage, DataAccessPages(intPageCount).Name, acSaveYes
Next intPageCount
```

The Forms, Reports, and DataAccessPages collections contain only open objects. To determine if an object is open, you can use the IsLoaded property of an item in the AllForms, AllReports, or AllDataAccessPages collections, or you can use the SysCmd method with the acSysCmdGetObjectState constant. You can also use the CurrentView property to determine if a form is open in Design, Form, or Datasheet view or if a data access page is open in Design or Page view. The following procedure uses the SysCmd method and the CurrentView property to determine if a form is open in Form or Datasheet view:

```
Function IsLoaded(ByVal strFormName As String) As Boolean
  ' Returns True if the specified form is open in Form view or Datasheet view.
  Const OBJ_STATE_CLOSED = 0
  Const DESIGN_VIEW = 0

  If SysCmd(acSysCmdGetObjectState, acForm, strFormName) <> OBJ_STATE_CLOSED Then
    If Forms(strFormName).CurrentView <> DESIGN_VIEW Then
      IsLoaded = True
    End If
  End If
End Function
```

The Data Behind Forms and Reports

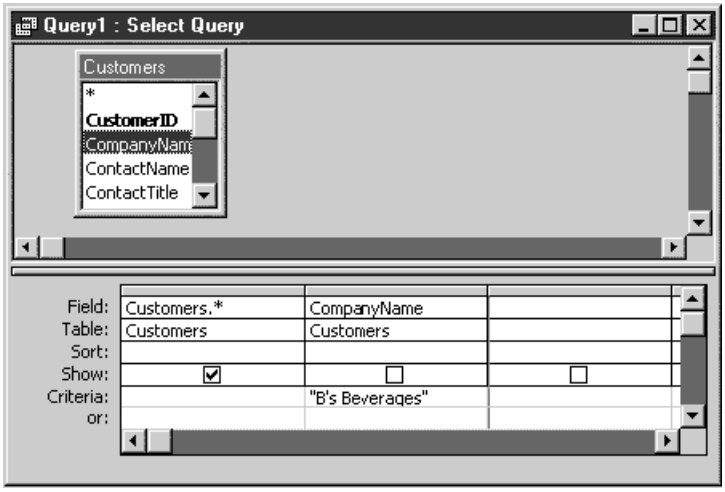
Most of the forms you create will be designed to display or collect data. Forms can display data for viewing, editing, or input. Forms are also used to create dialog boxes that collect information from a user, but do not display data. Reports display static data only, and aren't used to edit or collect data.

The source of the data behind a form or report is specified by the object's RecordSource property. The RecordSource property can be a table, a query, or a Structured Query Language (SQL) statement. You can display subsets of the data contained in the object's RecordSource property by using the Filter property to filter the data or by using the wherecondition argument of the OpenForm or OpenReport method to specify a subset of data. When you have specified a record source for a form or report, you can use the field list (in form or report Design view) to drag fields from the object's source of data to the object.

If you set the RecordSource property by using Microsoft® Visual Basic® for Applications (VBA), you can use the name of an existing table or query, or a SQL statement. The easiest way to create a SQL statement to use in code, whether from within a Microsoft® Access module or another Microsoft(Office application, is to use the Access query design grid to create a query that displays the appropriate records. When the query contains the records you want, click SQL View on the View menu and copy the SQL string that defines your query. You can then paste the SQL string into your VBA code and replace any hard-coded criteria with variables that will contain the data you want to use as criteria.

The following figure shows a query created in the query design grid that selects all fields from the Customers table for the customer named B's Beverages.

Specifying Criteria in the Query Design Grid



The SQL view for this query contains the following SQL statement:

*SELECT * FROM Customers WHERE CompanyName = "B's Beverages";*

You can modify this SQL statement for use in the following VBA procedure so that it will display a single customer record for any company passed in the strCompanyName variable:

```
Option Explicit
Dim frmTempForm As Form

Sub ShowCustomerRecord(strCompanyName As String)
  Dim strSQL As String

  strSQL = "SELECT * FROM Customers WHERE CompanyName = "  & """" & strCompanyName & """"
  Set frmTempForm = New Form_Customers
  With frmTempForm
    .RecordSource = strSQL
    .Visible = True
  End With
End Sub
```

Secifying String Criteria by Using Variables in Code

When you specify criteria for a query, filter, or wherecondition argument from code, you typically use a variable. For example, you could specify the wherecondition argument of the OpenReport method as in the following:

```
DoCmd.OpenReport ReportName:="CustomerPhoneList", _
        WhereCondition:="CompanyName = " & "'" & strCompanyName & "'"
```

When the criteria used is a string, the variable can be surrounded with single quotation marks ('). However, if the value of the variable contains a single quotation mark, this technique will not work. For example, if you are searching for records that match the criteria "CompanyName = 'B's Beverages'", you will encounter errors. If there is any chance that a variable will contain a value that itself contains a single quotation mark, you should surround the variable with two sets of double quotation marks ("), as shown in the following example:

```
DoCmd.OpenReport ReportName:="CustomerPhoneList", _
        WhereCondition:="CompanyName = " & """" & strCompanyName & """"
```

For more information about using quotation marks in strings, search the Microsoft® Access Visual Basic® Reference Help index for "quotation marks," and then open the topic "Quotation Marks in Strings."

When you are working with forms, you can also use the new Recordset property to specify the Recordset object that contains the records of the form or the subform. The following example illustrates how to change the source of data for a currently open form:

```
Sub ChangeRecordsetProperty()
  Dim frmNewRecords      As Form
  Dim rstNewRecordset   As New ADODB.Recordset

  Call ShowCustomerRecord("B's Beverages")
  Stop ' View Customers form containing 1 record.
  Set frmNewRecords = Forms(Forms.Count - 1)
  rstNewRecordset.Open "SELECT * FROM Customers",  CurrentProject.Connection, adOpenKeyset, adLockOptimistic
  Set frmNewRecords.Recordset = rstNewRecordset
  Stop ' View Customers form containing 91 records.
End Sub
```

The Recordset property of forms is new in Access. You use the Recordset property to specify or determine the Recordset object representing a form's source of data. The recordset represented by the Recordset property is a read-only recordset. If you must programmatically work with the data contained in the records displayed in a form, you must to use the Data Access Object (DAO) RecordsetClone property or the ActiveX Data Objects (ADO) Clone method to create a second recordset that you can manipulate with VBA code. The Recordset property can be accessed only by using VBA code and can be used to bind multiple forms to a single recordset or to synchronize multiple forms or multiple Recordset objects. When you change a form's Recordset property, you must use the Set statement, as illustrated in the preceding code sample.

Note
Changing a form's Recordset property might also change the RecordSource, RecordsetType, and RecordLocks properties. In addition, other data-related properties also might be overridden, for example, the Filter, FilterOn, OrderBy, and OrderByOn properties might all be affected when you change the Recordset property of a form.

Working with Controls on Forms and Reports

Although forms, reports, and data access pages are the objects you use to present or gather data from users, it is really the controls on these objects that do all the work. Access contains a wide variety of built-in controls that you can use on these objects.

Forms, reports, and data access pages all use controls to display information or to make it possible for the user to interact with the object or the data it contains. Forms and reports have a Controls property that returns a collection of all the controls on the object.

Cntrols Collections for Forms and Reports

You can refer to a control on a form or report as a member of the Controls collection or by using the name of the control itself. For example, the following lines of code illustrate three ways to return the RowSource property setting for a combo box control on a form. Because the Controls property is the default property of a Form object, you can refer to the control's name without explicitly specifying the Controls property, as shown in the second and third examples that follow:

```
strSource = Forms("SalesTotals").Controls("cboSelectSalesPerson").RowSource
strSource = Forms("SalesTotals")!cboSelectSalesPerson.RowSource
strSource = Forms!SalesTotals!cboSelectSalesPerson.RowSource
```

Note
The ! operator is used to refer to user-defined items, such as forms, reports, and controls on Access forms or reports.

You can also use the Controls property to work with all the controls on a form or report. For example, the following code loops through all the controls on a form and sets the Text property for each text box control to a zero-length string (""):

```
Sub ClearText(frmCurrent As Form)
  Dim ctlCurrent As Control

  For Each ctlCurrent In frmCurrent.Controls
    If ctlCurrent.ControlType = acTextBox Then
      ctlCurrent.Value = ""
    End If
  Next ctlCurrent
End Sub
```

You can pass the Form object to the ClearText procedure by using the Me property. The Me property returns an object representing the form, report, or class module where code is currently running. For example, you could call the ClearText procedure from a form by using the following syntax:

```
Call ClearText(Me)
```

Certain controls on forms and reports also have a Controls collection. For example, the option group control might contain a Controls collection representing option button, toggle button, check box, or label controls in the option group. The tab control has a Pages collection containing a Page object for each page in the tab control. Each Page object also has a Controls collection representing all the controls on a page in a tab control.

Subform and Subreport Controls

Forms and reports can also contain subform or subreport controls that contain another form or report. These controls make it possible for you to display related records from another form or report within a main form or report. A common example of this is a Customers form that contains a subform containing customer orders. You use the SourceObject property of the subform or subreport control to specify the form or report that will be displayed in the control.

The form or report in the subform or subreport control can share a common field, known as the linking field, with the records displayed in the main form or report. The linking field is used to synchronize the records between the subform or subreport and the main form or report. For example, if the record sources for an Orders subform and a Customers main form both contain a CustomerID field, this would be the common field that links the two forms. To specify the linking field, you use the LinkChildFields property of the subform or subreport control and LinkMasterFields property of the main form or report. However, the easiest way to create a linked subform or subreport is to open the main form or report in Design view, drag the appropriate form or report from the Database window to the main form or report, and then release the mouse button.

You use the Form property of a subform control to refer to controls on a subform. You use the Report property of a subreport control to refer to controls on a subreport. The following examples illustrate how to get the value of a control on a subform or subreport by using VBA. The first two lines show alternative ways to reference a control named Quantity on a subform. The last line shows how to use the RecordCount property to get the number of records contained in the recordset associated with a subreport control:

lngOrderQuantity = Forms("CustomerOrders").Controls("SubForm1").Form!Quantity
lngOrderQuantity = Forms!CustomerOrders!SubForm1.Form!Quantity
lngNumProducts = Reports!SuppliersAndProducts!SubReport1.Form.Recordset.RecordCount

List Box and Combo Box Controls on Forms

List box and combo box controls are very powerful and versatile tools for displaying information and making it possible for the user to interact with the data displayed on a form. These controls work differently in Access than list box and combo box controls in other Office applications, and it is important to understand these differences if you want to use these controls effectively.

If you are used to working with these controls in other applications, the most important difference is how you add items to and remove items from these controls. In other applications, these controls have AddItem and RemoveItem methods to add and remove items. These methods are not supported for Access list and combo box controls. Instead, you use combinations of RowSource and RowSourceType properties to specify the data that appears in a list box or combo box control. The relationship between the RowSource property setting and the RowSourceType property setting is illustrated in the following table.

RowSourceType property setting	RowSource property setting
Table/Query	Table name, query name, or SQL statement
Value List	Semicolon-delimited list of values
Field List	List of field names from a table, query, or SQL statement
User-defined function	No value specified

For more information about setting the RowSourceType and RowSource properties to fill a list box or combo box control, search the Microsoft® Access Visual Basic® Reference Help index for "RowSource property" or "RowSourceType property."

If you are creating list box or combo box controls through the Access user interface, you can take advantage of the List Box Wizard and the Combo Box Wizard to set the various properties required to display data in these controls. To use these wizards, make sure the Control Wizards tool in the toolbox is pressed in, then click the List Box or Combo Box tool in the toolbox, and then click the place on the form where you want the control to appear. Follow the instructions displayed by the wizard.

You can set the properties of a list box or combo box control without using the wizard by using the control's property sheet or VBA. You use the ControlSource property to bind a list box or combo box control to a field in the recordset specified in the form's RecordSource property. As mentioned earlier, you use the RowSource property in combination with the RowSourceType property to specify the source of data for the list box or combo box control.

The BoundColumn property specifies which column in the record source specified by the RowSource property will contain the value of the list box or combo box control. If a list box or combo box control does not have a ControlSource property setting, you can set the BoundColumn property to 0. When you do this, the Value property of the control will contain the row number of the selected row specified by the RowSource property. The row number of the selected row is the same as the value of the control's ListIndex property. The ColumnCount and ColumnWidths properties specify which columns are displayed in the control.

The following sample fills a combo box control with data from an SQL statement, specifies which column in the SQL statement specified by the RowSource property will contain the value for the control, and uses the ColumnWidths property to specify which columns are displayed in the control:

```
With Me!cboEmployees
  .RowSource = "SELECT EmployeeID, FirstName, "  & "LastName FROM Employees ORDER BY LastName"
  .RowSourceType = "Table/Query"
  .BoundColumn = 1
  .ColumnCount = 3
  .ColumnWidths = "0in;.5in;.5in"
  .ColumnHeads = False
  .ListRows = 5
End With
```

The preceding code fills a combo box with data from 3 fields (columns) from each record (row) in the Employees table, as specified by the RowSource property. The BoundColumn property is set to the first field in the Employees table, in this case, EmployeeID. When an item is selected from the combo box, the value of the EmployeeID field will be the control's value and is the value saved to the field specified by the ControlSource property. Note that the first column in the ColumnWidths property is set to 0 inches. This hides the bound column (EmployeeID) from the user when the combo box's drop-down list is displayed. The user sees only the FirstName and LastName fields, and these fields are displayed in .5-inch wide columns. Note also that the ColumnHeads property is set to False, meaning that the names of the FirstName and LastName fields are not shown in the control's drop-down list. And finally, the ListRows property is set to 5, specifying that the control's drop-down list will display only 5 records at a time.

Using a User-Defined Function to Fill a List Box or Combo Box Control

You can specify a user-defined function as the RowSourceType property setting for a list box or combo box control. The function you use for this property setting has to meet specific criteria in order to work correctly because the function is called repeatedly as Access fills the control with data. For more information about creating and using user-defined functions to fill a list box or combo box control, search the Microsoft® Access Visual Basi®Reference Help index for "RowSourceType property," then open the "RowSourceType, RowSource Properties" topic, and then use the See Also jump to open the "RowSourceType Property (User-Defined Function)-Code Argument Values" topic.

Adding New Values to a Combo Box Control

You use the LimitToList property to specify whether a user can add new values to a bound combo box from the user interface when the form is in Form view or Datasheet view. When this property is set to True (the default), the user can't add new items to the combo box. If the BoundColumn property is set to any column other than 1, Access will automatically set the LimitToList property to True. When this property is set to False, new values are added to the underlying record source specified by the RowSource property.

When the LimitToList property is set to True, any attempt to add a new item to a combo box control will cause the NotInList event to occur. You can add code to the NotInList event procedure to handle the attempt to add new data to the control. This event procedure uses the NewData and Response arguments to represent the new data the user has tried to enter and the response you want to provide in the attempt to add new data. Setting the Response argument to one of the following built-in constants specifies how you want to respond to the attempt to add data to the control: acDataErrAdded, acDataErrContinue, or acDataErrDisplay. For example, the following sample illustrates one way to add new data to a combo box control:

```
Private Sub CategoryID_NotInList(NewData As String, Response As Integer)
  If MsgBox("Do you want to add '" & NewData & "' to the items in this control?", _
                vbOKCancel, "Add New Item?") = vbOK Then
    ' Remove new data from combo box so control can be requeried  after the AddNewData form is closed.
    DoCmd.RunCommand acCmdUndo
    ' Display form to collect data needed for the new record.
    DoCmd.OpenForm "AddNewData", acNormal, , , acAdd, acDialog, NewData
    ' Continue without displaying default error message.
    Response = acDataErrAdded
  Else
    Response = acDataErrContinue
  End If
End Sub
```

For more information about how to use the NotInList event procedure, search the Microsoft®Access Visual Basic®Reference Help index for "NotInList event."

Enabling Multiple Selections in a List Box Control

To make it possible for users to make multiple selections from a list box control, you set the MultiSelect property. When the MultiSelect property is set to Simple (2) or Extended (1), the Value property of the control is Null. You work with multiple selections in a list box control by using the Selected, ItemsSelected, and Column properties.

Working with Data Access Pages

Data access pages are HTML documents comprised of HTML code, HTML intrinsic controls, and Microsoft® ActiveX®controls. Data access pages rely on DHTML and are designed to work best with Microsoft® Internet Explorer version 5 or later. (You can also use other Web browsers to display data access pages, but users will not be able to work with the data directly.)

A data access page can be a simple HTML document or can include data-bound controls that let users use a Web browser to interact with data stored in a database. Microsoft® Access provides a WYSIWYG design environment for creating data access pages and a means for deploying those pages and any necessary supporting files to a Web server, network server, or local file system. In addition, you can view and use data access pages within Access itself.

You might be tempted to think of data access pages as HTML documents that combine the best features of forms and reports for display on the Web, but that would be a very narrow definition. Data access pages do support much of the functionality you are used to in Access forms and reports, but they also provide a completely new way to interact with data from within an Access database or on the Web. These objects make it possible for users to use a Web browser to work with data in an interactive manner and in a way that has never been possible before. Data access pages are similar to forms in that you can use them to view, edit, or delete existing records, and you can also use them to add new records to an underlying record source. They are similar to reports in that you can sort and filter records as well as group records according to criteria you specify. In addition, while a page is displayed, you can manipulate the records that are displayed and change how the records are displayed.

You can create data access pages from scratch in Access or you can base them on existing HTML pages created by using some other HTML authoring tool. Only those pages created or modified within the Access design environment will be visible in the Pages object list in the Database window. This means that if you edit an HTML document in Access, a link to that document is created, even if you later use another tool to make additional changes. In addition, because the data access pages that appear in the Database window are links to the files stored on disk, you can delete a page in the Database window without deleting the file from disk. Unlike other objects in an Access database, data access pages are stored on disk as .htm files that are separate from the Access database in which they are created.

Creating a data access page in Access is similar to creating a form or report. Data access pages have their own object list in the Database window, and when they are opened in Design view, they have a toolbox and property sheet. The toolbox contains tools for inserting the HTML intrinsic controls, such as the text box, label, list box, and command button controls. In addition, the toolbox contains tools for inserting controls that are useful only on data access pages, such as expand, bound HTML, and scrolling text controls. The toolbox also contains tools for inserting the Microsoft® Office Web Component controls on a data access page.

Note
Because Access does not use the shared Microsoft® Office components related to Script objects or HTMLProject objects, you can't use these objects to work with scripts or the HTML code in data access pages through VBA code. To work with scripts or the HTML code in an Access DataAccessPage object, you use the Microsoft® Script Editor or a DataAccessPage object's Document property, which returns the Web browser's document object for an HTML page.

Creating, Saving, and Closing Data Access Pages

You will typically create data access pages in the data access page design environment in Access. However, there might be circumstances where you want to use VBA code to display a data access page within Access or to programmatically output a page to a separate location, such as a Web server on your local intranet.

You create a data access page programmatically by using the Application object's CreateDataAccessPage method. You can use the CreateDataAccessPage method to work with an existing HTML page as a data access page or to create a new, blank page. For example, the following code illustrates how to use this method to create a new page called BlankDAP.htm:

```
Application.CreateDataAccessPage FileName:="c:\WebPages\BlankDAP.htm" CreateNewFileName:=True
```

The CreateDataAccessPage method creates a new blank page by default and adds a link to that page in the Pages object list in the Database window. If the file specified in the FileName argument already exists when the method is called and the CreateNewFileName argument is set to True (the default), an error occurs. If you set the method's CreateNewFileName argument to False, the FileName argument must contain the path and name of an existing file. If the file does not exist, an error occurs. If the FileName argument contains the path and file name of a file for which there is already a link in the Pages object list in the Database window, a new, uniquely named link is created that points to the same file on disk. If you provide a name but do not specify the path to a new file, the page is created in the current directory.

You can determine the path and file name for pages that appear in the Pages object list in the Database window by using the read-only FullName property of the AccessObject object that represents a particular data access page. For example, the following code prints the Name and FullName properties for each page in the current database:

```
Dim objDAP As AccessObject

For Each objDAP In CurrentProject.AllDataAccessPages
Debug.Print "The '" & objDAP.Name & "' is located at: " & objDAP.FullName
Next objDAP
```

When you call the CreateDataAccessPage method, Access creates a temporary file on disk. To permanently save the page and create a pointer to it from the Pages object list in the Database window, you must use the Save method or the Close method of the DoCmd object.

The following code fragment illustrates how you could create a new data access page, work with the HTML in the page, and then create a link to the page and permanently save it to disk. The procedure also illustrates one way to use an error trap to handle files that already exist.

```
Function CreateDAP(strFileName As String) As Boolean
  ' This procedure illustrates how to create a data access  page, work with the HTML in the page, and then save the page.
  ' The procedure also shows how to use an error trap to avoid  the error that ocurrs if strFileName already exists.
  Dim dapNewPage As DataAccessPage
  Const DAP_EXISTS As Long = 2023

  On Error GoTo CreateDAP_Err
  ' Create the new page.
  Set dapNewPage = Application.CreateDataAccessPage(strFileName, True)
  ' Use the Document property to return the Internet Explorer 5  document object, and then use the objects on the page to
  ' work with the HTML in the page.
  With dapNewPage.Document
    .All("HeadingText").innerText = "This page was created programmatically!"
    .All("HeadingText").Style.display = ""
    .All("BeforeBodyText").innerText = "When you work " & "with the HTML in a data access page, you " _
      & "must use the document property of the page " & "to get to the HTML. "
    .All("BeforeBodyText").Style.display = ""
  End With
  ' Close the page and save all changes.
  DoCmd.Close acDataAccessPage, dapNewPage.Name, acSaveYes
  CreateDAP = True

CreateDAP_End:
  Exit Function
CreateDAP_Err:
  If Err = DAP_EXISTS Then ' The file specified in strFileName already exists,  so replace it with this new page.
     If MsgBox("'" & strFileName & "' already exists. Do you want to " & "replace it with a new, blank page?", vbYesNo, _
      "Replace existing page?") = vbYes Then
      Set dapNewPage = Application.CreateDataAccessPage(strFileName, False)
      Resume Next
    Else
      CreateDAP = False
      Resume CreateDAP_End
    End If
  Else
    CreateDAP = False
    Resume CreateDAP_End
  End If
End Function
```

Note
When you create a new data access page, the display property of the style object for the HeadingText and BeforeBodyText elements is set to None by default. The preceding example also illustrates how to change this setting so the text you insert is visible when the page is viewed.

The CreateDAP procedure uses the data access page's Document property to return the Internet Explorer 5 document object and then sets properties of elements in the page. This procedure also uses the innerText property of an HTML element to specify the text that appears in the element.

Opening and Working with Data Access Pages

Although data access pages are designed to be viewed in a browser, you can display data access pages in Access to let users view and work with data as they do with forms and reports.

To open an existing data access page for which a link exists in the Pages object list in the Database window, you use the DoCmd object's OpenDataAccessPage method. You use the View argument of the OpenDataAccessPage method to specify whether to view the page in Design view or Page view. The following example illustrates how to open the Employees page in Page view:

```
DoCmd.OpenDataAccessPage "Employees", acDataAccessPageBrowse
```

To determine whether a page is currently open in Page view or Design view, you use a DataAccessPage object's CurrentView property.

The DataAccessPages collection contains all currently open data access pages. You can access an open page as a member of this collection and gain access to the properties and methods of the page itself as well as any controls on the page. The following sample code opens the Employees page in Design view, applies a theme, adds some text to the main heading, and then displays the page to the user:

```
With DoCmd
  .Echo False
  .OpenDataAccessPage "Employees", acDataAccessPageDesign
  With DataAccessPages("Employees")
    .ApplyTheme "Blends"
    .Document.All("HeadingText").innerText = "Today is "& Format(Date, "mmmm d, yyyy"
  End With
  .OpenDataAccessPage "Employees", acDataAccessPageBrowse
  .Echo True
End With
```

Note

To save the changes you make to a data access page, you must make sure the page is in Design view, and then use the DoCmd object's Save method to save changes. If you programmatically make changes to a page while it is in Page view, those changes will be lost as soon as you call the Save method.

To get information about the pages in your database, including whether a page is currently open, you use the CurrentProject object's AllDataAccessPages collection. To specify or determine property settings for a page or controls on a page, you use the properties of a DataAccessPage object. The following sample uses both techniques to print information about data access pages to the Immediate window:

```
Sub DAPGetPageInfo()
  ' This procedure prints information about the data access pages  in this database to the Immediate window.
  Dim objCurrentDAP       As AccessObject
  Dim strPageInfo         As String
  Const DAP_DESIGNVIEW    As Integer = 0
  Const DAP_PAGEVIEW      As Integer = 1

  Debug.Print "There are "; CurrentProject.AllDataAccessPages.Count  & " data access pages in this database."
 For Each objCurrentDAP In CurrentProject.AllDataAccessPages
    Debug.Print objCurrentDAP.Name & ":"
    Debug.Print vbTab & "File name: " & objCurrentDAP.FullName
    If objCurrentDAP.IsLoaded <> True Then
      Debug.Print vbTab & "The '" & objCurrentDAP.Name  & "' page is not currently open."
    Else
      Select Case DataAccessPages(objCurrentDAP.Name).CurrentView
        Case DAP_DESIGNVIEW
          Debug.Print vbTab & "The '" & objCurrentDAP.Name  & "' page is open in Design view."
        Case DAP_PAGEVIEW
          Debug.Print vbTab & "The '" & objCurrentDAP.Name & "' page is open in Page view."
      End Select
    End If
  Next objCurrentDAP
End Sub
```

Note that the DataAccessPages collection contains DataAccessPage objects, whereas the AllDataAccessPages collection contains AccessObject objects.

When you work with data access pages inside Access, you can use VBA code in a form, for example, to specify or determine property settings of the page or controls on a page. In the next example, the SimplePageExample page is opened and the DataEntry property of the page's Data Source control is set to True so the page can be used only to enter new records:

```
Private Sub cmdSimpleDAPDataEntry_Click()
  With DoCmd
    .Echo False
    .OpenDataAccessPage "SimplePageExample", acDataAccessPageBrowse
    DataAccessPages("SimplePageExample").Document.All("MSODSC").DataEntry = True
    .Echo True
  End With
End Sub
```

Note

The ActiveX control that binds controls on a page to an underlying data source is the Microsoft® Office Data Source control (MSODSC). This control is included in every data access page you create but is not visible on the page itself. In the preceding example, the DataEntry property of the MSODSC is set to True. As you can see, the control is created by using an id property

setting of "MSODSC", and you use this id property to specify that the control is a member of the all collection in the data access page's Document object.

Caution
Although the HTML underlying the MSODSC is available, you should never modify the HTML directly either in Access or in any HTML authoring tool. To set properties of the MSODSC, you must use its property sheet in the Microsoft® Script Editor or use Microsoft® Visual Basic®Scripting Edition (VBScript) code in the data access page itself.

To open the SimplePageExample page so that it displays all records, you would use the following code:

DoCmd.OpenDataAccessPage "SimplePageExample", acDataAccessPageBrowse

You can also use the Microsoft® Script Editor to add to a page script that runs when the page is displayed or in response to events that occur on the page. The script you add to a page is part of the page itself and can run when the page is displayed in Access or in a Web browser.

Using the Microsoft Script Editor with Data Access Pages

The Microsoft® Script Editor is an editor and debugger that you can use to work with the HTML code and script in a data access page. This section describes how to use the Script Editor with data access pages.

When you view a data access page in the Script Editor, you see color-coded HTML code and script in the page. In addition, depending on the controls you have placed on the page, you might also see icons representing some controls. For example, the Data Source control is displayed as an icon. You can see the HTML and XML code underlying a control's icon by right-clicking the icon in the Script Editor and clicking Always View As Text on the shortcut menu.

When you create a new data access page, the page contains a two-dimensional section, represented by a <DIV> tag in the HTML code that uses the CLASS attribute MSOShowDesignGrid and a default ID attribute of SectionUnbound. When you add data-bound controls to this section of the page, Access automatically changes the ID attribute to reflect the controls you are using. For example, if you drag the Customers table to this section, Access changes the ID attribute to HeaderCustomers. You can place controls anywhere within the two-dimensional section as you can on a form or report. Outside of this section, controls cannot be positioned in this manner.

When you create event procedures, the Script Editor does not insert the event procedure arguments when they are required. You must insert these yourself. For example, every event associated with the Data Source control requires a single dscEventInfo argument. If you double-click the Current event for the Data Source control (MSODSC) in the Script Editor's Script Outline window, the following script block is inserted in your page:

```
<SCRIPT Language=vbscript FOR=MSODSC EVENT=Current>
<!--  -->
</SCRIPT>
```

You must add the event's argument or arguments by adding parentheses and a name for the argument or arguments. It does not matter what name you use for each argument, and it does not matter if the argument is actually used in your script. You must supply all the arguments to the event, even if your code does not use them, or the code will not work. For example, here is the corrected event handler for the Data Source control's Current event:

```
<SCRIPT Language=vbscript FOR=MSODSC EVENT=Current(EventInfo)>
<!--  -->
</SCRIPT>
```

Security Considerations for Data Access Pages

Because data access pages are designed to work both within and outside Access databases, security issues pertaining to data access pages require special attention. Understanding these issues requires an understanding of database security as well as Internet Explorer security.

Access and XML

Moving information between applications and across the Web often has been difficult because of differences in data formats and proprietary structures. With Extensible Markup Language (XML), however, data, metadata, and presentation information can be moved, translated, and stored without difficulty. XML is a transport format, however, and does not make it possible for you to combine information from multiple files and query the result or create merged subsets easily.

Microsoft® Access provides the means for you to import data formatted in XML, and makes it possible for you, the developer, and the users of Access, either to load pre-existing tables with information contained within an XML file or to create the tables on Import. Then, you can use the many other features of Access to manipulate and query your data.

Access also provides an export mechanism for sharing data stored in Microsoft® Jet, linked through Microsoft® SQL Server™ and referenced by views or queries. Adding a TransferXML method to support XML export and import makes it possible for Access to expand its information sharing capabilities.

Working with Microsoft Data Analyzer

Microsoft® Data Analyzer, part of the Business Intelligence offering from Office, is an easy-to-use tool that makes sophisticated business intelligence accessible to everyone, regardless of technical expertise. Developers can customize Data Analyzer by using the documented application programming interface (API), as well as automate Data Analyzer features with Microsoft Visual Basic®, Microsoft Visual Basic for Applications (VBA), Microsoft Visual Basic Scripting Edition, and Microsoft Visual C++®. Data Analyzer can also be hosted on Web pages, digital dashboards, or forms. This Web site will provide you with technical content on Data Analyzer and related Business Intelligence technologies like OLAP or the Office Web Components.

In This Section

Exploring Microsoft Data Analyzer Programmability
This article briefly explores the programmability features of the Microsoft Data Analyzer.

Max3API Library VBA Code Sample Reference for Microsoft Data Analyzer
Find VBA code samples for many of the objects and members of the Max3API library that is included with Microsoft Data Analyzer.

MdhInterfacesLib Library VBA Code Sample Reference for Microsoft Data Analyzer
Provides VBA code samples for many of the objects and members in the Microsoft Data Analyzer MdhInterfacesLib library.

Using the Microsoft Data Analyzer ActiveX Control in Microsoft Excel 2002
Learn how to embed and use the Microsoft Data Analyzer ActiveX control in Microsoft Excel 2002 to spot trends, opportunities, and potential problems with online analytical processing (OLAP) data.

Using the Microsoft Data Analyzer ActiveX Control in Web Pages
This article describes how to embed the Microsoft Data Analyzer ActiveX Control in Web pages hosted by Microsoft FrontPage, a Web site based on SharePoint Team Services by Microsoft, a SQL Server Digital Dashboard 3.0 dashboard, or Microsoft SharePoint Portal Server.

Understanding the Data Analyzer Application Object

Data Analyzer provides a rich infrastructure for accessing and modifying dimensional data. All the facilities provided in the second programmability layer are extensible using a collection of COM components bound together via a clever use of XML. The treasure house of the programmatic power in the application is the Max 3.0 API. The focus of the API is opening and managing views with the application. Thus, the two principal objects of interest to developers usually will be the **Application** object and **View** object. You can see the organization of some of the objects in the API in FIGURE 2. Keep in mind that the API objects do not always work as you might expect. For example, the **Qualities** object is not a collection that returns a set of **Quality** objects the way one might expect in a lot of libraries, such as a **Workbooks** collection that lets you do a **For...Each** loop to get all of the workbooks. Hence, you cannot iterate through a **Qualities** collection, get a **Quality** object, and then get some properties of that **Quality**. These and other little anomalies aside, the main objects are fairly easy to figure out.

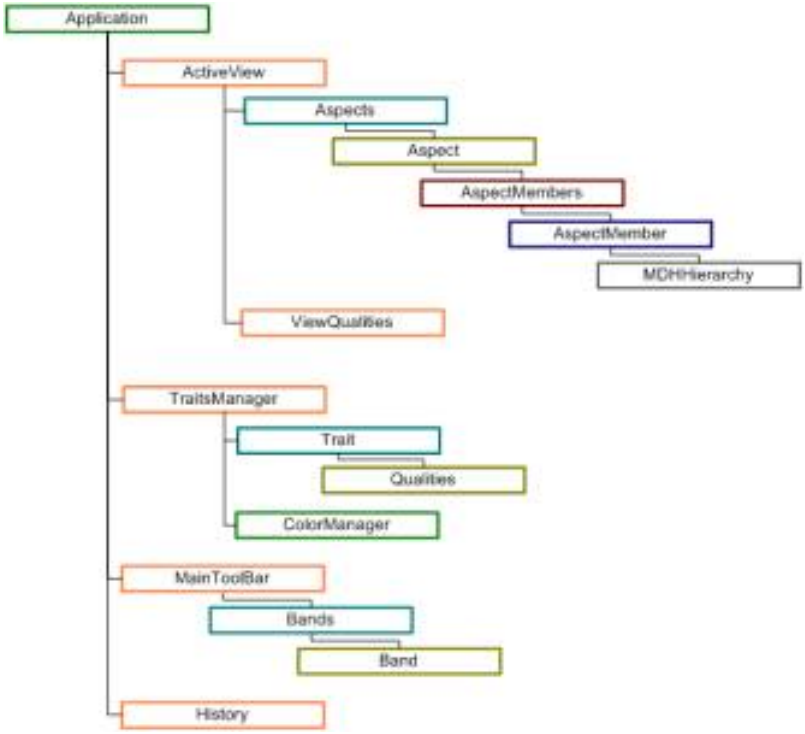


Figure 1: Main objects in the Data Analyzer API.

The **Application** object represents the Data Analyzer application. You can open views, show dialog boxes to export reports, and exit the application. The **View** object is the real workhorse of your programmatic solution because it is the focus of nearly all user interaction. The **View** object represents the entry into the dimensional data, the reports and so forth. You can open a view using the **OpenView** method and specifying the path to a max file. If you are really adventurous, you can build a view from scratch. The following code opens a view to the sample cube that ships with Data Analyzer:

```
Private Sub Form_Load()
  Dim daApp As Max3API.Application
  Set daApp = MSDA1.Application()
 daApp.ActiveView.OpenView "C:\Airline.Max", _
   vlocFileSystem
End Sub
```

What this code does is load the view definition into an xml file. Then, the Data Analyzer application object uses the **OpenView** method.

Exploring Microsoft Data Analyzer Programmability

The Microsoft® Data Analyzer is a software application, part of the Microsoft Office family of products, which enables you to graphically mine your organization's multidimensional data. Microsoft Data Analyzer is designed to work in conjunction with online analytical processing (OLAP) data based on Microsoft SQL Server™ 2000 with Analysis Services (currently, the only supported database is MSOLAP). For more information, visit the Microsoft Data Analyzer Web site.

Setting Up Microsoft Data Analyzer

To use Microsoft Data Analyzer, you need to connect to an OLAP database. For this column, I am using SQL Server 2000 with Analysis Services and the sample Food Mart 2000 database.

You can create a new connection to an existing database by clicking **New** on the **File** menu. In the **Define View—Connections** dialog box, use the **Add** or **Edit** buttons to create or edit a database connection.

If you click the **Add** button, you will need to provide a name for the connection, as well as the name of the OLAP database server, a local OLAP .cub file, or an HTTP URL containing the OLAP data. Once you have entered this information, click the **Connect** button. You will then need to pick the data catalog and the OLAP cube in the database.

For the Food Mart 2000 database, I will select the Sales OLAP cube.

Programming the Microsoft Data Analyzer ActiveX Control

In addition to the standalone user interface, Data Analyzer supplies as an ActiveX® control. This enables you to programmatically control Data Analyzer from COM-based development applications such as Microsoft Visual Basic®. For example, you could embed the Microsoft Data Analyzer ActiveX control on a UserForm and manipulate it with Visual Basic for Applications (VBA) code, embed the Microsoft Data Analyzer ActiveX control on a Visual Basic form and manipulate it with Visual Basic code, or embed the Microsoft Data Analyzer ActiveX control on a Web page and manipulate it with Microsoft Visual Basic Scripting Edition (VBScript) code.

Some of the features you can programmatically control with the Microsoft Data Analyzer ActiveX control are:

- Creating, loading, configuring, and saving database views.
- Changing properties such as colors, visualization methods, dialog boxes, analysis algorithms, and built-in functions.
- Running menu items.

Using the Microsoft Data Analyzer ActiveX Control in a UserForm

You can embed the Microsoft Data Analyzer ActiveX control in a UserForm and manipulate it with VBA code. To do so, on a computer with Data Analyzer installed, add a UserForm to a VBA project, then in the **Toolbox** (**View** menu), right-click and select **Additional Controls**. In the **Available Controls** dialog box, check the **Max3Ax Class** box, and then click **OK**. Finally, drag the Max3Ax Class control onto the UserForm. At design time, the Microsoft Data Analyzer ActiveX control's user interface will not be visible. At run time, the Microsoft Data Analyzer ActiveX control's user interface is visible and runs within the confines of the UserForm.

To program against the Microsoft Data Analyzer ActiveX control, you must also set a reference (**References** dialog box, **Tools** menu) to the **Max3API** DLL (in a default installation, this DLL can be found at C:\Program Files\Microsoft Data Analyzer\Data Analyzer 3.5\Max3API.dll).

Using the Microsoft Data Analyzer ActiveX Control in a Web Page

You can also include the Microsoft Data Analyzer user interface on a Web page. To do so in Microsoft FrontPage® 2002 with Data Analyzer installed:

- On the **Insert** menu, click **Web Component**.
- In the **Component Type** list, select Advanced Controls, and in the **Choose an effect** list, click **ActiveX Control**, and then click **Next**.
- In the **Choose a control** list, if **Max3Ax Class** is not available, click **Customize** and check the **Max3Ax Class** box, then click **OK**. After selecting the **Max3Ax Class** entry in the **Choose a control** list, click **Finish**.

Alternatively, if you are not using FrontPage to create your Web page, you can manually type the following HTML code in a Web page:

```
<object classid="clsid:E0ECA9C3-D669-4EF4-8231-00724ED9288F"  width="100%" height="100%" id="Max3Ax1"/>
```

The Microsoft Data Analyzer Object Model

The following tables includes a description of the objects and collections in the Microsoft Data Analyzer object model:

Object/collection	Description	Parent objects	Child objects/collections
Application	Provides programmatic access to the Microsoft Data	(None)	**View**, **History**, **Toolbar**

	Analyzer ActiveX Control.		
Aspect	Represents an aspect of a database view.	**Aspects**	**AspectMembers**
AspectMember	Represents an aspect member of a database view.	**AspectMembers**	(None)
AspectMembers	Represents all of the aspect members of a database view.	**Aspect**	**AspectMember**
Aspects	Represents all of the aspects of a database view.	**View**	**Aspect**
Band	Represents a toolbar band.	**Bands**	(None)
Bands	Represents a collection of toolbar bands.	**Toolbar**	**Band**
ColorManager	Represents color settings.	**TraitsManager**	(None)
History	Represents the history manager.	**Application**	(None)
Qualities	Represents a collection of qualities.	**Trait**	(None)
Toolbar	Represents a toolbar.	**Application**	**Bands**
Trait	Represents a single trait (like color or quantity) with a single quality, or a collection of traits (like a grid) with many qualities.	**TraitsManager**	**Qualities**
TraitsManager	Represents all of the traits of the active database view.	**View**	**ColorManager**, **Trait**
View	Represents a database view.	**Application**	**Aspects**, **ViewQualities**, **TraitsManager**
ViewQualities	Represents a collection of database view qualities.	**View**	(None)

In the balance of this article, I will show you how to programmatically manipulate the Microsoft Data Analyzer ActiveX control through VBA code. You can easily adapt this code to Visual Basic or VBScript, depending on your solution environment.

Working with Views

The **Application** object's **ActiveView** property returns a **View** object representing the active database view. Two of the most common methods of the **View** object are **OpenView** and **SaveView**, as shown in the following example code:

```
Private Const VIEW_NAME As String = _
  "C:\Program Files\Microsoft Data Analyzer\Data Analyzer 3.5\FoodMart2000Connection.max"

Private Sub Max3Ax1_Initialized()
 With Max3Ax1.Application.ActiveView
      .OpenView VIEW_NAME, vlocFileSystem
      ' Do some processing on the view here.
      .SaveView VIEW_NAME, vlocFileSystem
   End With
End Sub
```

Working with Built-In Dialog Boxes

You can use the **Application** object's **ShowDialog** method to display built-in dialog boxes. The **EMaxDialogs** enumerated type contains constants for the various built-in dialog boxes; for example, to show the **Change View** dialog box, you would use the following code:

```
Max3Ax1.Application.ShowDialog Dialog:=mxDlgChangeVie
```

Working with the Toolbar, Main Menu, and Status Bar

The Main Toolbar, the main menu, and the Status Bar can be made visible or hidden by setting the **Band** object's **Visible** property to **True** or **False**, as shown in the following example code:

```
' Hide the Main Toolbar, the main menu, and the Status Bar.
With Max3Ax1.Application.MainToolbar
   .Bands(Index:="MainToolBar").Visible = False
   .Bands(Index:="MainMenu").Visible = False
   .Bands(Index:="Main.StatusBar").Visible = False
End With
```

Note

If you hide the main menu, this setting will not persist if Microsoft Internet Explorer is refreshed.

This is a great technique if you want to disables many of the "bells and whistles" in the interface from the user at run time.

Working with Qualities and Traits

A *quality* is a unit of measurement in a database view. A *trait* is the type of measurement (for example, length or color). Traits, once defined, are the same for all aspects in a database view.

The following example code reports on the color trait:

' Reports the value of the Color trait (Change View dialog box (View menu), Measures tab, Color list).
MsgBox Prompt:=Max3Ax1.Application.ActiveView.TraitsManager.Trait(TraitID:=trtColor).Qualities.QualityID(v:=0)

You can use the **Add**, **Clear**, and **Remove** methods of the **Qualities** object to add, clear, and remove qualities.

Working with Aspects

An *aspect* is a dimension in an OLAP cube, such as time, geographical location, product, and so on. The **Aspects** collection allows you to use the **Add**, **Remove**, and **Clear** methods to add, remove, and clear aspects, respectively. When you use the **Add** or **Remove** methods, you must specify the unique dimension name in the data cube, and you must enclose the dimension's name in brackets "[ ]". For example:

```
With Max3Ax1.Application.ActiveView
   ' Next line of code will cause an error if  this dimension doesn't exist in the database or
   ' this dimesion is already visible in the view.
   .Aspects.Add "[Education Level]"
   ' Next line of code will cause an error if   this dimension is not already visible in the view.
   .Aspects.Remove "[Gender]"
End With
```

This technique is useful if you want to conditionally add or remove visible dimensions at run time.

Max3API Library VBA Code Sample Reference for Microsoft Data Analyzer

Microsoft® Data Analyzer ships with a compiled HTML Help (CHM) file titled **Microsoft Data Analyzer API Help** with the default file path of C:\Program Files\Microsoft Data Analyzer\Data Analyzer 3.5\MSDA35om.chm. This CHM file contains a few snippets of Microsoft Visual Basic® code and Microsoft Visual Basic Scripting Edition code; individual API topics contain C++ code stubs but no Visual Basic or VBScript code samples. This article contains Visual Basic for Applications (VBA) code samples for many of the API member topics.

For complete descriptions of each of the API's objects, collections, members, and enumerations, consult the Microsoft Data Analyzer API Help file (MSDA35om.chm) included with Data Analyzer.

For purposes of these code samples, this reference assumes an instance of the Microsoft Data Analyzer ActiveX® control has been embedded on a VBA UserForm and references have been set to the **Max3 ActiveX 3.0 Type Library** (using the default path C:\Program Files\Microsoft Data Analyzer\Data Analyzer 3.5\Max3ActiveX.dll) and the **Max3API** library (using the default path C:\Program Files\Microsoft Data Analyzer\Data Analyzer 3.5\Max3API.dll).

Only code for the **Max3API** library (Max3API.dll) will be presented in this reference. To gain entry into the **Max3API** library using VBA, you can use code similar to the following:

```
Private Const VIEW_FILE_PATH As String =  "C:\Program Files\Microsoft Data Analyzer\Data Analyzer3.5\Airline.max"
Private Sub UserForm_Initialize()
   Dim daApp As Max3API.Application
   Set daApp = Max3Ax1.Application
   daApp.ActiveView.OpenView VIEW_FILE_PATH, vlocFileSystem
End Sub
```

To adapt these code samples for use in VBScript code samples, embed an instance of the Microsoft Data Analyzer ActiveX control (C:\Program Files\Microsoft Data Analyzer\Data Analyzer 3.5\Max3ActiveX.dll) on a Web page, copy and paste the VBA code sample from this reference, and modify it to adhere to VBScript syntax rules. For example, to repurpose the previous code sample using VBScript on a Web page, use the following code:

```
<script id=clientEventHandlersVBS language=vbscript>
<!--
Sub window_onload
   Const vlocFileSystem = 1
   Const VIEW_FILE_PATH ="C:\Program Files\Microsoft Data Analyzer\Data Analyzer3.5\Airline.max"
   Max3Ax1.Application.ActiveView.OpenView VIEW_FILE_PATH, vlocFileSystem
End Sub
-->
</script>
```

Object Model Map

The following figure provides a graphical representation of the relationships among the objects in the **Max3API** library.

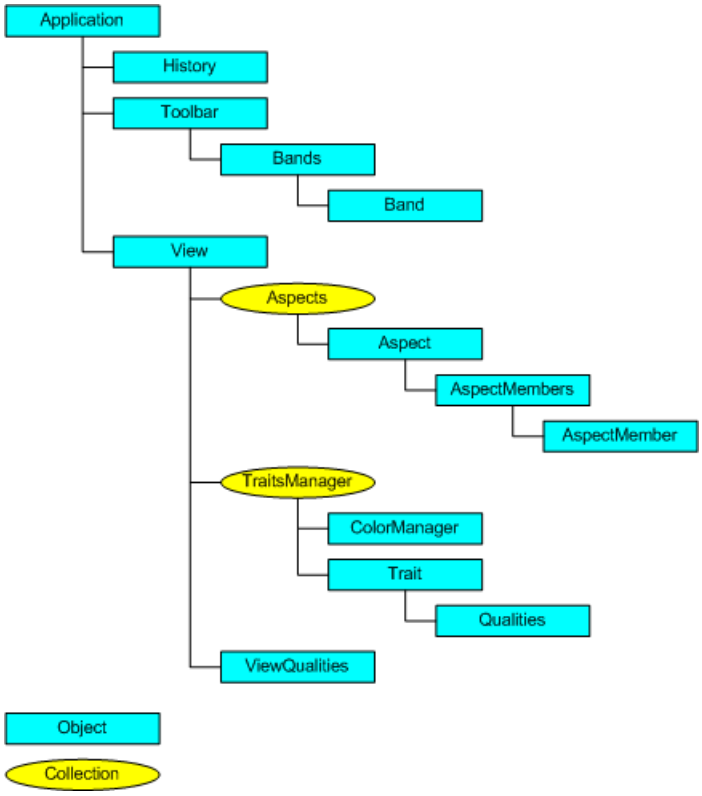


Figure 1. Max3API library object model map

Code Samples

Application Object: ActiveView Property/View Object: OpenView Method

The following code sample accesses the **Application** object to open a data view.

```
Private Const VIEW_FILE_PATH As String = "C:\Program Files\Microsoft Data Analyzer\Data Analyzer3.5\Airline.max"
Private Sub UserForm_Initialize()
   Dim daApp As Max3API.Application
   Set daApp = Max3Ax1.Application
   daApp.ActiveView.OpenView VIEW_FILE_PATH, vlocFileSystem
End Sub
```

Application Object: History Property / History Object: Back Method, Forward Method

The following code sample opens two data views and navigates back and forward between them.

```
Private Const AIRLINE_FILE_PATH As String = “C:\Program Files\Microsoft Data Analyzer\Data Analyzer3.5\Airline.max"
Private Const FOODMART_FILE_PATH As String = "C:\Program Files\Microsoft Data Analyzer\Data Analyzer _
    3.5\FoodMart2000.max"
Private Sub Max3Ax1_Initialized()
   Dim daApp As Max3API.Application
   Set daApp = Max3Ax1.Application
   daApp.ActiveView.OpenView AIRLINE_FILE_PATH, vlocFileSystem
   MsgBox "Airline.max opened."
   daApp.ActiveView.OpenView FOODMART_FILE_PATH, vlocFileSystem
   MsgBox "FoodMart2000.max opened."
   daApp.History.Back
   MsgBox "Back to Airline.max."
   daApp.History.Forward
   MsgBox "Forward to FoodMart2000.max."
End Sub
```

Application Object: MainToolbar Property/ Band Object: Visible Property/ Bands Object: Item Property / Toolbar Object: Bands Property

The following code sample hides the Data Analyzer ActiveX control's main toolbar, status bar, and main menu bar.

```
Private Sub Max3Ax1_Initialized()
   Dim daApp As Max3API.Application
   Set daApp = Max3Ax1.Application
   With daApp.MainToolbar.Bands
      .Item("MainToolBar").Visible = False
      .Item("MainMenu").Visible = False
      .Item("Main.StatusBar").Visible = False
```

```
    End With
End Sub
```

Application Object: ShowDialog Method

The following code sample displays the **About** dialog box.

```
Private Sub Max3Ax1_Initialized()
    Dim daApp As Max3API.Application

    Set daApp = Max3Ax1.Application
    daApp.ShowDialog mxDlgAbout
End Sub
```

Aspect Object: ID Property, Members Property, Minimized Property/ AspectMember Object: Selected Property / AspectMembers Object: Count Property, Item Property/ View Object: Aspects Property

The following code sample uses several **For Each** loops to list the visible OLAP dimensions and members in the Data Analyzer ActiveX control.

```
Private Const VIEW_FILE_PATH As String = "C:\Program Files\Microsoft Data Analyzer\Data Analyzer3.5\Airline.max"

Private Sub Max3Ax1_Initialized()
    Dim daApp As Max3API.Application
    Dim objAspect As Max3API.Aspect
    Dim intMember As Integer
    Dim strResults As String

    Set daApp = Max3Ax1.Application
    daApp.ActiveView.OpenView VIEW_FILE_PATH, vlocFileSystem
    strResults = "Aspects (OLAP Dimensions)" & _
        vbCrLf & vbTab & "Aspect Members (OLAP Members): Selected?"&  vbCrLf & vbCrLf
    For Each objAspect In daApp.ActiveView.Aspects
        objAspect.Minimized = True
        strResults = strResults & objAspect.ID & vbCrLf
        For intMember = 0 To objAspect.Members.Count - 1
            strResults = strResults & vbTab & _
                objAspect.Members.Item(intMember).ID & ": " &  objAspect.Members.Item(intMember).Selected & vbCrLf
        Next intMember
    Next objAspect
    MsgBox strResults
    For Each objAspect In daApp.ActiveView.Aspects
        objAspect.Minimized = False
    Next objAspect
End Sub
```

Aspect Object: MDHHeirarchy Property, AspectMember Object: MDHMember Property, View Object: MetaData Property

The **MDHHeirarchy** property returns an **IMdhHeirarchy** object. The **MDHMember** property returns an **IMdhMember** object. The **MetaData** property returns an **IMdhManager** object. All three of these returned objects are contained in the **MdhInterfacesLib** library (using the default path C:\Program Files\\Microsoft Data Analyzer\Data Analyzer 3.5\MDHInterfaces.tlb). Because none of these returned objects are contained in the **Max3API** library, they are not included in this reference. For more information, see the "Use Metadata" topic in the **Microsoft Data Analyzer API** help file (MSDA35om.chm).

Aspect Object: ExcludeUnselected Method, GotoLevel Method, ReverseSelection Method, SelectAll Method, SelectOnly Method, SmartSelectByName Method, SmartSelectByProperty Method, SmartSelectByQuality Method

Aspects Collection: Item Property, Qualities Object: Count Property, QualityID Property, QualityType Property

View Object: Qualities Property

The following code sample filters a data view using various methods.

```
Private Const VIEW_FILE_PATH As String = "C:\Program Files\Microsoft Data Analyzer\Data Analyzer3.5\Airline.max"
Private Const ASPECT_NAME As String = "[Destinations]"
Private Const ASPECT_LEVEL_ALL As String = ".[All Destinations]"
Private Const ASPECT_LEVEL As String = ".[Destination]"

Private Sub Max3Ax1_Initialized()
    Dim daApp As Max3API.Application
    Dim objAspect As Max3API.Aspect
    Dim intQuality As Integer
    Dim strResults As String

    Set daApp = Max3Ax1.Application
    daApp.ActiveView.OpenView VIEW_FILE_PATH, vlocFileSystem
    strResults = "Qualities (OLAP Measures), Quality Types: " & vbCrLf
    For intQuality = 0 To daApp.ActiveView.Qualities.Count - 1
        strResults = strResults & daApp.ActiveView.Qualities.QualityID(intQuality) & ", " & _
```

```
                daApp.ActiveView.Qualities.QualityType(intQuality) & vbCrLf
    Next intQuality
    MsgBox strResults
    Set objAspect = daApp.ActiveView.Aspects.Item(ASPECT_NAME)
    objAspect.SelectAll
    Call ListAspectMembers(objAspect)
    objAspect.GotoLevel ASPECT_NAME & ASPECT_LEVEL
    Call ListAspectMembers(objAspect)
    objAspect.GotoLevel ASPECT_NAME & ".[Region Name]"
    objAspect.SelectOnly ASPECT_NAME & ASPECT_LEVEL_ALL & ".[West Europe]"
    objAspect.ExcludeUnselected
    Call ListAspectMembers(objAspect)
    objAspect.GotoLevel ASPECT_NAME & ".[Region Name]"
    objAspect.SmartSelectByName "", ssopExclude, ssorStartsWith,  "West", "", False
    objAspect.ReverseSelection
    Call ListAspectMembers(objAspect)
    objAspect.GotoLevel ASPECT_NAME & ASPECT_LEVEL
    objAspect.SmartSelectByProperty "", ssopExclude, _
       "Total Revenue", ssorGT, 20000, "", False
    Call ListAspectMembers(objAspect)
    objAspect.SmartSelectByQuality "", ssopExclude, qtypMeasure, "[Measures].[Profitability]", ssorGT, 30#, "", False
    Call ListAspectMembers(objAspect)
End Sub

Private Sub ListAspectMembers(objAspect As Max3API.Aspect)
    Dim intAspectMember As Integer
    Dim strResults As String

    strResults = "Aspects (OLAP Dimensions)" & vbCrLf & vbTab & "Aspect Members (OLAP Members)" & vbCrLf
    strResults = strResults & objAspect.ID & vbCrLf
    For intAspectMember = 0 To objAspect.Members.Count - 1
       strResults = strResults & vbTab & objAspect.Members.Item(intAspectMember).ID & vbCrLf
    Next intAspectMember
    MsgBox strResults
End Sub
```

AspectMember Object: Values Property

The following code sample lists the properties and data values of a data view.

```
Private Const VIEW_FILE_PATH As String = "C:\Program Files\Microsoft Data Analyzer\Data Analyzer3.5\Airline.max"

Private Sub Max3Ax1_Initialized()
    Dim daApp As Max3API.Application
    Dim objAspect As Max3API.Aspect
    Dim intQuality As Integer
    Dim intMember As Integer
    Dim strResults As String

    Set daApp = Max3Ax1.Application
    daApp.ActiveView.OpenView VIEW_FILE_PATH, vlocFileSystem
    strResults = "Aspect Members (OLAP Measures), Selected?" & vbCrLf & vbTab & "Values" & vbCrLf
    For Each objAspect In daApp.ActiveView.Aspects
       For intMember = 0 To objAspect.Members.Count - 1
          strResults = strResults & objAspect.Members.Item(intMember).ID & ", " & _
                       objAspect.Members.Item(intMember).Selected & vbCrLf
          For intQuality = 0 To daApp.ActiveView.Qualities.Count - 1
             strResults = strResults & vbTab & daApp.ActiveView.Qualities.QualityID(intQuality) & _
                   ": " & objAspect.Members.Item(intMember).Values   intQuality) & vbCrLf
          Next intQuality
       Next intMember
    Next objAspect
    Debug.Print strResults
End Sub
```

Aspects Collection: Add Method, Clear Method, Remove Method / AspectMembers Object: Add Method, Remove Method

The following code sample removes, adds, modifies, and clears various components of a data view.

```
Private Const VIEW_FILE_PATH As String = "C:\Program Files\Microsoft Data Analyzer\Data Analyzer3.5\Airline.max"

Private Const ASPECT_NAME As String = "[Destinations]"

Private Sub Max3Ax1_Initialized()
    Dim daApp As Max3API.Application
    Dim objAspect As Max3API.Aspect
```

```
    Dim intQuality As Integer
    Dim intMember As Integer
    Dim strResults As String

    Set daApp = Max3Ax1.Application
    daApp.ActiveView.OpenView VIEW_FILE_PATH, vlocFileSystem
    Call ListAspects(daApp)
    daApp.ActiveView.Aspects.Remove ASPECT_NAME
    Call ListAspects(daApp)
    daApp.ActiveView.Aspects.Add ASPECT_NAME
    Call ListAspects(daApp)
    daApp.ActiveView.Aspects.Item(ASPECT_NAME).Members.Remove ASPECT_NAME & ASPECT_LEVEL & ".[East Asia]"
    Call ListAspects(daApp)
    daApp.ActiveView.Aspects.Item(ASPECT_NAME).Members.Add  ASPECT_NAME & ASPECT_LEVEL & ".[East Asia]"
    Call ListAspects(daApp)
    daApp.ActiveView.Aspects.Clear
End Sub

Private Sub ListAspects(daApp As Max3API.Application)
    Dim objAspect As Max3API.Aspect

    For Each objAspect In daApp.ActiveView.Aspects
        Call ListAspectMembers(objAspect)
    Next objAspect
End Sub

Private Sub ListAspectMembers(objAspect As Max3API.Aspect)
    Dim intAspectMember As Integer
    Dim strResults As String

    strResults = "Aspects (OLAP Dimensions)" & vbCrLf & _
        vbTab & "Aspect Members (OLAP Members)" & vbCrLf
    strResults = strResults & objAspect.ID & vbCrLf
    For intAspectMember = 0 To objAspect.Members.Count - 1
        strResults = strResults & vbTab & _
            objAspect.Members.Item(intAspectMember).ID & vbCrLf
    Next intAspectMember
    MsgBox strResults
End Sub
```

ColorManager Object: ColorScaleVisible Property, Qualities Object: Add Method, Clear Method, Remove Method, View Object: TraitsManager Property, Trait Object: ID Property, Qualities Property, SingleQualityID Property

The following code sample modifies and lists various data view properties and property traits.

```
Private Const VIEW_FILE_PATH As String = "C:\Program Files\Microsoft Data Analyzer\Data Analyzer3.5\Airline.max"
Private strResults As String

Private Sub Max3Ax1_Initialized()
    Dim daApp As Max3API.Application
    Dim objQualities As Max3API.Qualities
    Dim objTrait As Max3API.Trait

 Set daApp = Max3Ax1.Application
    daApp.ActiveView.OpenView VIEW_FILE_PATH, vlocFileSystem
    daApp.ActiveView.TraitsManager.ColorManager.ColorScaleVisible =False
Set objTrait = daApp.ActiveView.TraitsManager.Trait(trtColor)
    Call ListTraitProperties(objTrait)
Set objTrait = daApp.ActiveView.TraitsManager.Trait(trtGrid)
Call ListTraitProperties(objTrait)
Set objTrait = daApp.ActiveView.TraitsManager.Trait(trtLength)
Call ListTraitProperties(objTrait)
Set objQualities = daApp.ActiveView.TraitsManager.Trait(trtColor).Qualities
    strResults = "Color Qualities (Properties):" & vbCrLf
Call ListQualities(objQualities)
Set objQualities = daApp.ActiveView.TraitsManager.Trait(trtGrid).Qualities
    strResults = "Grid Qualities (Properties):" & vbCrLf
    objQualities.Clear
    strResults = "Grid Qualities (Properties):" & vbCrLf
Call ListQualities(objQualities)
    objQualities.Add qtypMeasure, "[Measures].[Number Of Seats]"
    strResults = "Grid Qualities (Properties):" & vbCrLf
Call ListQualities(objQualities)
    objQualities.Remove qtypMeasure, "[Measures].[Number Of Seats]"
    strResults = "Grid Qualities (Properties):" & vbCrLf
```

```
Call ListQualities(objQualities)
Set objQualities =daApp.ActiveView.TraitsManager.Trait(trtLength).Qualities
   strResults = "Length Qualities (Properties):" & vbCrLf
 Call ListQualities(objQualities)
End Sub

Private Sub ListQualities(objQualities As Max3API.Qualities)
   Dim intQuality As Integer

   For intQuality = 0 To objQualities.Count - 1
      strResults = strResults & objQualities.QualityID(intQuality) & vbCrLf
   Next intQuality
   MsgBox strResults
End Sub

Private Sub ListTraitProperties(objTrait As Max3API.Trait)

   strResults = "Trait ID, Single Quality ID, Single Quality Type" & vbCrLf & objTrait.ID & ", " &_
         objTrait.SingleQualityID &  ", " & objTrait.SingleQualityType
   MsgBox strResults
End Sub
```

View Object: Catalog Property, Connect Method, CloseView Method, Cube Property, IsOpen Property, Refresh Method

The following code sample saves a data view using various methods. This code sample also opens a new view based on a connection to a Microsoft SQL Server running Analysis Services.

```
Private Const VIEW_FILE_PATH As String = "C:\Program Files\Microsoft Data Analyzer\Data Analyzer3.5\Airline.max"
Private Const VIEW_SAVE_PATH As String = _
         "C:\Program Files\Microsoft Data Analyzer\Data Analyzer3.5\Airline_1.max"
Private Const VIEW_HTML_PATH_1 As String = _
         "C:\Program Files\Microsoft Data Analyzer\Data Analyzer3.5\Airline_Bars.htm"
Private Const VIEW_HTML_PATH_2 As String = _
         "C:\Program Files\Microsoft Data Analyzer\Data Analyzer3.5\Airline_Grid.htm"

Private Sub Max3Ax1_Initialized()
   Dim daApp As Max3API.Application
   Dim objQualities As Max3API.Qualities
   Dim objTrait As Max3API.Trait

Set daApp = Max3Ax1.Application
   daApp.ActiveView.OpenView VIEW_FILE_PATH, vlocFileSystem
   daApp.ActiveView.SaveView VIEW_SAVE_PATH, vlocFileSystem
MsgBox "Saved to " & VIEW_SAVE_PATH
   daApp.ActiveView.SaveAsWebPage VIEW_HTML_PATH_1,  wpgkBars, "Airline (Bars)"
MsgBox "Saved to " & VIEW_HTML_PATH_1
   daApp.ActiveView.SaveAsWebPageEx VIEW_HTML_PATH_2,  "Maximal.Grid", "Airline (Grid)"
MsgBox "Saved to " & VIEW_HTML_PATH_2
   daApp.ActiveView.Refresh rfrkSoft
MsgBox "View refreshed."
   daApp.ActiveView.CloseView
   daApp.ActiveView.Connect "location=MyOLAPServerName;provider=msolap"
   daApp.ActiveView.Catalog = "FoodMart 2000"
   daApp.ActiveView.Cube = "Sales"
MsgBox "Catalog Name: " & daApp.ActiveView.Catalog
MsgBox "Cube Name: " & daApp.ActiveView.Cube
MsgBox "View Open: " & daApp.ActiveView.IsOpen
End Sub
```

ViewQualities Object: Count Property, QualityID Property, QualityType Property

The following code sample provides a list of data view properties.

```
Private Const VIEW_FILE_PATH As String =  "C:\Program Files\Microsoft Data Analyzer\Data Analyzer3.5\Airline.max"

Private Sub Max3Ax1_Initialized()
   Dim daApp As Max3API.Application
   Dim intQuality As Integer
   Dim strResults As String

Set daApp = Max3Ax1.Application
   daApp.ActiveView.OpenView VIEW_FILE_PATH, vlocFileSystem
   strResults = "View Qualities (Properties), View Quality Types" & vbCrLf
   For intQuality = 0 To daApp.ActiveView.Qualities.Count - 1
      strResults = strResults & _
    daApp.ActiveView.Qualities.QualityID(intQuality) & ", " & daApp.ActiveView.Qualities.QualityType(intQuality) & vbCrLf
   Next intQuality
MsgBox strResults
End Sub
```

MdhInterfacesLib Library VBA Code Sample Reference for Microsoft Data Analyzer

Microsoft® Data Analyzer ships with a compiled HTML Help (CHM) file titled **Microsoft Data Analyzer API Help** with the default file path of C:\Program Files\Microsoft Data Analyzer\Data Analyzer 3.5\MSDA35om.chm. This CHM file contains a few snippets of Microsoft Visual Basic® code and Microsoft Visual Basic Scripting Edition code; individual API topics contain C++ code stubs but no Visual Basic or VBScript code samples. This article contains Visual Basic for Applications (VBA) code samples for some of the API topics.

For complete descriptions of each of the API's objects, collections, members, and enumerations, consult the Microsoft Data Analyzer API Help file (MSDA35om.chm) included with Data Analyzer.

For purposes of these code samples, this reference assumes an instance of the Microsoft Data Analyzer ActiveX® control has been embedded on a VBA UserForm and references have been set to the:

- **Max3 ActiveX 3.0 Type Library** (using the default path C:\Program Files\Microsoft Data Analyzer\Data Analyzer 3.5\Max3ActiveX.dll)
- **Max3API** library (using the default path C:\Program Files\Microsoft Data Analyzer\Data Analyzer 3.5\Max3API.dll)
- **MdhInterfacesLib** library (using the default path C:\Program Files\Microsoft Data Analyzer\Data Analyzer 3.5\MdhInterfacesLib.tlb)

Only code for the **MdhInterfacesLib** library (MdhInterfacesLib.tlb) will be presented in this reference.

The **MDHHeirarchy** property in the **Max3API** library returns an **IMdhHeirarchy** object in the **MdhInterfacesLib** library. The **MDHMember** property in the **Max3API** library returns an **IMdhMember** object in the **MdhInterfacesLib** library. The **MetaData** property in the **Max3API** library returns an **IMdhManager** object in the **MdhInterfacesLib** library. Because none of these three properties are contained in the **MdhInterfacesLib** library, they are not included in this reference. For more information on these properties, see the **Microsoft Data Analyzer API** help file (MSDA35om.chm). For example, to gain entry into the **MdhInterfacesLib** library from the **Max3API** library using VBA, you can use code similar to the following:

```
Private Sub Max3Ax1_Initialized()
   Dim daApp As Max3API.Application
   Dim objCMdhInterfaces As MdhInterfacesLib.CMdhInterfaces
   Dim intCube As Integer
   Dim strResults As String
Set daApp = Max3Ax1.Application
   daApp.ActiveView.Connect  "location=MyOLAPServerName;provider=msolap"
Set objCMdhInterfaces = daApp.ActiveView.MetaData
   strResults = "Cubes:" & vbCrLf
   For intCube = 0 To objCMdhInterfaces.Cubes.Count - 1
      strResults = strResults & objCMdhInterfaces.Cubes.Item(intCube).Name & vbCrLf
   Next intCube
MsgBox strResults
End Sub
```

Object Model Map

The following figure provides a graphical representation of the relationships among the objects in the **MdhInterfacesLib** library.

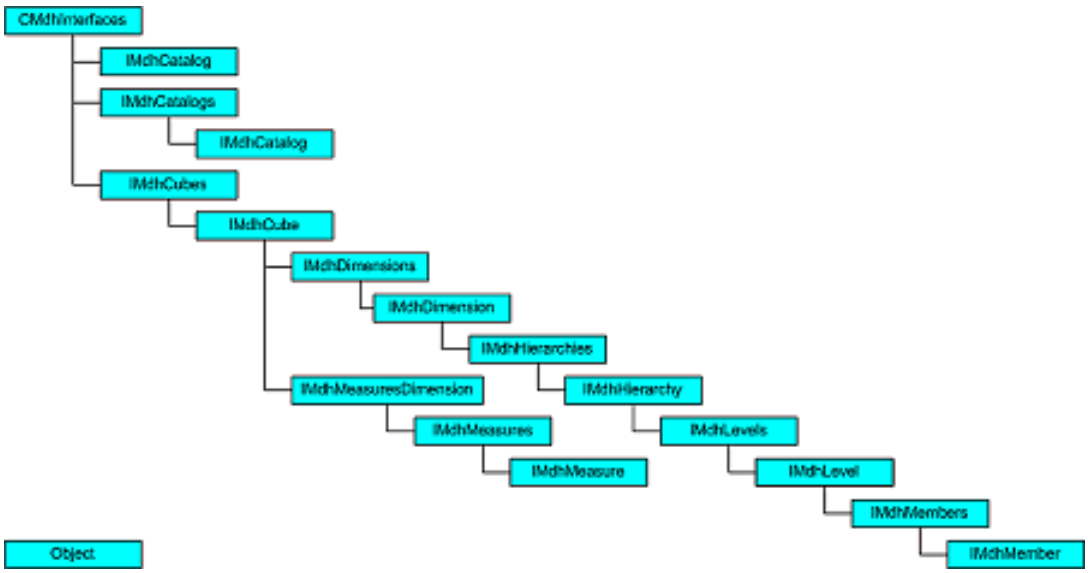


Figure 1. MdhInterfacesLib library object model map (click picture to see larger image)

Code Samples

CMdhInterfaces Object: ActiveCatalog Property, Catalogs Property, ConnectionDescription Property

IMdhCatalog Object: Name Property, IMdhCatalogs Object: Count Property, Item Property

The following code sample provides high-level information about the catalogs on a particular OLAP server.

```
Private Sub Max3Ax1_Initialized()
```

```
   Dim daApp As Max3API.Application
   Dim objCMdhInterfaces As MdhInterfacesLib.CMdhInterfaces
   Dim intCatalog As Integer
   Dim strResults As String
Set daApp = Max3Ax1.Application
   daApp.ActiveView.Connect  "location= MyOLAPServerName;provider=msolap"
Set objCMdhInterfaces = daApp.ActiveView.MetaData
   strResults = "Connection Description: " & objCMdhInterfaces.ConnectionDescription & vbCrLf
   strResults = strResults & "Catalogs:" & vbCrLf
   For intCatalog = 0 To objCMdhInterfaces.Catalogs.Count - 1
      strResults = strResults & " " & objCMdhInterfaces.Catalogs.Item(intCatalog).Name & vbCrLf
   Next intCatalog
   strResults = strResults & "Active Catalog: " & objCMdhInterfaces.ActiveCatalog.Name
MsgBox strResults
End Sub
```

CMdhInterfaces Object: Cubes Property, IMdhCube Object: Dimensions Property, LastUpdate Property, Name Property, Type Property, IMdhCubes Object: Count Property, Item Property, IMdhDimension Object: Name Property, IMdhDimensions Object: Count Property, Item Property

The following code sample iterates through the cubes and dimensions on a particular OLAP server.

```
Private Sub Max3Ax1_Initialized()
   Dim daApp As Max3API.Application
   Dim objCMdhInterfaces As MdhInterfacesLib.CMdhInterfaces
   Dim objIMdhCube As MdhInterfacesLib.IMdhCube
   Dim objIMdhDimension As MdhInterfacesLib.IMdhDimension
   Dim intCube As Integer
   Dim intDimension As Integer
   Dim strResults As String
Set daApp = Max3Ax1.Application
   daApp.ActiveView.Connect  "location= MyOLAPServerName;provider=msolap"
Set objCMdhInterfaces = daApp.ActiveView.MetaData
   strResults = "Cubes / Dimensions: " & vbCrLf
   For intCube = 0 To objCMdhInterfaces.Cubes.Count - 1
      Set objIMdhCube = objCMdhInterfaces.Cubes.Item(intCube)
      strResults = strResults & "Cube: " & objIMdhCube.Name & vbCrLf & " Last Update: " & _
                   objIMdhCube.LastUpdate & vbCrLf & " Type: " & objIMdhCube.Type & vbCrLf
         For intDimension = 0 To objIMdhCube.Dimensions.Count - 1
            Set objIMdhDimension = objIMdhCube.Dimensions.Item(intDimension)
            strResults = strResults & "  Dimension: " & objIMdhDimension.Name & vbCrLf
         Next intDimension
   Next intCube
Debug.Print strResults
End Sub
```

IMdhCube Object: MeasuresDimension Property, IMdhMeasure Object: AggregatorType Property, Caption Property, DataType Property, Dimension Property, NumericPrecision Property, NumericScale Property, Type Property, UniqueName Property, IMdhMeasures Object: Count Property, Item Property, IMdhMeasuresDimension Object: Measures Property

The following code sample iterates through the cubes and measures on a particular OLAP server.

```
Private Sub Max3Ax1_Initialized()
   Dim daApp As Max3API.Application
   Dim objCMdhInterfaces As MdhInterfacesLib.CMdhInterfaces
   Dim objIMdhCube As MdhInterfacesLib.IMdhCube
   Dim objIMdhMeasure As MdhInterfacesLib.IMdhMeasure
   Dim intCube As Integer
   Dim intMeasure As Integer
   Dim strResults As String
Set daApp = Max3Ax1.Application
   daApp.ActiveView.Connect  "location=MyOLAPServerName;provider=msolap"
Set objCMdhInterfaces = daApp.ActiveView.MetaData
   strResults = "Cubes / Measures: " & vbCrLf
   For intCube = 0 To objCMdhInterfaces.Cubes.Count - 1
      Set objIMdhCube = objCMdhInterfaces.Cubes.Item(intCube)
      strResults = strResults & "Cube: " & objIMdhCube.Name & vbCrLf
      strResults = strResults & " Measures:" & vbCrLf
      For intMeasure = 0 To objIMdhCube.MeasuresDimension.Measures.Count - 1
         Set objIMdhMeasure =  objIMdhCube.MeasuresDimension.Measures.Item(intMeasure)
```

```
        strResults = strResults & "  Unique Name: " & objIMdhMeasure.UniqueName & vbCrLf & _
                     "  Aggregator Type: " & objIMdhMeasure.AggregatorType & vbCrLf & _
                     "  Caption: " & objIMdhMeasure.Caption & vbCrLf & _
                     "  Data Type: " & objIMdhMeasure.DataType & vbCrLf & _
                     "  Dimension Name: " & objIMdhMeasure.Dimension.Name & vbCrLf &_
                     "  Numeric Precision: " & objIMdhMeasure.NumericPrecision & vbCrLf &_
                     "  Numeric Scale: " & ojIMdhMeasure.NumericScale & vbCrLf & _
                     "  Type: " & objIMdhMeasure.Type & vbCrLf
      Next intMeasure
    Next intCube
Debug.Print strResults
End Sub
```

IMdhDimension Object: Caption Property, Hierarchies Property, IsMeasuresDimension Property, UniqueName Property
IMdhHierarchies Object: Count Property, Item Property
IMdhHierarchy Object: Caption Property, Cardinality Property, DefaultMember Property, Dimension Property, UniqueName Property

The following code sample iterates through the cubes, hierarchies, and members on a particular OLAP server.

```
Private Sub Max3Ax1_Initialized()
    Dim daApp As Max3API.Application
    Dim objCMdhInterfaces As MdhInterfacesLib.CMdhInterfaces
    Dim objIMdhCube As MdhInterfacesLib.IMdhCube
    Dim objIMdhDimension As MdhInterfacesLib.IMdhDimension
    Dim objIMdhHierarchy As MdhInterfacesLib.IMdhHierarchy
    Dim intCube As Integer
    Dim intDimension As Integer
    Dim intHierarchy As Integer
    Dim strResults As String

Set daApp = Max3Ax1.Application
    daApp.ActiveView.Connect  "location= MyOLAPServerName;provider=msolap"
Set objCMdhInterfaces = daApp.ActiveView.MetaData
    For intCube = 0 To objCMdhInterfaces.Cubes.Count - 1
      Set objIMdhCube = objCMdhInterfaces.Cubes.Item(intCube)
        strResults = strResults & "Cube: " & objIMdhCube.Name & vbCrLf
        For intDimension = 0 To objIMdhCube.Dimensions.Count - 1
          Set objIMdhDimension =  objIMdhCube.Dimensions.Item(intDimension)
            strResults = strResults & _
              " Dimension Name: " & objIMdhDimension.Name & vbCrLf & _
              "  Unique Name: " & objIMdhDimension.UniqueName & vbCrLf & _
              "  Caption: " & objIMdhDimension.Caption & vbCrLf & _
              "  Cube Name: " & objIMdhDimension.Cube.Name & vbCrLf & _
              "  Is This A Measures Dimension: " & objIMdhDimension.IsMeauresDimension & vbCrLf
            For intHierarchy = 0 To _
              objIMdhDimension.Hierarchies.Count - 1
              Set objIMdhHierarchy = objIMdhDimension.Hierarchies.Item(intHierarchy)
          strResults = strResults & "  Dimension Hierarchy Unique Name: " & objIMdhHierarchy.UniqueName & vbCrLf & _
                                    "  Caption: " & objIMdhHierarchy.Caption & vbCrLf & _
                                    "  Cardinality: " &  objIMdhHierarchy.Cardinality & vbCrLf & _
                                    "  Default Member Caption: " & objIMdhHierarchy.DefaultMember.Caption & vbCrLf & _
                                    "  Dimension Caption: " & objIMdhHierarchy.Dimension.Caption & vbCrLf
            Next intHierarchy
        Next intDimension
    Next intCube
Debug.Print strResults
End Sub
```

IMdhHierarchy Object: Levels Property, IMdhLevel Object: Caption Property, Dimension Property, LevelNumber Property, MemberProperties Property, Members Property, MemberUserProperties Property, Name Property, Type Property, UniqueName Property, IMdhLevels Object: Item Property, IMdhMember Object: Children Property, Dimension Property, Level Property, Name Property, Properties Property, Type Property, UniqueName Property, UserProperties Property, IMdhMembers Object: Item Property

The following code lists information about the first cube on a particular OLAP server, the first dimension in the cube, the first hierarchy in the dimension, the first level in the hierarchy, and the first member in the level.

```
Private Sub Max3Ax1_Initialized()
    Dim daApp As Max3API.Application
    Dim objCMdhInterfaces As MdhInterfacesLib.CMdhInterfaces
    Dim objIMdhCube As MdhInterfacesLib.IMdhCube
```

```
   Dim objIMdhDimension As MdhInterfacesLib.IMdhDimension
   Dim objIMdhHierarchy As MdhInterfacesLib.IMdhHierarchy
   Dim objIMdhLevel As MdhInterfacesLib.IMdhLevel
   Dim objIMdhMember As MdhInterfacesLib.IMdhMember
   Dim intCube As Integer
   Dim strResults As String

   Set daApp = Max3Ax1.Application
   daApp.ActiveView.Connect  "location= MyOLAPServerName;provider=msolap"
   Set objCMdhInterfaces = daApp.ActiveView.MetaData
   For intCube = 0 To objCMdhInterfaces.Cubes.Count - 1
      Set objIMdhCube = objCMdhInterfaces.Cubes.Item(intCube)
      strResults = strResults & "Cube: " & objIMdhCube.Name & vbCrLf
      Set objIMdhDimension = objIMdhCube.Dimensions.Item(0)
      strResults = strResults & " First Dimension Name: " & objIMdhDimension.Name & vbCrLf
      Set objIMdhHierarchy = objIMdhDimension.Hierarchies.Item(0)
      strResults = strResults & "  First Hierarchy Unique Name: " & objIMdhHierarchy.UniqueName & vbCrLf
      Set objIMdhLevel = objIMdhHierarchy.Levels.Item(0)
      strResults = strResults & _
         "    First Level Name: " & objIMdhLevel.Name & vbCrLf & _
         "     Unique Name: " & objIMdhLevel.UniqueName & vbCrLf & _
         "     Caption: " & objIMdhLevel.Caption & vbCrLf & _
         "     Dimension Name: " & objIMdhLevel.Dimension.Name & vbCrLf & _
         "     Level Number: " & objIMdhLevel.LevelNumber & vbCrLf & _
         "     Count of Member Properties: " & objIMdhLevel.MemberProperties.Count & vbCrLf & _
         "     Count of Members: " & objIMdhLevel.Members.Count & vbCrLf & _
         "     Count of Member User Properties: " & objIMdhLevel.MemberUserProperties.Count & vbCrLf & _
         "     Type: " & objIMdhLevel.Type & vbCrLf
      Set objIMdhMember = objIMdhLevel.Members.Item(0)
      strResults = strResults & "     First Member Name: " & objIMdhMember.Name & vbCrLf & _
                                "     Unique Name: " & objIMdhMember.UniqueName & vbCrL & _
                                "     Caption: " & objIMdhMember.Caption & vbCrLf & _
                                "     Count of Children: " & objIMdhMember.Children.Count & vbCrLf & _
                                "     Dimension Name: " & objIMdhMember.Dimension.Name & vbCrLf & _
                                "     Count of Properties: " & objIMdhMember.Properties.Count & vbCrLf & _
                                "     Count of User Properties: " & objIMdhMember.UserProperties.Count & vbCrLf & _
                                "     Type: " & objIMdhMember.Type & vbCrLf & _
                                "     Level Name: " & objIMdhMember.Level.Name & vbCrLf
   Next intCube
   Debug.Print strResults
End Sub
```

IMdhLevels Object: Count Property, IMdhMembers Object: Count Property

The following code sample provides a count of cubes on a particular OLAP server, a count of dimensions in the first cube, a count of hierarchies in the first dimension, a count of levels in the first hierarchy, and a count of members in the first level.

```
Private Sub Max3Ax1_Initialized()
   Dim daApp As Max3API.Application
   Dim objCMdhInterfaces As MdhInterfacesLib.CMdhInterfaces
   Dim objIMdhCube As MdhInterfacesLib.IMdhCube
   Dim objIMdhDimension As MdhInterfacesLib.IMdhDimension
   Dim objIMdhHierarchy As MdhInterfacesLib.IMdhHierarchy
   Dim objIMdhLevel As MdhInterfacesLib.IMdhLevel

Set daApp = Max3Ax1.Application
   daApp.ActiveView.Connect  "location= MyOLAPServerName;provider=msolap"
Set objCMdhInterfaces = daApp.ActiveView.MetaData
   Debug.Print "Cubes: " & objCMdhInterfaces.Cubes.Count
Set objIMdhCube = objCMdhInterfaces.Cubes.Item(0)
   Debug.Print " Dimensions in First Cube (" & objIMdhCube.Name & "): " & objIMdhCube.Dimensions.Count
Set objIMdhDimension = objIMdhCube.Dimensions.Item(0)
   Debug.Print "  Hierarchies in First Dimension (" & objIMdhDimension.UniqueName & "): " & _
      objIMdhDimension.Hierarchies.Count
Set objIMdhHierarchy = objIMdhDimension.Hierarchies.Item(0)
   Debug.Print "   Levels in First Hierarchy (" & objIMdhHierarchy.UniqueName & "): " &  objIMdhHierarchy.Levels.Count
Set objIMdhLevel = objIMdhHierarchy.Levels.Item(0)
   Debug.Print "    Members in First Level (" & objIMdhLevel.UniqueName & "): " & objIMdhLevel.Members.Count
End Sub
```

Using the Microsoft Data Analyzer ActiveX Control in Microsoft Excel 2002

Download Odc_dactrlexcel.exe.

Microsoft® Data Analyzer belongs to the Microsoft Office family of software applications. It provides a graphical analysis interface for users to apply business intelligence to their operational data. Data Analyzer is designed to work with online analytical processing (OLAP) data, based on Microsoft SQL Server™ 2000 with Analysis Services. By using Data Analyzer, users can spot trends, opportunities, and potential problems regardless of their level of technical expertise.

In addition to a standalone user interface, Data Analyzer also provides a Microsoft ActiveX® control that can be embedded in Microsoft Office application documents such as Microsoft Excel worksheets, Microsoft PowerPoint® slides, Microsoft Word documents, and manipulated with Microsoft Visual Basic® for Applications (VBA) code. The Data Analyzer ActiveX control can also be embedded in a UserForm in Component Object Model (COM)-based applications like Excel or Word. Additionally, the control can be embedded in Web pages in Microsoft FrontPage® and manipulated with Microsoft Visual Basic Scripting Edition (VBScript) code. Developers can use the Data Analyzer ActiveX control to access the Data Analyzer application-programming interface (API) in order to run most of the user-interface operations available in the standalone Data Analyzer. The Data Analyzer API contains a rich object model that can be used to programmatically control many features of Data Analyzer including the ability to:

- Create, load, configure, and save views
- Change the properties such as colors, visualization methods, dialogs, and functions
- Run menu items

For additional information on the Microsoft Data Analyzer Object Model, see Exploring Microsoft Data Analyzer Programmability.

In this article, we will demonstrate creating the forms and sample code to do the following:

- Embed the Data Analyzer ActiveX control into a UserForm in Excel 2002 and connect to a Data Analyzer view file
- Open a dialog box where the user can select from a list of Data Analyzer operations normally available from the user interface
- Open up another dialog box that displays a hierarchical view of the different components that make up a Data Analyzer view
- Add code to a workbook so that the sample opens automatically when the workbook is opened

Why Embed the Microsoft Data Analyzer ActiveX Control in Excel?

Why would you want to embed the Microsoft Data Analyzer ActiveX control inside of an Excel 2002 worksheet? On its own, Excel provides a number of features to help you work with OLAP data and analyze your multidimensional data. For example, the PivotTable® report is a special type of table that you can use to summarize information from a data source. The data source can be a relational file, an OLAP cube, or an Excel list. After specifying information during the creation of the PivotTable such as the fields that you are interested in, the layout of the fields, and the types of calculations you want, you can rearrange the data to provide any number of alternative perspectives.

Additionally, you can connect to OLAP data sources in Excel just as you do to other external data sources. For example, you can work with databases created with Microsoft SQL Server 2000 Analysis Services, the Microsoft OLAP server product. Excel can also work with third-party OLAP products that are compatible with OLE DB for OLAP.

Excel also provides a large number of functions that you can use to perform complex calculations on your data. By combining the calculation features of Excel with the data analysis capabilities of Data Analyzer, you can create very powerful applications.

For example, you can embed the Data Analyzer ActiveX control inside of an Excel UserForm to enable your users to quickly view their data and then provide custom VBA code that users can execute to retrieve selected data from the Data Analyzer view and insert that data into cells on a worksheet for further analysis and calculations. Or you could create a macro that retrieves data from a Data Analyzer view, defines special print settings, and prints a document containing the data with custom formatting and border styles.

Combining the capabilities of Excel and the Data Analyzer allows you to provide an easy-to-use data analysis tool to your users with the powerful calculation engine of Excel.

Embedding the Microsoft Data Analyzer ActiveX Control in Excel

First, we will embed the Microsoft Data Analyzer ActiveX control into a UserForm in Excel 2002 and manipulate it with VBA code. To do so, on a computer with Data Analyzer installed:

1. Start Excel 2002 and create a new, blank workbook.
2. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor** to switch to the Visual Basic Editor (VBE).
3. On the **Insert** menu, click **UserForm**.
4. In the Properties window for the UserForm, replace the text in the Name property with **frmControl** and the text in the Caption property with **Microsoft Data Analyzer ActiveX control—Sample**.
5. In the **Toolbox** (**View** menu), right-click, and click **Additional Controls**.
6. In the **Additional Controls** dialog box, check the **Max3Ax Class** box, and then click **OK**.
7. Finally, drag the Max3ax Class control onto the frmControl form.

Note At design time, the Data Analyzer ActiveX control's user interface will not be visible (see **Figure 1**). At run time, the Data Analyzer ActiveX control's user interface is visible and runs within the confines of the UserForm.

To program against the Data Analyzer ActiveX control, you must set a reference (on the **Tools** menu, click **References**) to the Max3API DLL (in a default installation of Data Analyzer, this dynamic-link library (DLL) can be found at C:\Program

Files\Microsoft Data Analyzer\Data Analyzer 3.5\Max3API.dll). Additionally, in this example we will be using interfaces from the MaxODBO 1.0 Type Library to retrieve OLAP metadata so you will need to set a reference to the MaxODBO 1.0 Type Library DLL (in a default installation of Data Analyzer, this DLL can be found at C:\Program Files\Microsoft Data Analyzer\Data Analyzer 3.5\MaxODBO.dll). And finally, in order to use the TreeView ActiveX control in this example, you will need to set a reference to the Microsoft TreeView Control, version 5.0 (SP2) OCX (usually found at C:\WINNT\System32\Comctl32.ocx).

To get started, you will first need to declare some variables so that they are available to all of the forms and procedures. To do this, we will create a standard module and declare the variables with the **Public** keyword. On the **Insert** menu, click **Module** and type the following:

Public mobjMSDA As Max3API.Application
Public mobjView As Max3API.View
Public mintDialog As Integer
Public strFileName As String
Public intUniqueID As Integer

The mobjMSDA variable represents the Max3API **Application** object that gives us programmatic control over the Data Analyzer user interface. The mobjView variable represents the Max3API **View** object, which gives us access to the methods and properties of a Data Analyzer view. The mintDialog variable will contain an **Integer** value that will be used to display a Data Analyzer dialog box. The strFileName **String** variable will contain the filename of the view that appears in Data Analyzer. And finally, the intUniqueID **Integer** variable will be used as a counter to append a unique identifier to a node name.

Adding Controls to the frmControl Form

We will now add three command buttons to the frmControl form. The form should resemble **Figure 1** when all of the controls have been added.

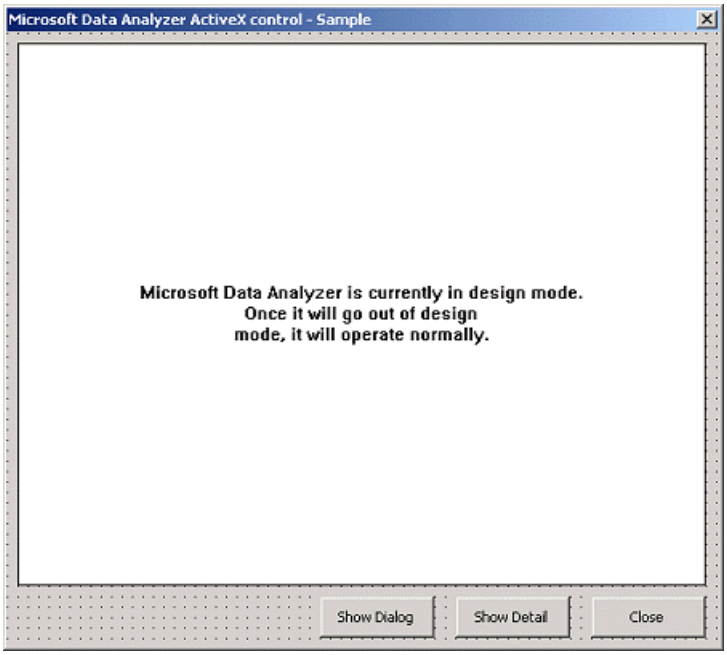


Figure 1. The frmControl form in Design View

The **Show Dialog** button will display the frmDialogs form from which the user can select from a list of actions. The **Show Detail** button will open a form that displays a tree view of the different objects (aspects, traits, and so forth) making up a Data Analyzer view. We will also add a **Close** button to the form.

1. In the **Toolbox**, drag a **CommandButton** control onto the UserForm in the lower right hand corner.
2. In the Properties window for the control, replace the text in the Name property with **cmdClose** and the text in the Caption property with **Close**.
3. Double-click the control to open up the cmdClose_Click event procedure. Between the **Sub** and the **End Sub** statements, type the following:

End

Executing this command will close the frmControl form.

4. In the **Toolbox**, drag a **CommandButton** control onto the form to the left of the **Close** button.
5. In the Properties window for the control, replace the text in the Name property with **cmdDetail** and the text in the Caption property with **Show Detail**.
6. Double-click the control to open up the cmdDetail_Click event procedure. Between the **Sub** statement and the **End Sub** statement, type the following:

frmDetail.Show

Executing this command will display the frmDetail form.

7. In the **Toolbox**, drag a **CommandButton** control onto the form to the left of the **Show Detail** button.
8. In the Properties window for the control, replace the text in the Name property with **cmdDialog** and the text in the Caption property with **Show Dialog**.
9. Double-click the control to bring up the cmdDialog_Click event procedure. In the procedure, type the following:

```
' Displays a list of action dialogs.
frmDialogs.Show vbModal
mobjMSDA.ShowDialog mintDialog
```

Clicking this button will display the frmDialogs form and execute the **ShowDialog** method of the Data Analyzer **Application** object. This will be explained in more detail later in this article. **Figure 2** shows how the frmControl form will appear when opened to a view.

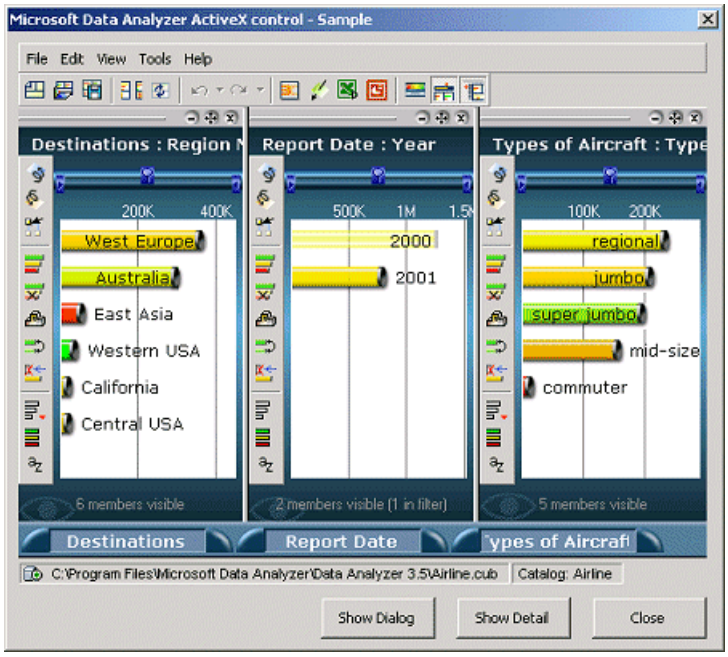


Figure 2. The frmControl Form at Run Time

Adding Code to the frmControl Form

Now, we will add some additional code to the frmControl form. First, we will add code that will allow our form to be resizable. This will allow users to change the size of the form to fit their screens. This code will also resize the Microsoft Data Analyzer ActiveX control and reposition the form's command buttons as the form is resized. To do this, we will need to make calls to the Microsoft Windows® API. Before we can use the procedures in the Windows API, we need to tell VBA where the DLLs containing those procedures can be found, what parameters they accept, and what information they return. We do this by using a **Declare** statement. In the Declarations section of the frmControl form's code window, type the following:

```
' Windows API calls used to make the frmControl form resizable.
'Find the form's window handle
   Private Declare Function FindWindow Lib "user32" _
   Alias "FindWindowA" ( ByVal lpClassName As String, ByVal lpWindowName As String) As Long
'Get the form's window style
   Private Declare Function GetWindowLong Lib "user32" _
   Alias "GetWindowLongA" ( ByVal hWnd As Long, ByVal nIndex As Long) As Long
'Set the form's window style
Private Declare Function SetWindowLong Lib "user32" _
   Alias "SetWindowLongA" ( ByVal hWnd As Long, ByVal nIndex As Long, ByVal dwNewLong As Long) As Long
'The offset of a window's style
   Private Const GWL_STYLE As Long = (-16)
'Style to add a sizable frame
   Private Const WS_THICKFRAME As Long = &H40000
```

A detailed discussion of these declarations is beyond the scope of this article. However, more information on Windows APIs can be found in the article Overview of the Windows API.

Next, in the frmControl form's code window, select the **Activate** event from the right drop-down list box at the top of the code window. The **Activate** event occurs when the form becomes the active window. Type the following code into the procedure:

```
' Purpose: Open up a view file selected by the user in the frmIntro form. Contains the form's Windows handle value.
Dim mhWndForm As Long
Dim iStyle As Long

On Error GoTo UserForm_Activate_Err
' Get the handle for the form (for Excel 2000 or later).
    mhWndForm = FindWindow("ThunderDFrame", frmControl.Caption)
'Make the form resizable
    iStyle = GetWindowLong(mhWndForm, GWL_STYLE)
    iStyle = iStyle Or WS_THICKFRAME
    SetWindowLong mhWndForm, GWL_STYLE, iStyle
' Set a reference to the Data Analyzer ActiveX control.
    Set mobjMSDA = Max3Ax1.Application()
```

```
' Open up the view.
    mobjMSDA.ActiveView.OpenView strFileName, vlocFileSystem
UserForm_Activate_End:
   Exit Sub
UserForm_Activate_Err:
   If Err.Number = -2147221394 Then
      MsgBox "Cannot open the View. Either the View file does not " & "exist or it is not a valid View.", vbOKOnly
      GoTo UserForm_Activate_End
   Else
      MsgBox "Error " & Err.Number & ": " & Err.Description, , "UserForm_Activate"
   End If
```

Examining this procedure, the mhWndForm variable is declared as a **Long** value and contains a handle (a unique number used by Windows to identify a window) to the active window (the frmControl form). The iStyle variable contains the **Style** property setting that Windows uses for this window. Then, we call the **FindWindow** method in the Windows API to get the handle for form. **ThunderDFrame** is the class name Windows uses to identify Excel 2002 and the **Caption** property contains the name of the form.

Next, we call the **GetWindowsLong** method in the Windows API to get the current **Style** property setting for the form. We perform an **Or** operation using that value with the **WS_THICKFRAME** constant and then use the result in a call to the **SetWindowLong** method in the Windows API to reset the **Style** property. This identifies the form as resizable.

We then create a reference to the Max3Ax **Application** object and open a view based on the strFileName **String** variable.

mobjMSDA.ActiveView.OpenView strFileName, vlocFileSystem

The strFileName **String** variable gets its value from the frmIntro form that we will discuss shortly.

We also include error-handling code in the event in case the view is not a valid Data Analyzer view.

Next, we will insert and examine the procedure that executes when the user resizes the form. In the frmControl form's code window, select the **Resize** event from the right drop-down list at the top of the code window and insert the following:

```
' Purpose: Maintains the size of the Data Analyzer control and the position of the three command buttons relative
' to the size of the frmControl form.
Dim ctlControl As Msforms.Control
' Const for the size and positions of the controls
' on the frmControl form.
Const intDAControlHeightOffset As Integer = 60
Const intDAControlWidthOffset As Integer = 20
Const intCmdButtonHeightOffset As Integer = 48
Const intCloseBtnWidthOffset As Integer = 78
Const intDetailBtnWidthOffset As Integer = 156
Const intDialogBtnWidthOffset As Integer = 234
For Each ctlControl In frmControl.Controls ' Loop through each control on the form and set size or' position.
   If ctlControl.Name = "Max3Ax1" Then
      ctlControl.Height = frmControl.Height - intDAControlHeightOffset
      ctlControl.Width = frmControl.Width - intDAControlWidthOffset
   Else
      ctlControl.Top = frmControl.Height - intCmdButtonHeightOffset
      Select Case ctlControl.Name
         Case "cmdClose"
            ctlControl.Left = frmControl.Width –  intCloseBtnWidthOffset
         Case "cmdShowDetail"
            ctlControl.Left = frmControl.Width –  intDetailBtnWidthOffset
         Case "cmdShowDialog"
            ctlControl.Left = frmControl.Width –  intDialogBtnWidthOffset
      End Select
   End If
Next
```

First, we declared a variable that can be used to represent each control on the form.

```
Dim ctlControl As Msforms.Control
```

Then, we declared a series of constants to define where the controls should be located on the form relative to the form's borders.

```
Const intDAControlHeightOffset As Integer = 60
Const intDAControlWidthOffset As Integer = 20
Const intCmdButtonHeightOffset As Integer = 48
Const intCloseBtnWidthOffset As Integer = 78
Const intDetailBtnWidthOffset As Integer = 156
Const intDialogBtnWidthOffset As Integer = 234
```

Next, we loop through the form's **Controls** collection. For each control on the form, we set the **Width** and **Height** properties to position the control to fit the form, based on an offset value.

```
For Each ctlControl In frmControl.Controls
   If ctlControl.Name = "Max3Ax1" Then
      ctlControl.Height = frmControl.Height - intDAControlHeightOffset
      ctlControl.Width = frmControl.Width - intDAControlWidthOffset
   Else
      ctlControl.Top = frmControl.Height - intCmdButtonHeightOffset
      Select Case ctlControl.Name
         Case "cmdClose"
            ctlControl.Left = frmControl.Width –  intCloseBtnWidthOffset
         Case "cmdShowDetail"
            ctlControl.Left = frmControl.Width –  intDetailBtnWidthOffset
         Case "cmdShowDialog"
            ctlControl.Left = frmControl.Width –  intDialogBtnWidthOffset
      End Select
   End If
Next
```

Now the controls will maintain size and position whenever the user changes the size of the form.

Creating and Adding Code to the frmIntro Form

The frmIntro form is the initial form displayed when Excel opens. The form contains text that instructs the user to open a Data Analyzer view file. When the user clicks **OK**, a **FileOpen** dialog box appears and the user can then navigate to and open a view file. Data Analyzer views can be contained in .max files or in .xml files. **Figure 3** shows the frmIntro form when it is completed.

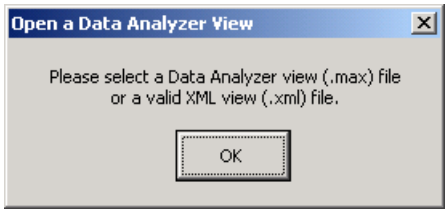


Figure 3. The frmIntro form when it is completed

Let's create the form and add code to it:

1. From the **Insert** menu, click **UserForm**.
2. In the Properties window for the UserForm, replace the text in the Name property with **frmIntro** and the text in the Caption property with **Open a Data Analyzer View**.
3. In the Toolbox, click the **Label** control and drag the control to the form.
4. In the Properties window for the control, replace the text in the Caption property with the following:

Please select a Data Analyzer view (.max) file or a valid XML view (.xml) file.

5. In the Toolbox, click the **TextBox** control and drag the control to the form under the label, centered in the form.
6. In the Properties window for the control, replace the text in the Name property with **cmdOK** and the text in the Caption property with **OK**.
7. Double-click the control to bring up the cmdOK_Click event procedure. Between the **Sub** statement and the **End Sub** statement, type the following:

```
' Purpose: This procedure opens a file open dialog and allows the user to choose a view or cube file.
'Dim FDialog As FileDialog
Dim FFilters As FileDialogFilters

On Error GoTo cmdOK_Click_Err
Set FDialog = Application.FileDialog(msoFileDialogOpen) ' Set up the File | Open dialog.
With FDialog
   Set FFilters = .Filters ' Set up the file filters.
   With FFilters
      .Clear
      .Add "Data Analyzer Views", "*.max"
      .Add "XML Views", "*.xml"
   End With
   .AllowMultiSelect = False ' Allow user to select just one file.
   If .Show = False Then ' Exit if the user selects CANCEL.
      GoTo cmdOK_Click_End
   End If
   strFileName = .SelectedItems(1) ' Assigned the file selection.
End With
Unload frmIntro
frmControl.Show
cmdOK_Click_End:
   Unload frmIntro
   Exit Sub
```

```
cmdOK_Click_Err:
    MsgBox "Error " & Err.Number & ": " & _
        Err.Description, , "cmdOK_Click"
```

This procedure uses the **FileDialog** object to display the **File Open** dialog box. First, we use the **FileDialog** property of the **Application** object to return a reference to the **FileDialog** object and then specify the type of dialog we want by using the **msoFileDialogOpen** constant. Next we use the **Filters** property of the **FileDialog** object to return a reference to the **FileDialogFilters** collection. The **FileDialogFilters** collection contains a number of preset filters. The **Clear** method of the **FileDialogFilters** collection removes any pre-existing filters. We then add our own filters to show only .max and .xml files. The **Show** method of the **FileDialog** object displays the **File Open** dialog box. If the user clicks the **Open** button, the **Show** method returns a value of **True**. If the user clicks the **Cancel** button, the **Show** method returns **False** and we exit the procedure.

Creating the frmDetail Form

The frmDetails form is used to display a hierarchical view of the current Data Analyzer view. It does this using the TreeView ActiveX control, which is available in the Microsoft TreeView Control, version 5.0 (SP2) OCX (Comctl32.ocx). The VBA code behind the form populates the TreeView control by using the collections, objects, and methods of the Data Analyzer API object model and the interfaces in the MaxODBO 1.0 Type Library. The Data Analyzer API object model provides access to the components that make up a view. This includes aspects (similar to dimensions in OLAP terminology), aspect members, qualities (similar to measures in OLAP terminology), and traits (how qualities are displayed in a view such as by length or by color). The interfaces in the MaxODBO 1.0 Type Library provide access to the objects, methods, and properties that can be used manipulate and gather information (metadata) on the OLAP objects (catalogs, cubes, dimensions, measures, hierarchies) from which the view is derived.

Figure 4 shows the frmDetail form when it is completed.

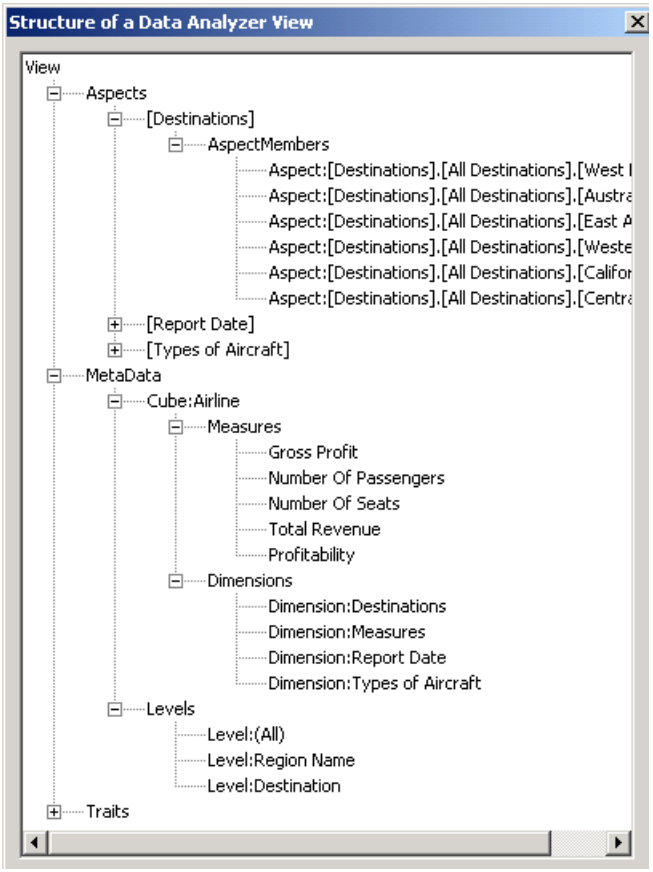


Figure 4. The frmDetail form when it is completed

Let's create the form and add code to it:

1. On the **Insert** menu, click **UserForm**.
2. In the Properties window for the UserForm, replace the text in the Name property with **frmDetail** and the text in the Caption property with **Structure of a Data Analyzer View**.
3. In the Toolbox, right-click, and click **Additional Controls**.
4. In the **Additional Controls** dialog box, check the **Microsoft TreeView Control, version 5.0 (SP2)** box, and then click **OK**.
5. Drag the TreeView control onto the frmDetail form.
6. Next, add a command button to the form. In the Toolbox, drag a **CommandButton** control onto the lower right hand corner of the frmDetail form.
7. In the Properties window for the control, replace the text in the Name property with **cmdClose** and the text in the Caption property with **Close**.
8. Double-click the control to open up the cmdClose_Click event procedure. Between the **Sub** and the **End Sub** statements, enter the following:

```
Unload frmDetail
```

9. Next, select the **Activate** event from the right drop-down list at the top of the code window. Type the following code into the frmDetail_Activate event procedure:

```
' The LoadView procedure contains routines that populate the TreeView control.
' Initialize the unique identifier that will be appended to the node name when needed.
intUniqueID = 0
LoadView
' Show first node of the TreeView control expanded.
TreeView1.Nodes.Item(1).Expanded = True
```

This code calls the LoadView procedure and expands the TreeView control to display the first node, once it is populated.

10. Type the following procedure into the form's code window:

```
Private Sub LoadView()
   'Purpose: Calls routines that populate the TreeView control with information on Aspects (dimensions) and Traits (measures).
      Set mobjView = mobjMSDA.ActiveView
   ' Add initial "View" level to the tree.
   TreeView1.Nodes.Add , , "Root", "View"
   LoadAspects
   LoadTraits
End Sub
```

The LoadView procedure creates a reference to the active Data Analyzer view by using the **ActiveView** property of the Data Analyzer **Application** object. The code then appends the initial node (named View) to the tree by using the TreeView control's **Add** method. Next, the LoadAspects and LoadTraits subroutines are called to populate the tree.

Adding Aspects and MetaData to the Tree

An aspect is a dimension in an OLAP cube. Examples of aspects include time, customers, products, and regions. As we will see in the next procedure, the **Aspects** collection of the Data Analyzer object model provides the **Count** property and **Item** method. These members are used to determine the number of **Aspect** objects in the collection and access individual **Aspect** objects.

Type the following procedure into the frmDetail form code window:

```
Private Sub LoadAspects()
   ' Purpose: Appends the Aspects label to tree and then calls routines to append individual aspect members. Also calls routines that append the cube metadata.
   Dim objAspects As Aspects
   Dim objAspect As Aspect
   Dim objHierarchy As IMdhHierarchy
   Dim i As Integer

   Set objAspects = mobjView.Aspects()
   TreeView1.Nodes.Add "Root", tvwChild, "Aspects", "Aspects"  ' Add an Aspects label node to the tree.
' Add information about each Aspect object to the tree. Then call the routines that add the aspect members to the tree.
   For i = 0 To objAspects.Count - 1
      Set objAspect = objAspects.Item(i)
      TreeView1.Nodes.Add "Aspects", tvwChild, objAspect.ID, objAspect.ID
      GetAspectMembers objAspect.Members, objAspect.ID ' Get the aspect members
   Next i
' Add an MetaData label node to the tree and then calls the routines to get the cube metadata.
   TreeView1.Nodes.Add "Root", tvwChild, "MetaData", "MetaData"
   Set objAspect = objAspects.Item(0)
   Set objHierarchy = objAspect.MDHHierarchy
   GetCube objHierarchy.Dimension.Cube, "MetaData"
End Sub
```

This procedure appends an Aspects label to the tree and then iterates through the **Aspects** collection to add each **Aspect** object (dimension) to the tree. Then, the GetAspectMembers subroutine is called to retrieve the members of the **Aspect** object. And finally, a MetaData label is appended to the tree and the GetCube subroutine is called to retrieve information from the cube. If you are unsure about how the nodes relate to each other in the tree, refer to **Figure 4**.

Adding Aspect Members

Next we will examine the GetAspectMembers subroutine. This procedure appends an AspectMembers label to the tree view and then iterates through the **AspectMembers** collection, and adds each **Aspect** object to the tree. Type the following code into the code window of the frmDetail form:

```
Private Sub GetAspectMembers(ByVal asptMembers As AspectMembers, ByVal Node As String)
   ' Purpose: Appends the aspect members to the tree.  Accepts: asptMembers - Collection of aspect members.
   'Node -       Branch ID of node where this  information will be appended.
   Dim keytext As String
   Dim i As Integer
   Dim objMember As AspectMember
   Dim immediateparent As String
   Const NODE_NAME_NOT_UNIQUE = 35602
```

```
    On Error GoTo GetAspectMembers_Err
    ' Add an AspectsMembers label node and then appends each aspect member to the tree.
    keytext = "AspectMembers"
    TreeView1.Nodes.Add Node, tvwChild, keytext, "AspectMembers"
    immediateparent = keytext
    For i = 0 To asptMembers.Count - 1
        Set objMember = asptMembers.Item(i)
        TreeView1.Nodes.Add immediateparent, tvwChild, keytext, "Aspect:" & objMember.ID
    Next i
GetAspectMembers_End:
    Exit Sub
GetAspectMembers_Err:
    If Err.Number = NODE_NAME_NOT_UNIQUE Then
        keytext = keytext & intUniqueID + 1
        Err.Clear
        Resume
    End If
End Sub
```

This procedure accepts the following arguments: asptMembers, which represents the **AspectMembers** collection, and the Node **String** variable that contains the ID of the parent node.

Notice the keytext **String** variable in the procedure. This variable is assigned the ID value of the node we are working with. The node ID must be unique so that the subroutine knows which node to append the information to. However, because the value for the node ID of the parent node may be the same as the value of the node ID for the child node, an error may occur when we try to append the node information to the tree. To overcome this, we've added some code to the error handling routine. The intUniqueID + 1 expression adds a value of 1 to the public counter variable intUniqueID. The value of the counter is then appended to the node ID (the value of the keytext variable) in order to make the node ID unique. The error is then cleared and the procedure resumes execution from where the error was generated. This procedure is repeated in other procedures in the form.

Adding Cube Information

The next procedure is called from the LoadAspects subroutine and adds cube information to the tree.

Type the following procedure into the frmDetail form code window:

```
Private Sub GetCube(ByVal mtdCube As IMdhCube, ByVal Node As String)
    ' Purpose: Appends the Cube label to the tree and then calls routines that append dimension and level information.
    ' Accepts: mtdCube - Contain information on the cube.
    'Node -  Branch ID of node where this information will be appended.
    Dim keytext As String
    Const NODE_NAME_NOT_UNIQUE = 35602
    On Error GoTo GetCube_Err
    keytext = "Cube:" & mtdCube.Name
    ' Adds label node and then calls the routines that
    ' append dimension and measure information.
    TreeView1.Nodes.Add Node, tvwChild, keytext, "Cube:" & mtdCube.Name
    GetMeasuresDimension mtdCube.MeasuresDimension, keytext
    GetDimensions mtdCube.Dimensions, keytext
GetCube_End:
    Exit Sub
GetCube_Err:
    If Err.Number = NODE_NAME_NOT_UNIQUE Then
        keytext = keytext & intUniqueID + 1
        Err.Clear
        Resume
    End If
End Sub
```

This procedure accepts the following arguments: mtdCube, which is an **IMdhCube** object that contains metadata for the **Cube** object, and the Node **String** which contains the ID of the parent node. The procedure appends a **Cube** label to the tree view and then calls the GetMeasuresDimension subroutine to get information on the measures associated with the cube. Then, the GetDimensions subroutine is called to retrieve information on the dimensions associated with the cube.

Adding Measures Information

The next procedure adds information on the **Measures** object to the tree. Type the following procedure into the frmDetail form code window:

```
Private Sub GetMeasuresDimension(ByVal mtdMeasuresDim As IMdhMeasuresDimension, ByVal Node As String)
    ' Purpose: Appends measures information to the tree.
    ' Accepts: mtdMeasureDim - Collection containing information on the measures in the cube.
    'Node -  Branch ID of node where this information will be appended.
```

```
    Dim keytext As String
    Dim i As Integer
    Dim objMeasure As IMdhMeasure
    Const NODE_NAME_NOT_UNIQUE = 35602
    On Error GoTo GetMeasuresDimension_Err
    ' Adds label node and then appends measures information to the tree.
    TreeView1.Nodes.Add Node, tvwChild, "Measures", "Measures"
    For i = 0 To mtdMeasuresDim.Measures.Count - 1
        Set objMeasure = mtdMeasuresDim.Measures.Item(i)
        keytext = objMeasure.Caption
        TreeView1.Nodes.Add "Measures", tvwChild, keytext, keytext
    Next i
GetMeasuresDimension_End:
    Exit Sub
GetMeasuresDimension_Err:
    If Err.Number = NODE_NAME_NOT_UNIQUE Then
        keytext = objMeasure.Caption & intUniqueID + 1
        Err.Clear
        Resume
    End If
End Sub
```

This procedure accepts the following arguments: mtdMeasuresDim, which represents an **IMdhMeasuresDimension** collection, which contains the **Measures** objects, and the Node **String**, which contains the ID of the parent node. The procedure appends a **MeasuresDimension** label to the tree and then iterates through the **IMdhMeasuresDimension** collection and adds information on each measure to the tree.

Adding Dimension Information

The following procedure appends a Dimensions label node to the tree and then iterates through the **IMdhDimensions** collection to add information on each **Dimension** object to the tree.

Type the following code to the frmDetail form code window:

```
Private Sub GetDimensions(ByVal mtdDimensions As IMdhDimensions, ByVal Node As String)
    ' Purpose: Appends the dimension information to the tree.
    ' Accepts: mtdDimension - Collection containing information  on the dimensions in the cube.
    'Node - Branch ID of node where this information will be appended.
    Dim keytext As String
    Dim i As Integer
    Dim objDim As IMdhDimension
    Dim immediateparent As String
    Const NODE_NAME_NOT_UNIQUE = 35602
    On Error GoTo GetDimensions_Err
    keytext = "Dimensions"
    ' Adds label node and then appends dimension information to the tree.
    TreeView1.Nodes.Add Node, tvwChild, keytext, "Dimensions"
    immediateparent = keytext
    For i = 0 To mtdDimensions.Count - 1
        Set objDim = mtdDimensions.Item(i)
        keytext = objDim.Caption
        TreeView1.Nodes.Add immediateparent, tvwChild, keytext, "Dimension:" & objDim.Caption
    Next i
GetDimensions_End:
    Exit Sub
GetDimensions_Err:
    If Err.Number = NODE_NAME_NOT_UNIQUE Then
        keytext = keytext & intUniqueID + 1
        Err.Clear
        Resume
    End If
End Sub
```

This procedure accepts the following arguments: mtdDimensions, which represents the **IMdhDimensions** collection, and the Node **String**, which contains the ID of the parent node.

Adding Trait Information

Traits in a Data Analyzer view are a way to graphically represent qualities to the user. A trait may be single (like length or color) and contain one quality value or multiple (like a grid) and contain a list of qualities.

Qualities are essentially the same as measures in OLAP terminology. Each view has a list of qualities, which are common to all aspects (for example, the measures shown in the Grid view are the same in all aspects).

Note
The term quality is used instead of measure because a quality is not necessarily an OLAP measure. Currently, other quality types are supported: template measures and member properties (caption) for example.

In Data Analyzer, the **TraitManager** object manages the traits of a view. The following procedure adds a Traits label node to the tree and then calls the GetTraitQualities subroutine, once for each type of trait.

Type the following procedure into the frmDetail form code window:

```
Private Sub LoadTraits()
    ' Purpose: Populates the TreeView control with Trait information; single traits (length, color) and multiple traits (grid).
    Dim objTM As TraitsManager

     Set objTM = mobjView.TraitsManager()
    TreeView1.Nodes.Add "Root", tvwChild, "Traits", "Traits"   ' Add Traits label to the tree.
    ' Routines which append the various traits.
    GetTraitQualities objTM.Trait(trtLength), "Traits", "Length"
    GetTraitQualities objTM.Trait(trtColor), "Traits", "Color"
    GetTraitQualities objTM.Trait(trtGrid), "Traits", "Grid"
End Sub
```

Next, we will examine the GetTraitQualities subroutine. This procedure appends the TraitQualities label node to the tree and then iterates through the **Qualities** collection of the **Trait** object to add information on each **Quality** object to the tree. Type the following procedure into the frmDetail form code window:

```
Private Sub GetTraitQualities(ByVal trtTrait As Trait, ByVal Node As String, ByVal trtType As String)
    ' Purpose: Retrieves the trait qualities such as profitability, total revenue, and so forth and appends them to the tree.
    ' Accepts: trtTrait - Trait such as length or color.
    'Node -  Branch ID of node where this information will be appended.
    ' trtType -  Type of trait such as length, color, or grid.
    Dim objQuals As Qualities
    Dim i As Integer
    Dim keytext As String
    Dim immediateparent As String
    Const NODE_NAME_NOT_UNIQUE = 35602

    On Error GoTo GetTraitQualities_Err
    keytext = "TraitQualities:" & trtType
' Qualities is a collection of Quality measures for each aspect such as Profitability and Total Revenue.
    Set objQuals = trtTrait.Qualities
' Add trait container (parent) node to tree.
    TreeView1.Nodes.Add Node, tvwChild, keytext, "TraitQualities:" & trtType
    immediateparent = keytext
' Add individual quality child node(s) to the tree.
    For i = 0 To objQuals.Count
        TreeView1.Nodes.Add immediateparent, tvwChild, keytext, "Quality:" & objQuals.QualityID(i)
    Next i

GetTraitQualities_End:
    Exit Sub

GetTraitQualities_Err:
    If Err.Number = NODE_NAME_NOT_UNIQUE Then
        keytext = keytext & intUniqueID + 1
        Err.Clear
        Resume
    End If
End Sub
```

This procedure accepts the following arguments: trtTrait, which represents the **Trait** object, and the Node **String**, which contains the ID of the parent node.

Creating the frmDialogs Form

The final form that we will create is the frmDialogs form. This form is used to present a list of operations that perform tasks also available from the user interface. For example, there is an **Open File** operation that displays the **Open File** dialog box, a **Save As File** dialog box, a **New View** dialog box, and a **Change View** dialog box. When the user selects an operation from the list, the appropriate dialog box is displayed. **Figure 5** shows the frmDialogs form when it is completed.

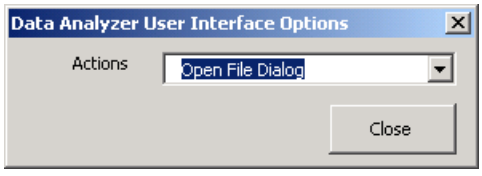


Figure 5. The frmDialogs form when it is completed

Let's create the form and add code to it:

1. On the **Insert** menu, click **UserForm**.
2. In the Properties window for the UserForm, replace the text in the Name property with **frmDialogs** and the text in the Caption property with **Data Analyzer User Interface Options**.
3. In the Toolbox, drag a **ComboBox** control to the form.
4. In the Properties window for the control, replace the text in the Name property with **cboDialogs** and click **2—fmStyleDropDownList** in the Style property to prevent users from typing in the drop-down list box.
5. In the Toolbox, drag a **CommandButton** control onto the lower right hand corner of the form.
6. In the Properties window for the control, replace the text in the Name property with **cmdOK** and the text in the Caption property with **OK**.
7. Double-click the control to bring up the cmdOK_Click event procedure. Between the **Sub** statement and the **End Sub** statement, add the following:

```
Unload frmDialogs
```

8. Click **cboDialogs** from the left drop-down list box at the top of the frmDialogs form code window. This will create the cboDialogs_Click event procedure. Type the following in that procedure:
9. mintDialog = cboDialogs.ListIndex + 1

In the left drop-down list box at the top of the frmDialogs form code window, click **UserForm**. This will create the frmDialogs_Activate event procedure. Type the following code in that procedure:

```
' Purpose: Populates the cboDialog combo box control.
With cboDialogs
    .AddItem "About Dialog"
    .AddItem "Open File Dialog"
    .AddItem "Save As File Dialog"
    .AddItem "HTML Report Dialog"
    .AddItem "New View Dialog"
    .AddItem "Change View Dialog"
    .AddItem "Drill Through Dialog"
    .AddItem "Export to XL Pivot Table Dialog"
    .AddItem "Open Using Connection Dialog"
    .AddItem "Business Center Dialog"
    .AddItem "Export to XL Static Dialog"
    .ListIndex = 1
End With
```

Now let's examine how this form works. When the user clicks **Show Dialog** on the frmControl form, the cmdDialog_Click event procedure executes and opens the frmDialogs form. The frmDialogs_Activate event procedure executes which populates the cboDialogs combo box. The user then clicks the cboDialogs combo box and selects one of the available options. This selection assigns the value of the cboDialogs **ListIndex** property to the mintDialog **Integer** variable. When the user closes the frmDialogs form, programmatic control returns to the cmdDialog_Click event procedure. The next statement executes the **ShowDialog** method of the Data Analyzer **Application** object and passes the ID of the user's selection (as contained in the mintDialog variable). The requested dialog box appears.

Running Code When the Workbook Opens

The only thing left is to add the code that will open the frmIntro form when the workbook is opened. To do this, we need to add a statement to the **Workbook_Open** subroutine:

1. From the Project Explorer window, double-click the **Workbook** icon.
2. In the left drop-down list box at the top of the code window, click **Workbook**. This will create the **Workbook_Open** event procedure.
3. Between the **Sub** statement and the **End Sub** statement, type the following statement:
4. frmIntro.Show

Now, when the workbook is opened, the frmIntro form will be displayed.

5. Save the workbook and close Excel.

Putting It All Together

Now that we created the forms we need and added the code, let's try out the sample.

1. Open the sample workbook file you just created in Excel.
2. Click **OK** in the **Open a View** dialog box. The **File Open** dialog box is displayed (see **Figure 6**).
3. In the **File Open** dialog box, navigate to a Data Analyzer view file and click **Open**. The **Microsoft Data Analyzer ActiveX Control—Sample** dialog box is displayed.

Hint

In a default installation of Data Analyzer, the Airline.max file is located at C:\Program Files\Microsoft Data Analyzer\Data Analyzer 3.5\Airline.max.

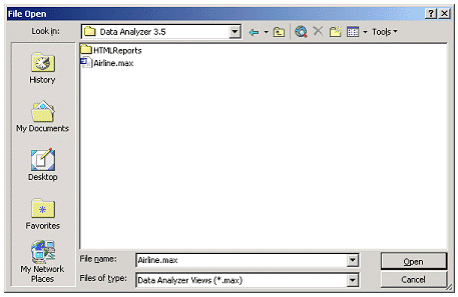


Figure 6. The File Open dialog box

You can now use and explore the features of the Data Analyzer just as if you were in the Data Analyzer application.

4. Click **Show Dialog**. In the **Data Analyzer User Interface Options** dialog box, click **About Dialog** in the drop-down list box. The Data Analyzer **About Dialog** dialog box appears (see **Figure 7**).
5. Click **OK**. The **Microsoft Data Analyzer ActiveX Control—Sample** dialog box appears.

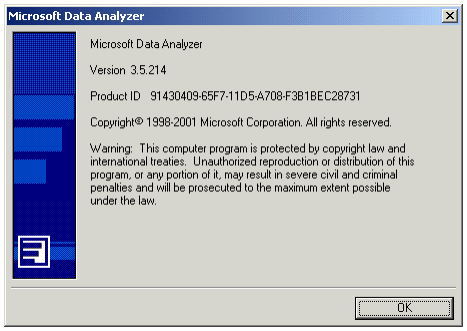


Figure 7. The Data Analyzer About Dialog box

6. Click **Show Detail**. The **Structure of a Data Analyzer View** dialog box appears. Click on the **+** icons to explore the different nodes of the tree. Click **Close**. The **Microsoft Data Analyzer ActiveX Control—Sample** dialog box appears.
7. Click **Close** to close the **Microsoft Data Analyzer ActiveX Control—Sample** dialog box.

Conclusion

In this article, we demonstrated embedding the Microsoft Data Analyzer ActiveX control into a UserForm in Excel 2002. We created the forms and sample code to connect the control to a Data Analyzer view file and display a dialog that allows the user to select from a list of actions normally available from the user interface. We also created a form and sample code that allows the user to display and populate a tree view of the different components that make up a Data Analyzer View. And lastly, we added code to the workbook so that a form is displayed when the workbook is opened.

By using the sample code and procedures in this article, you can embed the Data Analyzer ActiveX control into your applications and explore the structure of your data using the Data Analyzer object models.

Using the Microsoft Data Analyzer ActiveX Control in Web Pages

The Microsoft Data Analyzer is a member of the Microsoft Office family of software applications. Data Analyzer enables you to graphically mine your organization's multidimensional data. Solution developers can expose the core features of Data Analyzer as an ActiveX® control in a Visual Basic® for Applications (VBA) UserForm, as well as on a Web page. This article describes how to use the Data Analyzer ActiveX control in Web pages.

For more information about using the Data Analyzer ActiveX control in VBA UserForms, as well as an overview of the Data Analyzer ActiveX control's overall object model, see my earlier article Exploring Microsoft Data Analyzer Programmability.

For general information about Data Analyzer, see the Microsoft Data Analyzer Web site.

Development Approaches

There are two main approaches to using the Data Analyzer ActiveX control in Web pages. You can:

- Add a new instance of the control to a Web page, or edit an existing instance of the control in a Web page, using Microsoft FrontPage®.
- Create a Web Part containing an instance of the control in a dashboard created with SQL Server™ Digital Dashboard 3.0 or Microsoft SharePoint™ Portal Server 2001.

Note To view and interact with the Data Analyzer ActiveX control, a user must have a licensed copy of Data Analyzer installed on their computer. The Data Analyzer ActiveX control cannot be distributed separately from the full Data Analyzer product.

Before we examine both approaches, let's look at the code that is used to create and automate an instance of the Data Analyzer ActiveX control.

Examining the Code

Regardless of the development approach you take, here is the basic HTML code to create an instance of the Data Analyzer ActiveX control:

<object classid="clsid:E0ECA9C3-D669-4EF4-8231-00724ED9288F" d="Max3Ax1" height=100% width=100%></object>
You can also modify the control's startup behavior at design time by entering one or more <param> tags between the <object> and </object> tags as follows:

<param name="Member" value="Value">

Where "Member" is one of the following members, and "Value" is the corresponding member's value:

- **Application** Retrieves the **Application** object from the control. Although you can add a <param name="Application"> tag to the <object> tag, the **Application** method has no meaningful purpose at design time.
- **ExitCommandVisible** Determines whether the **Exit** command on the control's **File** menu is visible (value="-1"). The default value is to hide the **Exit** command (value="0"). Example:

  <param name="ExitCommandVisible" value="0">
- **InteractiveStartup** Determines whether the **Microsoft Data Analyzer Startup** dialog box is displayed when the page containing the control is opened (value="-1"). The default value is to hide the **Microsoft Data Analyzer Startup** dialog box (value="0"). Example:

  <param name="InteractiveStartup" value="0">
- **SupportIEHistory** If the control is viewed on a Web page hosted in Microsoft Internet Explorer 4.0 or later, this method determines whether the control retains its state when the user navigates away from the Web page containing the control and back again, which is the default (value="-1"). To disregard the control's state during page navigation, set value="0". Example:

  <param name="SupportIEHistory" value="-1">
- **ViewData** When the value of this method is set to a valid XML string, this method sets the control's view to the contents of the XML. If a value is specified for this method, it overrides any value provided for the **ViewURL** method. Although you can add a <param name="ViewURL"> tag to the <object> tag, the **ViewData** method has no meaningful purpose at design time (unless you were to handcraft a valid XML string containing the entire view's visual representation!).
- **ViewURL** Loads the view specified in the value parameter when the control is first displayed on the Web page. If a value is also provided for the **ViewData** method, the value of the **ViewURL** method is ignored. Example:

  *<param name="ViewURL" value="C:\Program Files\Microsoft Data Analyzer*Data Analyzer 3.5\Airline.max">

Note
Contrary to its name, the **ViewURL** method cannot open views using HTTP URLs. It can only open views using file system paths or a valid XML string that conforms to the Data Analyzer view schema.

Once you add the Data Analyzer ActiveX control to a Web page, you can write script against it using the id value provided in the <object> tag (in this case, Max3Ax1). For example, the following HTML code opens a view:

```
<script id=clientEventHandlersVBS language=vbscript>
<!--
Sub window_onload
   'The number 1 represents the constant vlocFileSystem(corresponds to a file in the file system).
   Max3Ax1.Application.ActiveView.OpenView _
      "C:\Program Files\Microsoft Data Analyzer\Data Analyzer 3.5\Airline.max", 1
End Sub
-->
</script>
```

Note
Similar to the **ViewURL** method, the **OpenView** method cannot open views using HTTP URLs. It can only open views using file system paths or a valid XML string that conforms to the Data Analyzer view schema.

The following code hides the main toolbar, the main menu, and the status bar from the current view:

```
<script id=clientEventHandlersVBS language=vbscript>
<!--
Sub window_onload
   With Max3Ax1.Application
      .ActiveView.OpenView _
         "C:\Program Files\Microsoft Data Analyzer\Data Analyzer 3.5\Airline.max", 1
      'Hide the main toolbar, main menu, and status bar.
      .MainToolbar.Bands.Item("MainToolBar").Visible = False
      .MainToolbar.Bands.Item("MainMenu").Visible = False
      .MainToolbar.Bands.Item("Main.StatusBar").Visible = False
   End with
End Sub
-->
</script>
```

Adding the Control to a Microsoft FrontPage 2002 Web Page or a Web Site Based on SharePoint Team Services

To add the Data Analyzer ActiveX control to a Microsoft FrontPage 2002 Web page or a Web site based on SharePoint™ Team Services from Microsoft, you must have Data Analyzer installed on the same computer from which you perform the following steps:

1. Start FrontPage 2002.
2. Do one of the following:
   - On the **File** menu, point to **New**, and click **Page or Web**. In the **New Page or Web** task pane, click **Blank Page**.
   - On the **File** menu, click **Open**, and select an existing Web page.
   - On the **File** menu, click **Open Web**, and select an existing FrontPage Web or a Web based on SharePoint Team Services. Once the Web is open, on the **View** menu, click **Folders**, and select an existing Web page.
3. Position your insertion point at the place on the Web page that you want to insert the Data Analyzer ActiveX control.
4. On the **Insert** menu, click **Web Component**.
5. In the **Component Type** pane, click **Advanced Controls**.
6. In the **Choose a control** pane, click **ActiveX Control**, and click **Next**.
7. Click **Customize**.
8. In the **Control** list, check the **Max3Ax Class** box, and click **OK**.
9. In the **Choose a control** list, click **Max3Ax Class**, and click **Finish**.
10. Switch to HTML view to modify the code that FrontPage generates.
11. Switch to Preview view to view and interact with the results. Notice that you can use the Data Analyzer user interface just as you would if you had started Data Analyzer from your **Programs** menu (you may need to switch back to Normal view to resize the Data Analyzer user interface).

Creating an Instance of the Control in a SQL Server Digital Dashboard 3.0 Dashboard

To manually add the Data Analyzer ActiveX control to a dashboard based on SQL Server Digital Dashboard 3.0. This procedure assumes that you have both Data Analyzer and the Digital Dashboard Resource Kit installed on an appropriate Web server (for more information on Digital Dashboard installation, see the Microsoft Digital Dashboard Web site).

1. Navigate to your DDRK Web site. The URL for this site will generally take the form **http://*WebServerName*/Dashboard**.
2. Click the **Administration** link in the top link bar.
3. In the **Dashboard View** pane, expand **DAVCatalog**, click **Parts**, and click **New**.
4. In the **Dashboard Properties** pane, in the **Name** box, type **DataAnalyzer**, and in the **Display Name** box, type **Data Analyzer**.
5. In the **Web Part List** pane, click **New**.
6. In the **Web Part Properties** pane, on the **General** tab, in the **Name** box, type **DataAnalyzer**, and in the **Display Name** box, type **Data Analyzer**.
7. On the **Advanced** tab, in the **Content Type** box, click **HTML**.
8. In the **Embedded Content** box, type the code to represent the Data Analyzer ActiveX control, and then click **Save**. For example:

```
<object classid="clsid:E0ECA9C3-D669-4EF4-8231-00724ED9288F"
id="Max3Ax1" height=325 width=745>
<param name="InteractiveStartup" value="0">
<param name="ViewURL" value="C:\Program Files\Microsoft Data
Analyzer\Data Analyzer 3.5\airline.max">
<param name="SupportIEHistory" value="-1">
<param name="ExitCommandVisible" value="0">
</object>
```

9. In the **Dashboard View** pane, click **Go** to see the dashboard.
10. To export this Web Part for use in other dashboards, in the **Web Part List** pane, click the Data Analyzer Web Part. In the **Web Part Properties** pane, click **Export**. Browse to a location where you want to store the Web Part, and click **Save**.

Creating an Instance of the Control for Use in a Microsoft SharePoint Portal Server 2001 Web Site

These steps assume that you have Data Analyzer and Microsoft SharePoint Portal Server 2001 installed on an appropriate Web server and have rights to publish on that server.

1. Navigate to your SharePoint Portal Server workspace. The URL for this site will generally take the form **http://*WebServerName*/*WorkspaceName***.
2. Click the **Management** tab in the top-most link bar.
3. Click **Create a new personal dashboard**.
4. Leave all of the default settings, and click **Save**.
5. Click **Content** in the upper right corner of your new personal dashboard.
6. Click **Create a New Web Part**.
7. In the **Name** box, type **Data Analyzer**.
8. Type the code to represent the Data Analyzer ActiveX control in the **Embedded content** box, and then click **Save**. For example:

```
<object classid="clsid:E0ECA9C3-D669-4EF4-8231-00724ED9288F"
id="Max3Ax1" width=325 height=745>
<param name="InteractiveStartup" value="0">
```

<param name="ViewURL" value="C:\Program Files\Microsoft Data Analyzer\Data Analyzer 3.5\airline.max">
<param name="SupportIEHistory" value="-1">
<param name="ExitCommandVisible" value="0">
</object>

9. Click **Save** again to view your dashboard.
10. To export this Web Part for use in other dashboards, click **Content** in the upper right corner of your new personal dashboard. In the **Web Parts** area, click the entry that corresponds to the Web Part you want to export, and then click **Export**. Browse to a location where you want to save the Web Part, and click **Save**.

Conclusion

In this article, you were shown how to add an instance of the Data Analyzer ActiveX control to a Microsoft FrontPage 2002 Web page, a Web site based on SharePoint Team Services, a SQL Server Digital Dashboard 3.0 dashboard, and a SharePoint Portal Server 2001 workspace. You were also shown how to script against the control to extend the control's features. Using the sample code and procedures in this article, you can now add robust data analysis features to your Web sites using the Data Analyzer ActiveX control.

Working with Microsoft Excel Objects

The Microsoft® Excel object model contains several dozen objects that you can manipulate through Microsoft® Visual Basic® for Applications (VBA) code. Almost anything you can do with Excel from its user interface, you can do by manipulating its objects through VBA. In addition, you can do things through VBA that can't be done through the user interface.

When you use VBA to work with Excel objects, from either within Excel itself or another Office application, you have access to every part of Excel. The objects you will work with include cells, ranges, sheets, workbooks, charts, and more. In other words, every element in Excel can be represented by an object that you can manipulate through VBA.

There are four Excel objects you will work with more than any others: the Application object, the Workbook object, the Worksheet object, and the Range object.

In This Section

Understanding the Excel Application Object
Use the Microsoft® Excel Application object to determine or specify application-level properties or execute application-level methods.

Understanding the Workbook Object
Work with the Workbook object to use with a single Microsoft® Excel workbook, and use the Workbooks collection to work with all currently open Workbook objects.

Understanding the Worksheet Object
Use a worksheet, containing a grid of cells, to work with data and hundreds of properties, methods, and events.

Understanding the Range Object
Develop a full understanding of the Range object and how to use it effectively in Microsoft® Visual Basic®for Applications (VBA) procedures and harness the power of Microsoft®Excel.

Understanding the Excel Application Object

The Microsoft® Excel Application object is the top-level object in Excel's object model. You use the Application object to determine or specify application-level properties or execute application-level methods. The Application object is also the entry point into the rest of the Excel object model.

Figure 1. Microsoft Excel Object Model (page 1)

Figure 2. Microsoft Excel Object Model (page 2)

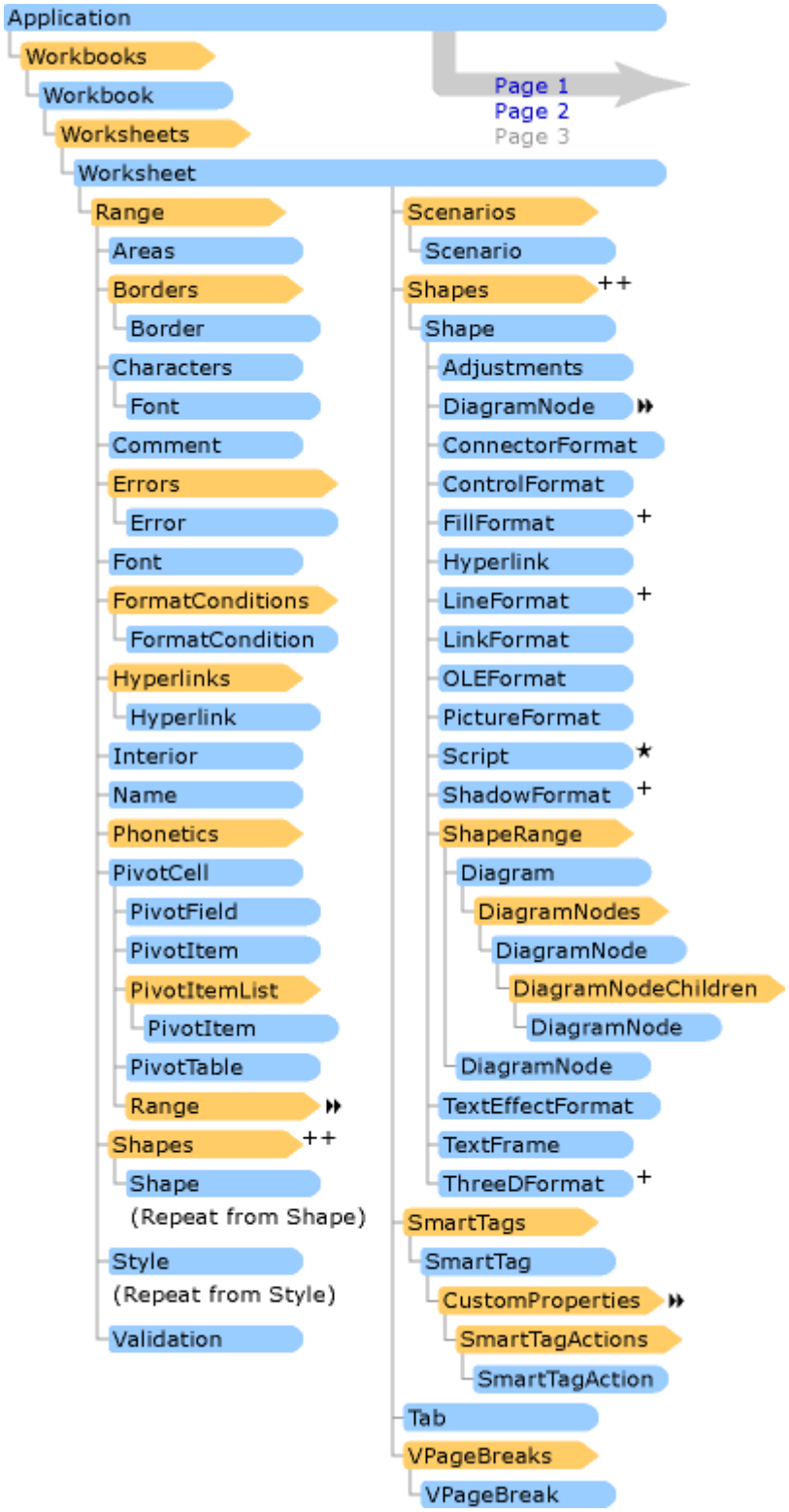


Figure 3. Microsoft Excel Object Model (page 3)

When you work with properties and methods of the Application object by using Microsoft® Visual Basic® for Applications (VBA) from within Excel, the Application object is available to you by default. This is known as an implicit reference to the object. If you work with Excel objects from another Office application, then you must create an object variable representing the Excel Application object. This is known as an explicit reference to the object. For example, the following two procedures return the name of the currently active Worksheet object. The ShowNameFromInsideXL procedure is designed to work from within Excel and uses an implicit reference to the Application object. In other words, it references the ActiveSheet property of the Application object without explicitly referencing the Application object itself. The ShowNameFromOutsideXL procedure is designed to be run from outside Excel and so must use an explicit reference to the Application object.

```
Sub ShowNameFromInsideXL()
  MsgBox """ & ActiveSheet.Name & "' is the currently active worksheet."
End Sub

Sub ShowNameFromOutsideXL()
  Dim xlApp As Excel.Application
  Const XL_NOTRUNNING As Long = 429
  On Error GoTo ShowName_Err
```

```
  Set xlApp = GetObject(, "Excel.Application")
  MsgBox """ & ActiveSheet.Name & "' is the currently active worksheet."
  xlApp.Quit
  Set xlApp = Nothing

  ShowName_End:
  Exit Sub
ShowName_Err:
  If Err = XL_NOTRUNNING Then
    ' Excel is not currently running.
    Set xlApp = New Excel.Application
    xlApp.Workbooks.Add
    Resume Next
  Else
    MsgBox Err.Number & " - " & Err.Description
  End If
  Resume ShowName_End
End Sub
```

Shortcuts to Active Objects

As with other Microsoft® Office XP application object models, the Microsoft® Excel Application object exposes several properties you can use to work with a currently active Excel object. For example, you often will write Microsoft® Visual Basic® for Applications (VBA) procedures designed to work with information in the currently selected cell, or with the currently active worksheet. The Application object exposes the ActiveCell, ActiveChart, ActivePrinter, ActiveSheet, ActiveWindow, and ActiveWorkbook properties, which you can use to return a reference to the currently active cell, chart, printer, sheet, window, or workbook. The following examples illustrate various ways you might use some of these properties:

```
' ActiveWorkbook property example:
Function SaveBookAs(strFileName As String) As Boolean
  ActiveWorkbook.SaveAs ActiveWorkbook.Path & "\" & strFileName
End Function

' ActiveCell property example:
Function CustomFormatCell()
  With ActiveCell
    If IsNumeric(.Text) And .Formula < 0 Then
      With .Font
        .Bold = True
        .Italic = True
      End With
      .Borders.Color = 255
    End If
  End With
End Function
' ActiveSheet property example:
Function ChangeName(strNewName As String) As Boolean
  ActiveSheet.Name = strNewName
End Function
```

In addition to the ActiveWorkbook property, you can use the Application object's Workbooks and Worksheets properties to return equivalent Excel objects. The Workbooks property returns the Workbooks collection that contains all the currently open Workbook objects. The Worksheets property returns the Sheets collection associated with the currently active workbook. The following example uses the Workbooks property to determine if a workbook is already open, and if not, to open it:

```
Function OpenBook(strFilePath As String) As Boolean
  ' This procedure checks to see if the workbook  specified in the strFilePath argument is open.
  ' If it is open, the workbook is activated. If it is  not open, the procedure opens it.
  Dim wkbCurrent      As Excel.Workbook
  Dim strBookName     As String

  On Error GoTo OpenBook_Err
  ' Determine the name portion of the strFilePath argument.
  strBookName = NameFromPath(strFilePath)
  If Len(strBookName) = 0 Then Exit Function
  If Workbooks.Count > 0 Then
    For Each wkbCurrent In Workbooks
      If UCase$(wkbCurrent.Name) = UCase$(strBookName) Then
        wkbCurrent.Activate
        Exit Function
      End If
    Next wkbCurrent
```

```
    End If
    Workbooks.Open strBookName
    OpenBook = True
    OpenBook_End:
    Exit Function
OpenBook_Err:
    OpenBook = False
    Resume OpenBook_End
End Function
```

Note
In the preceding example, the OpenBook procedure calls a custom procedure named NameFromPath that returns the file name portion of the full path and file name passed to the OpenBook procedure in the strFilePath argument.

Understanding the Workbook Object

In the Microsoft® Excel object model, the Workbook object appears just below the Application object. The Workbook object represents an Excel .xls or .xla workbook file. You use the Workbook object to work with a single Excel workbook. You use the Workbooks collection to work with all currently open Workbook objects.

You can also use the Application object's ActiveWorkbook property to return a reference to the currently active workbook. The Workbooks collection has a Count property you can use to determine how many visible and hidden workbooks are open. By default, Excel typically has one hidden workbook named Personal.xls. The Personal.xls workbook is created by Excel as a place to store macros. If the hidden Personal.xls workbook is the only open workbook, the ActiveWorkbook property returns Nothing, but the Workbooks collection's Count property returns 1. The Workbooks collection's Count property will return 0 only when there are no hidden or visible open workbooks.

Creating, Saving, Opening, and Closing Workbook Objects

You create a new Workbook object by using the Workbooks collection's Add method. The Add method not only creates a new workbook, but also immediately opens the workbook as well. The Add method also returns an object variable that represents the new workbook just created. The new workbook will contain the number of worksheets specified in the Sheets In New Workbook dialog box on the General tab of the Options dialog box (Tools menu). You can also specify the number of sheets a new workbook will have by using the Application object's SheetsInNewWorkbook property.

You can save a new workbook by using the Workbook object's SaveAs method and specifying the name of the workbook you want to save. If a workbook by that name already exists, an error occurs. When a workbook has been saved by using the SaveAs method, additional changes are saved by using the Workbook object's Save method. You can also save a copy of an existing workbook with a different file name by using the SaveCopyAs method. You can supply a file name to be used with the SaveAs or SaveCopyAs method, or you can use the Application object's GetSaveAsFileName method to let the user supply the name to be used to save the workbook. If the user clicks Cancel in the Save As dialog box, the GetSaveAsFileName method returns False.

Before you save a new workbook by using the SaveAs method, the Workbook object's Name property setting is a value assigned by Excel, such as Book1.xls. After you save the workbook, the Name property contains the name you supplied in the Filename argument of the SaveAs method. The Name property is read-only; to change the name of a workbook, you must use the SaveAs method again, and pass a different value in the Filename argument.

Note
A Workbook object's FullName property contains the object's path and file name, whereas the Path property contains only the saved path to the current workbook. Before a new workbook is saved, the FullName property has the same as the Name property, and the Path property has no value.

The Workbooks collection's Open method opens an existing workbook. When you open a workbook by using the Open method, it also becomes the active workbook. You can supply a file name to be used with the Open method, or you can use the Application object's GetOpenFileName method to let the user select the workbook to open. If the user clicks Cancel in the Open dialog box, the GetOpenFileName method returns False.

You use a Workbook object's Close method to close an open workbook. To specify whether pending changes to the workbook should be saved before the object is closed, you use the SaveChanges argument. If the SaveChanges argument is omitted, the user is prompted to save pending changes. You can also use the Close method of the Workbooks object to close all open workbooks. If there are unsaved changes to any open workbook when this method is used, the user is prompted to save changes. If the user clicks Cancel in this Save dialog box, an error occurs. You can suppress this Save dialog box by setting the Application object's DisplayAlerts property to False before executing the Close method. When you use the Workbooks object's Close method in this manner, any unsaved changes to open workbooks are lost. After the Close method has run, remember to set the DisplayAlerts property to True.

Note
The Auto_Open and Auto_Close procedures are ignored when a workbook is opened or closed by using the Open or Close methods. You can force these procedures to run by using the Workbook object's RunAutoMacros method. The Microsoft®

Visual Basic® for Applications (VBA) code in a workbook's Open and BeforeClose event procedures will be executed when the workbook is opened or closed by using the Open or Close methods.

The following example illustrates how to create a new workbook and specify the number of worksheets it will have:

```
Function CreateNewWorkbook(Optional strBookName As String = "",  Optional intNumSheets As Integer = 3) As Workbook
   ' This procedure creates a new workbook file and saves it by using the path and name specified in the strBookName
argument. You use the intNumsheets  argument to specify the number of worksheets in the workbook;  the default is 3.
   Dim intOrigNumSheets      As Integer
   Dim wkbNew                As Excel.Workbook

   On Error GoTo CreateNew_Err
   intOrigNumSheets = Application.SheetsInNewWorkbook
   If intOrigNumSheets <> intNumSheets Then
      Application.SheetsInNewWorkbook = intNumSheets
   End If
   Set wkbNew = Workbooks.Add
   If Len(strBookName) = 0 Then strBookName = Application.GetSaveAsFilename
   wkbNew.SaveAs strBookName
   Set CreateNewWorkbook = wkbNew
   Application.SheetsInNewWorkbook = intOrigNumSheets
CreateNew_End:
   Exit Function
CreateNew_Err:
   Set CreateNewWorkbook = Nothing
   wkbNew.Close False
   Set wkbNew = Nothing
   Resume CreateNew_End
End Function
```

Note
A Workbook object's Saved property is a Boolean value indicating whether the workbook has been saved. The Saved property will be True for any new or opened workbook where no changes have been made and False for a workbook that has unsaved changes. You can set the Saved property to True. Doing this prevents the user from being prompted to save changes when the workbook closes but does not actually save any changes made since the last time the workbook was saved by using the Save method.

A Note About Working with Workbooks Through Automation

When you are using Automation to edit an Excel workbook, keep the following in mind.

Creating a new instance of Excel and opening a workbook results in an invisible instance of Excel and a hidden instance of the workbook. Therefore, if you edit the workbook and save it, the workbook is saved as hidden. The next time the user opens Excel manually, the workbook is invisible and the user has to click Unhide on the Window menu to view the workbook.

To avoid this behavior, your Automation code should unhide the workbook before editing it and saving it. Note that this does not mean Microsoft® Excel itself has to be visible.

Understanding the Worksheet Object

Most of the work you will do in Microsoft® Excel will be within the context of a worksheet. A worksheet contains a grid of cells you can use to work with data and hundreds of properties, methods, and events you can use to work with the data in a worksheet.

To work with the data contained in a worksheet, in a cell or within a range of cells, you use a Range object. The Worksheet and Range objects are the two most basic and most important components of any custom application you create within Excel.

The Workbook object's Worksheets property returns a collection of all the worksheets in the workbook. The Workbook object's Sheets property returns a collection of all the worksheets and chart sheets in the workbook.

Each Excel workbook contains one or more Worksheet objects and can contain one or more chart sheets as well. Charts in Excel are either embedded in a worksheet or contained on a chart sheet. You can have only one chart on a chart sheet, but you can have multiple charts on a worksheet. Each embedded chart on a worksheet is a member of the Worksheet object's ChartObjects collection. Worksheet objects are contained in the Worksheets collection, which you can access by using the Workbook object's Worksheets property. When you use Microsoft® Visual Basic® for Applications (VBA) to create a new workbook, you can specify how many worksheets it will contain by using the Application object's SheetsInNewWorkbook property.

Referring to a Worksheet Object

Because a Worksheet object exists as a member of a Worksheets collection, you refer to a worksheet by its name or its index value. In the following example, both object variables refer to the first worksheet in a workbook:

```
Sub ReferToWorksheetExample()
  ' This procedure illustrates how to programmatically refer to  a worksheet.
  Dim wksSheetByIndex     As Excel.Worksheet
  Dim wksSheetByName      As Excel.Worksheet

  With ActiveWorkbook
    Set wksSheetByIndex = Worksheets(1)
    Set wksSheetByName = Worksheets("Main")
    If wksSheetByIndex.Index = wksSheetByName.Index Then
      MsgBox "The worksheet indexed as #" & wksSheetByIndex.Index & vbCrLf  & "is the same as the worksheet named '" _
        & wksSheetByName.Name & "'", vbOKOnly, "Worksheets Match!"
    End If
  End With
End Sub
```

Note

You can also use the Application object's ActiveSheet property to return a reference to the currently active worksheet in the currently active workbook.

You can use the Microsoft®Visual Basic®for Applications (VBA) Array function to work with multiple worksheets at the same time, as shown in the following example:

```
Sub ReferToMultipleSheetsExample()
  ' This procedure shows how to programmatically refer to multiple worksheets.
  Dim wksCurrent As Excel.Worksheet

  With ActiveWorkbook.Worksheets(Array("Employees", "Sheet2", "Sheet3"))
    .FillAcrossSheets (Worksheets("Employees").UsedRange)
  End With
  Stop
  ' The worksheets named "Sheet2" and "Sheet3" should now  contain the same table that is found on the "Employees"
  ' sheet. Press F5 to clear the contents from these worksheets.
  For Each wksCurrent In ActiveWorkbook _
    .Worksheets(Array("Sheet2", "Sheet3"))
    wksCurrent.UsedRange.Clear
  Next wksCurrent
End Sub
```

You can specify or determine the name of a worksheet by using its Name property. To change the name of a new worksheet, you first add it to the Worksheets collection and then set the Name property to the name you want to use.

Adding, Deleting, Copying, and Moving a Worksheet Object

You can add one or more worksheets to the Worksheets collection by using the collection's Add method. The Add method returns the new Worksheet object. If you add multiple worksheets, the Add method returns the last worksheet added to the Worksheets collection. If the Before or After arguments of the Add method are omitted, the new worksheet is added before the currently active worksheet. The following example adds a new worksheet before the active worksheet in the current collection of worksheets:

```
Dim wksNewSheet As Excel.Worksheet

Set wksNewSheet = Worksheets.Add
With wksNewSheet
  ' Work with properties and methods of the
  ' new worksheet here.
End With
```

You use the Worksheet object's Delete method to delete a worksheet from the Worksheets collection. When you try to programmatically delete a worksheet, Microsoft® Excel will display a message (alert); to suppress the message, you must set the Application object's DisplayAlerts property to False, as illustrated in the following example:

```
Function DeleteWorksheet(strSheetName As String) As Boolean
  On Error Resume Next
  Application.DisplayAlerts = False
  ActiveWorkbook.Worksheets(strSheetName).Delete
  Application.DisplayAlerts = True
  ' Return True if no error occurred;
  ' otherwise return False.
  DeleteWorksheet = Not CBool(Err.Number)
End Function
```

Note

When you set the DisplayAlerts property to False, always set it back to True before your procedure has finished executing, as shown in the preceding example.

You can copy a worksheet by using the Worksheet object's Copy method. To copy a worksheet to the same workbook as the source worksheet, you must specify either the Before or After argument of the Copy method. You move a worksheet by using the Worksheet object's Move method. For example:

Worksheets("Sheet1").Copy After:=Worksheets("Sheet3")
Worksheets("Sheet1").Move After:=Worksheets("Sheet3")

The next example illustrates how to move a worksheet so that it is the last worksheet in a workbook:
Worksheets("Sheet1").Move After:=Worksheets(Worksheets.Count)

Note
When you use either the Copy or the Move method, if you do not specify the Before or After argument, Excel creates a new workbook and copies the specified worksheet to it.

Understanding the Range Object

In Microsoft® Excel, the Range object is the most powerful, dynamic, and often-used object. When you develop a full understanding of the Range object and how to use it effectively in Microsoft® Visual Basic® for Applications (VBA) procedures, you will be well on your way to harnessing the power of Excel.

The Excel Range object is somewhat unique in terms of objects. In most cases, an "object" is a thing with some clearly identifiable corollary in the Excel user interface. For example, a Workbook object is recognizable as an .xls file. In a workbook, the collection of Worksheet objects is represented in the user interface by separate tabbed sheets. But the Range object is different. A range can be a different thing in different circumstances. A Range object can be a single cell or a collection of cells. It can be a single object or a collection of objects. It can be a row or column, and it can represent a three-dimensional collection of cells that span multiple worksheets. In addition, unlike other objects that exist as objects and as members of a collection of objects, there is no Ranges collection containing all Range objects in a workbook or worksheet. It is probably easiest to think of the Range object as your handle to the thing you want to work with.

In This Section

The Range Property
Use the Range property to return a Range object in many different circumstances.

The ActiveCell and Selection Properties
Learn how the ActiveCell property returns a Range object representing the currently active cell and the Selection property returns a Range object representing all the cells within the current selection when a cell or group of cells is selected.

Using the CurrentRegion and UsedRange Properties
Use the CurrentRegion and UsedRange properties to work with a range of cells whose size you have no control over.

Using the Cells Property
Understand how the Cells property loops through a range of cells on a worksheet or refers to a range by using numeric row and column values.

Using the Offset Property
Use the Offset property to return a Range object with the same dimensions as a specified Range object but offset from the specified range.

The Range Property

You will use the Range property to return a Range object in many different circumstances. The Application object, the Worksheet object, and the Range object all have a Range property. The Application object's Range property returns the same Range object as that returned by the Worksheet object. In other words, the Application object's Range property returns a reference to the specified cell or cells on the active worksheet. The Range property of the Range object has a subtle difference that is important to understand. Consider the following example:

Dim rng1 As Range
Dim rng2 As Range
Dim rng3 As Range

Set rng1 = Application.Range("B5")
Set rng2 = Worksheets("Sheet1").Range("B5")
Set rng3 = rng2.Range("B5")

The three Range objects do not all return a reference to the same cell. In this example, rng1 and rng2 both return a reference to cell B5. But rng3 returns a reference to cell C9. This difference occurs because the Range object's Range property returns a reference relative to the specified cell. In this case, the specified cell is B5. Therefore, the "B" means that the reference will be one column to the right of B5, and the "5" means the reference will be the fifth row below the row specified by B5. In other words, the Range object's Range property returns a reference to a cell that is n columns to the right and y rows down from the specified cell.

Typically, you will use the Range property to return a Range object, and then use the properties and methods of that Range object to work with the data in a cell or group of cells. The following table contains several examples illustrating usage of the Range property.

To	Use this code
Set the value of cell A1 on Sheet1 to 100	Worksheets("Sheet1").Range("A1").Value = 100
Set the value for a group of cells on the active worksheet	Range("B2:B14").Value = 10000
Set the formula for cell B15 on the active worksheet	Range("B15").Formula = "=Sum(B2:B14)"
Set the font to bold	Range("B15").Font.Bold = True
Set the font color to green	Range("B15").Font.Color = RGB(0, 255, 0)
Set an object variable to refer to a single cell	Set rngCurrent = Range("A1")
Set an object variable to refer to a group of cells	Set rngCurrent = Range("A1:L1")
Format all the cells in a named range	Range("YTDSalesTotals").Font.Bold = True
Set an object variable to a named range	Set rngCurrent = Range("NovemberReturns")
Set an object variable representing all the used cells on the Employees worksheet	Set rngCurrent = Worksheets("Employees").UsedRange
Set an object variable representing the group of related cells that surround the active cell	Set rngCurrent = ActiveCell.CurrentRegion
Set an object variable representing the first three columns in the active worksheet	Set rngCurrent = Range("A:C")
Set an object variable representing rows 3, 5, 7, and 9 of the active worksheet	Set rngCurrent = Range("3:3, 5:5, 7:7, 9:9")
Set an object variable representing multiple noncontiguous groups of cells on the active sheet	Set rngCurrent = Range("A1:C4, D6:G12, I2:L7")
Remove the contents for all cells within a specified group of cells (B5:B10) while leaving the formatting intact	Range("B5", "B10").ClearContents

As you can see from the examples in the preceding table, the Cell argument of the Range property is either an A1-style string reference or a string representing a named range within the current workbook.

You will also use the Range property to return Range objects as arguments to other methods in the Microsoft® Excel object model. When you use the Range property in this way, make sure you fully qualify the Worksheet object to which the Range property applies. Failing to use fully qualified references to the Range property in arguments for Excel methods is one of the most common sources of error in range-related code.

The ActiveCell and Selection Properties

The ActiveCell property returns a Range object representing the currently active cell. When a single cell is selected, the ActiveCell property returns a Range object representing that single cell. When multiple cells are selected, the ActiveCell property represents the single active cell within the current selection. When a cell or group of cells is selected, the Selection property returns a Range object representing all the cells within the current selection.

To understand how the ActiveCell and Selection properties relate to one another, consider the case where a user selects cells A1 through F1 by clicking cell A1 and dragging until the selection extends over cell F1. In this case, the ActiveCell property returns a Range object that represents cell A1. The Selection property returns a Range object representing cells A1 through F1.

When you work with the Microsoft® Excel user interface, you typically select a cell or group of cells and then perform some action on the selected cell or cells, such as entering a value for a single cell or formatting a group of cells. When you use Microsoft® Visual Basic® for Applications (VBA) to work with cells, you are not required to make a selection before performing some action on a cell or group of cells. Instead, you only must return a Range object representing the cell or cells you want to work with. For example, to enter "January" as the value for cell A1 by using the user interface, you would select cell A1 and type January. The following sample performs the same action in VBA:

ActiveSheet.Range("A1").Value = "January"

Using VBA to work with a Range object in this manner does not change the selected cells on the current worksheet. However, you can make your VBA code act upon cells in the same way as a user working through the user interface by using the Range object's Select method to select a cell or range of cells and then using the Range object's Activate method to activate a cell within the current selection. For example, the following code selects cells A1 through A6 and then makes cell A3 the active cell:

```
With ActiveSheet
  .Range("A1:A6").Select
  .Range("A3").Activate
End With
```

When you use the Select method to select multiple cells, the first cell referenced will be the active cell. For example, in the preceding sample, after the Select method is executed, the ActiveCell property returns a reference to cell A1, even though cells A1 through A6 are selected. After the Activate method is executed in the next line of code, the ActiveCell property returns a reference to cell A3 while cells A1 through A6 remain selected. The next example illustrates how to return a Range object by using the ActiveCell property or the Selection property:

```
Dim rngActiveRange As Excel.Range

' Range object returned from the Selection property.
Set rngActiveRange = Selection
Call PrintRangeInfo(rngActiveRange)
' Range object returned from the ActiveCell property.
Set rngActiveRange = ActiveCell
Call PrintRangeInfo(rngActiveRange)
```

Note
The PrintRangeInfo custom procedure called in the preceding example prints information about the cell or cells contained in the Range object passed in the argument to the procedure.

The Macro Recorder and the Selection Object

When you are learning to work with the Excel object model, it is often helpful to turn on the macro recorder and carry out the steps you want to accomplish and then examine the VBA code that results to see which objects, properties, and methods are used. You should be aware, however, that in many cases the macro recorder records your actions from the perspective of a user interacting with the user interface. This means that the Selection object, the Select method, and the Activate method are used over and over.

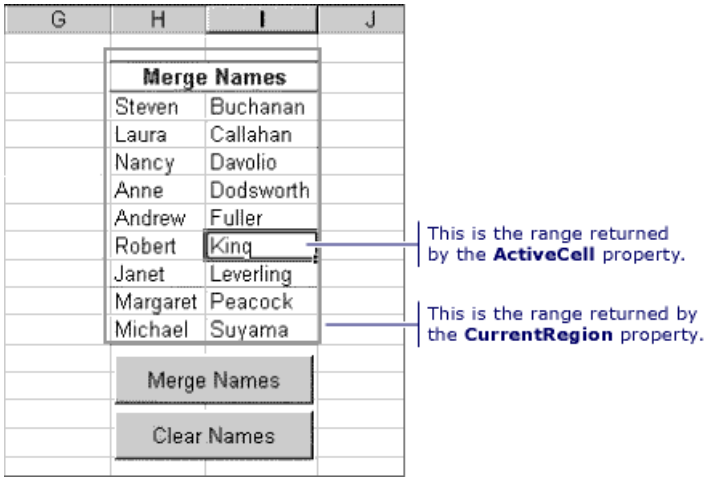
When you get a solid grasp on the most efficient way to work with Excel objects, you will find yourself rewriting or restructuring the VBA code written by the macro recorder to use the Range object instead.

Using the CurrentRegion and UsedRange Properties

There are many circumstances where you will write code to work against a range of cells, but at the time you write the code, you will not have information about the range. For example, you might not know the size or location of a range or the location of a cell in relation to another cell. You can use the CurrentRegion and UsedRange properties to work with a range of cells whose size you have no control over. You can use the Offset property to work with cells in relation to other cells where the cell location is unknown.

As shown in the following figure, the Range object's CurrentRegion property returns a Range object representing a range bounded by (but not including) any combination of blank rows and blank columns or the edges of the worksheet.

The Ranges Returned by the ActiveCell and CurrentRegion Properties



The CurrentRegion property can return many different ranges on a single worksheet. This property is useful for operations where you must know the dimensions of a group of related cells, but all you know for sure is the location of a cell or cells

within the group. For example, when the active cell is inside a table of cells, you could use the following line of code to apply formatting to the entire table:

```
ActiveCell.CurrentRegion.AutoFormat xlRangeAutoFormatAccounting4
```

You could also use the CurrentRegion property to return a collection of cells. For example:

```
Dim rngCurrentCell As Excel.Range
For Each rngCurrentCell In ActiveCell.CurrentRegion.Cells
  ' Work with individual cells here.
Next rngCurrentCell
```

Every Worksheet object has a UsedRange property that returns a Range object representing the area of a worksheet that is being used. The UsedRange property represents the area described by the farthest upper-left and farthest lower-right nonempty cells in a worksheet and includes all cells in between. For example, imagine a worksheet with entries in only two cells: *A1* and *G55*. The worksheet's UsedRange property would return a Range object containing 385 cells between and including *A1* and *G55*.

You might use the UsedRange property together with the SpecialCells method to return a Range object representing all cells in a worksheet of a specified type. For example, the following code returns a Range object that includes all the cells in the active worksheet that contain a formula:

```
Dim rngFormulas As Excel.Range
Set rngFormulas = ActiveSheet.UsedRange.SpecialCells(xlCellTypeFormulas)
```

Using the Cells Property

You use the Cells property to loop through a range of cells in a worksheet or to refer to a range by using numeric row and column values. The Cells property returns a Range object representing all the cells, or a specified cell, in a worksheet. To work with a single cell, you use the Item property of the Range object returned by the Cells property to specify the index of a specific cell. The Item property accepts arguments specifying the row or the row and column index for a cell.

Because the Item property is the default property of the Range object, it is not necessary to explicitly reference it. For example, the following Set statements both return a reference to cell B5 on Sheet1:

```
Dim rng1 As Excel.Range
Dim rng2 As Excel.Range
Set rng1 = Worksheet("Sheet1").Cells.Item(5, 2)
Set rng2 = Worksheet("Sheet1").Cells(5, 2)
```

The row and column index arguments of the Item property return references to individual cells beginning with the first cell in the specified range. For example, the following message box displays "G11" because that is the first cell in the specified Range object:

```
MsgBox Range("G11:M30").Cells(1,1).Address
```

The following procedure illustrates how you would use the Cells property to loop through all the cells in a specified range. The OutOfBounds procedure looks for values that are greater than or less than a specified range of values and changes the font color for each cell with such a value:

```
Function OutOfBounds(rngToCheck As Excel.Range, lngLowValue As Long, lngHighValue As Long, _
            Optional lngHighlightColor As Long = 255) As Boolean
  ' This procedure illustrates how to use the Cells property to iterate through a collection of cells in a range.
  ' For each cell in the rngTocheck range, if the value of the  cell is numeric and it falls outside the range of values
  ' specified by lngLowValue to lngHighValue, the cell font is changed to the value of lngHighlightColor (default is red).
  Dim rngTemp          As Excel.Range
  Dim lngRowCounter    As Long
  Dim lngColCounter    As Long
  ' Validate bounds parameters.
  If lngLowValue > lngHighValue Then
    Err.Raise vbObjectError + 512 + 1,  "OutOfBounds Procedure", _
      "Invalid bounds parameters submitted: "  & "Low value must be lower than high value."
    Exit Function
  End If
  ' Iterate through cells and determine if values  are outside bounds parameters. If so, highlight value.
  For lngRowCounter = 1 To rngToCheck.Rows.Count
    For lngColCounter = 1 To rngToCheck.Columns.Count
      Set rngTemp = rngToCheck.Cells(lngRowCounter, lngColCounter)
      If IsNumeric(rngTemp.Value) Then
        If rngTemp.Value < lngLowValue Or rngTemp.Value > lngHighValue Then
          rngTemp.Font.Color = lngHighlightColor
          OutOfBounds = True
        End If
      End If
```

```
    Next lngColCounter
  Next lngRowCounter
End Function
```

In addition, you can use a For Each...Next statement to loop through the range returned by the Cells property. The following code could be used in the OutOfBounds procedure to loop through cells in a range:

```
' Iterate through cells and determine if values are outside bounds parameters. If so, highlight value.
For Each rngTemp in rngToCheck.Cells
  If IsNumeric(rngTemp.Value) Then
    If rngTemp.Value < lngLowValue Or rngTemp.Value > lngHighValue Then
      rngTemp.Font.Color = lngHighlightColor
      OutOfBounds = True
    End If
  End If
Next rngTemp
```

Using the Offset Property

You can use the Offset property to return a Range object with the same dimensions as a specified Range object but offset from the specified range. For example, you could use the Offset property to create a new Range object adjacent to the active cell to contain calculated values based on the active cell.

The Offset property is useful in circumstances where you do not know the specific address of the cells you must work with, but you do know where the cell is located in relation to other cells you must work with. For example, you might have a command bar button in your custom application that fills the active cell with the average of the values in the two cells immediately to the left of the active cell:

```
ActiveCell.Value = (ActiveCell.Offset(0, -2) + ActiveCell.Offset(0, -1)/2)
```

Working with Microsoft FrontPage Objects

Microsoft® FrontPage® is a powerful and popular application used to create, deploy, and manage Web sites. You also can use FrontPage to create individual Web pages or modify existing Web pages.

FrontPage supports the Microsoft® Visual Basic® Editor and Microsoft® Visual Basic® for Applications (VBA). In addition, to make it possible for you to work with the various parts of a FrontPage-based web or a Web page, FrontPage now exposes a complete object model that you can use either from within a FrontPage VBA project or from another application through Automation. The new VBA language elements replace the FrontPage 98 language elements. To ensure backward compatibility, the FrontPage 98 language elements are included as hidden elements in the latest version of FrontPage object model, but these language elements are not recommended for use in FrontPage.

Note
You can use the Object Browser and Microsoft FrontPage Visual Basic Reference Help to learn more about individual objects, properties, methods, and events.

Although all Office applications support VBA, it is used a bit differently in FrontPage and Microsoft® Outlook® than it is in the other Office applications. FrontPage and Outlook support a single VBA project that is associated with a running instance of the application. The other Office applications make it possible for you to associate a VBA project with each Office document. For example, you can have several workbooks open in Excel at one time, and each workbook can have its own VBA project that contains modules, class modules, and UserForms. In FrontPage, you can have several webs or Web pages open at one time, but there is only one VBA project. The FrontPage VBA project is stored in a file named Microsoft FrontPage.fpm.

In This Section

Understanding the FrontPage Object Model
Learn how to use Microsoft®FrontPage®to work with Web sites.

Understanding the Page Object Model
Write script to work with the HTML elements in a Web page through the DHTML document object model.

Understanding the FrontPage Object Model

Microsoft® FrontPage® is designed to work with Web sites. A Web site can exist on a local intranet or on a server on the World Wide Web. There are several different metaphors you can use when envisioning a Web site. For example, you can think of a Web site as a collection of windows showing different parts of the Web site in an open instance of FrontPage. Or, you might think of a Web site as a collection of files on disk, organized in folders, all of which are organized under one main folder that contains the entire web. Finally, you could think of a Web site as the relationship between pages that exists as a result of the navigation structure between the pages.

When you have one or more Web sites open in FrontPage, the Webs collection contains a Web object representing each open Web site. Each Web object has a WebWindow object that represents the main application window containing the web. The WebWindow object looks like a separate instance of FrontPage, for example, each WebWindow object has its own FrontPage

icon on the Windows taskbar. Each WebWindow object has a PageWindows collection that contains a PageWindow object for each open Web page. In addition, each PageWindow object has a Document property that returns the DHTML document object for a Web page in the FrontPage web.

Figure 1. Microsoft FrontPage Object Model

When you first open FrontPage, you see a blank Web page open in Page view. You can use the Normal, HTML, and Preview tabs along the bottom of the page to edit the page in different ways or to preview it in your Web browser. The FrontPage menu bar and toolbars appear above the page, and the FrontPage Views bar appears along the left side of the page. At this point, FrontPage contains a single blank document only. There is no open Web site, and as a result, the Webs collection object's Count property returns zero. In addition, the WebWindows collection contains a single WebWindow object, and the PageWindows collection contains one PageWindow object that contains the blank page.

Each of these objects and collections has methods and properties you can use to work with the object. For example, the PageWindow object has a Document property you can use to work with the HTML elements contained in the Web page and an IsDirty property you can use to determine if the page has been changed. In addition, the PageWindow object has an ApplyTheme method you can use to apply a FrontPage theme to the page, as well as Save and SaveAs methods you can use to save the page.

Webs based on FrontPage exist on disk as a collection of files organized in folders. The base for the web is the root folder in the directory structure. For example, if you created a web called MyPersonalWeb on your hard disk, the root folder for the web could be C:\MyWebs\MyPersonalWeb. All the directories that make up your Web site would be under the MyPersonalWeb folder. For example, when you use FrontPage to create a Web site based on the Personal Web template, FrontPage creates nine subfolders for the web that contain supporting files, such as images and style sheets.

The file structure of a web as it exists on disk also is available through the object model. Each folder in the web is represented by a WebFolder object. The RootFolder property returns the root WebFolder object in the web. The WebFolders collection contains a WebFolder object for each folder in the web. Each WebFolder object has, among others, a Folders property and a

Files property. The Folders property returns a WebFolders collection for all the subfolders under a folder. The Files property returns a WebFiles collection that contains a WebFile object for each file in a folder.

The navigation structure of a Web site starts with a home page that can branch off to other pages in the Web site. In FrontPage, you can move programmatically through the navigation structure of a Web site by using NavigationNode objects. A NavigationNode object represents a node in the navigation structure of a Web site. The NavigationNodes collection contains all the NavigationNode objects in a Web site. You start navigation at the home page by using the HomeNavigationNode property, which returns the NavigationNode object for the Web site's home page. Each NavigationNode object has a Children property that returns the NavigationNodes collection representing all of the pages you can navigate to from a NavigationNode object. You use the Move method to move among NavigationNode objects in the NavigationNodes collection returned by the Children property. You use the Next and Prev properties to return the next or previous NavigationNode object.

Whether you use windows, files, or navigation nodes to move among objects in FrontPage will depend on what you are trying to accomplish.

Note
You can get more information about the objects, methods, and properties in the FrontPage object model by using Microsoft FrontPage Visual Basic Reference Help.

Understanding the Application Object

The Application object is the top-level object in the Microsoft® FrontPage® object model. It represents FrontPage itself and provides access to all of the objects in the FrontPage object model. If you are automating FrontPage from another Microsoft® Office application, you should set a reference to the Microsoft FrontPage Page Object Reference library by clicking References on the Tools menu in the Microsoft® Visual Basic® Editor in the application from which you are working. Then, you can write code to create an instance of an Application object variable, as shown in the following example:

```
Dim fpApp As FrontPage.Application
Set fpApp = New FrontPage.Application
```

To create a FrontPage Application object without setting a reference to the FrontPage 4.0 Object Reference library, you can use the CreateObject function.

If you are writing Microsoft® Visual Basic® for Applications (VBA) code from within the FrontPage VBA project, you can refer to the Application object directly without creating an object variable.

From the Application object, you can reach any other object in the FrontPage object model. In addition, the properties, methods, and events of the Application object are also global properties that you can use to return currently active objects. A global property is a property that you can use to return an object without having to refer to the Application object or any top-level objects. The global properties that represent active objects in FrontPage are ActiveDocument, ActivePageWindow, ActiveWeb, and ActiveWebWindow. The following examples illustrate how you can work with these properties and the objects they represent:

```
' Apply the classic theme to the active document and specify  vivid colors and active graphics.
ActiveDocument.ApplyTheme "classic", fpThemeVividColors + fpThemeActiveGraphics
' Locate the HelpInformation.htm file in the currently active web.
Dim wflCurrentFile As WebFile
Set wflCurrentFile = ActiveWeb.LocateFile("HelpInformation.htm")
If Not wflCurrentFile Is Nothing Then
  With wflCurrentFile
    ' Code to work with found file here.
  End With
End If
' Check to see if the page in the active page window has changed and,  if so, save it to disk.
With ActivePageWindow
  If .IsDirty Then
    .Refresh SaveChanges:=True
  End If
End With
' Display a message showing the window captions for all open  documents in the active web window.
Dim pgeCurr         As PageWindow
Dim strCaptions     As String
If ActiveWebWindow.PageWindows.Count > 0 Then
  For Each pgeCurr In ActiveWebWindow.PageWindows
    strCaptions = strCaptions & pgeCurr.Caption & vbCrLf
  Next pgeCurr
  If Len(strCaptions) > 0 Then
    MsgBox "The following pages are currently open:" & vbCrLf & strCaptions
  End If
End If
```

The Application object also exposes properties you can use to get information about the current machine, such as the user's name, the version of FrontPage, the language settings, the registry values, and so on. You can use the System property to return the System object, which provides information about the operating system and screen resolution. You also can use the ProfileString property of the System object to read and write FrontPage registry values.

In addition to getting information about the current machine, the Application object provides 10 application-level events you can use to run VBA code when the events occur.

Understanding the Page Object Model

When you write script to work with the HTML elements in a Web page, you are working with the page through the DHTML document object model. FrontPage exposes nearly all of the methods and properties of this object model through the FrontPage Page object model. To see the restrictions of the Page object model, search the Microsoft FrontPage Visual Basic Reference Help index for "object model," and then open the "Exploring the FrontPage Object Model" topic.

In FrontPage, you can use Microsoft®Visual Basic® for Applications (VBA) code to work with the HTML elements in a Web page. You use the Document property or the ActiveDocument property to return a DHTML document object. When you have the document object, you have access to all HTML elements contained in a Web page.

Working with Microsoft Outlook Objects

You can create custom Microsoft® Outlook® objects and manipulate those objects from within Outlook or from another application. You can manipulate Outlook objects by using Microsoft® Visual Basic® for Applications (VBA) code from within Outlook or another Microsoft® Office XP application by using Automation. The Outlook object model exposes Outlook objects, which you can use to gain programmatic access to Outlook functionality. Before you use VBA to access Outlook objects, methods, or properties from another application, you must first set a reference to the Microsoft Outlook object library by clicking References on the Tools menu in the Visual Basic Editor.

In This Section

Understanding the Application and NameSpace Objects
Learn how to use the Application object's CreateItem method to create a new Microsoft® Outlook® item and access existing Outlook items by using the NameSpace object.

Working with Outlook Folders and Items
Access folders for any built-in Microsoft® Outlook® item.

Understanding the Explorer and Inspector Objects
Open the Explorer and Inspector objects programmatically, and display items for the user.

Understanding VBA in Outlook
Associate a Microsoft® Visual Basic®for Applications (VBA) project as code behind an individual Microsoft® Office XP document.

Understanding Events in Outlook
Work with application-level events associated with the application itself or top-level objects within the application or item-level events associated with a particular Microsoft®Outlook®item.

Understanding the Application and NameSpace Objects

When you manipulate Microsoft® Outlook® objects, you always start with the Application object. If you are using Microsoft® Visual Basic® for Applications (VBA) in Outlook, there is a reference to the Outlook object library set by default. If you are using Automation to work with Outlook objects from another application, you must first set a reference to the Outlook object library by using the References dialog box in the application you are working from. If you have set a reference to the Outlook object library, you create a new instance of an Outlook Application object by using the New keyword as follows:

Dim olApp As Outlook.Application
Set olApp = New Outlook.Application

If you have not set a reference to the Outlook object library, you must use the CreateObject function. There can only be one instance of Outlook available at one time. Therefore, when Outlook is not running, the New keyword (or the CreateObject function) creates a new, hidden, instance of Outlook. If an instance of Outlook is already running, then using the New keyword (or the CreateObject function) returns a reference to the running instance.

You use the Application object's CreateItem method to create a new Outlook item. You access existing Outlook items by using the NameSpace object.

Figure 1. Microsoft Outlook Object Model

All the sample procedures discussed in this section use global object variables to represent the Application object and the NameSpace object. Each procedure first checks to see if the Application object variable has been created and, if not, calls the InitializeOutlook procedure to create an instance of the global Application and NameSpace object variables. For example:

```
' Declare global Outlook Application and NameSpace variables.  These are declared as global variables so that they need not
' be re-created for each procedure that uses them.
Public golApp         As Outlook.Application
Public gnspNameSpace   As Outlook.NameSpace

Function InitializeOutlook() As Boolean
   ' This function is used to initialize the global Application and  NameSpace variables.
   On Error GoTo Init_Err
   Set golApp = New Outlook.Application    ' Application object.
   Set gnspNameSpace = golApp.GetNamespace("MAPI") ' Namespace object.
InitializeOutlook = True

Init_End:
   Exit Function

Init_Err:
   InitializeOutlook = False
   Resume Init_End
End Function
```

You use an olItemType constant as the CreateItem method's single argument to specify whether you want to create a new appointment, contact, distribution list, journal entry, mail message, note, posting to a public folder, or task. The CreateItem method returns an object of the type specified in the olItemType constant; you can then use this object to set additional properties of the item. For example, the following procedure creates a new mail message and sets the recipients, attachments, subject, and message text by using the information passed to the procedure as arguments:

```
Function CreateMail(astrRecip As Variant,  strSubject As String,  strMessage As String, _
                Optional astrAttachments As Variant) As Boolean
   ' This procedure illustrates how to create a new mail message  and use the information passed as arguments to set message
   ' properties for the subject, text (Body property), attachments, and recipients.
   Dim objNewMail          As Outlook.MailItem
   Dim varRecip            As Variant
```

```
  Dim varAttach           As Variant
  Dim blnResolveSuccess   As Boolean
  On Error GoTo CreateMail_Err
  ' Use the InitializeOutlook procedure to initialize global Application and NameSpace object variables, if necessary.
  If golApp Is Nothing Then
    If InitializeOutlook = False Then
      MsgBox "Unable to initialize Outlook Application " & "or NameSpace object variables!"
      Exit Function
    End If
  End If
  Set golApp = New Outlook.Application
  Set objNewMail = golApp.CreateItem(olMailItem)
  With objNewMail
    For Each varRecip In astrRecip
      .Recipients.Add varRecip
    Next varRecip
    blnResolveSuccess = .Recipients.ResolveAll
    For Each varAttach In astrAttachments
      .Attachments.Add varAttach
    Next varAttach
    .Subject = strSubject
    .Body = strMessage
    If blnResolveSuccess Then
      .Send
    Else
      MsgBox "Unable to resolve all recipients. Please check the names."
      .Display
    End If
  End With
  CreateMail = True
CreateMail_End:
  Exit Function
CreateMail_Err:
  CreateMail = False
  Resume CreateMail_End
End Function
```

The preceding procedure also illustrates how to use a MailItem object's Recipients and Attachments properties to return the respective collection objects and then add one or more recipients or attachments to a mail message.

You use the Application object's GetNameSpace method to instantiate an object variable representing a recognized data source. Currently, Outlook supports the "MAPI" message store as the only valid NameSpace object. To see an example of how to use the GetNameSpace method to create a NameSpace object variable, see the InitializeOutlook procedure earlier in this section.

If Outlook is not running when you create a NameSpace object variable, the user will be prompted for a profile if the user's mail services startup setting is set to Prompt for a profile to be used. Startup settings are on the Mail Services tab of the Options dialog box (Tools menu). You can use the NameSpace object's Logon method to specify a profile programmatically. Profiles are stored in the Windows registry under the \HKEY_CURRENT_USER\Software\Microsoft\Windows Messaging Subsystem\Profiles subkey.

Working with Outlook Folders and Items

You can think of the NameSpace object as the gateway to all existing Microsoft® Outlook® folders. By default, Outlook creates two top-level folders representing all public folders and all mailbox folders. Mailbox folders contain all Outlook built-in and custom folders. Each folder is a MAPIFolder object. MAPIFolder objects can contain subfolders (which are also MAPIFolder objects), as well as individual Outlook item objects, such as MailItem objects, ContactItem objects, JournalItem objects, and so on.

Note
In Outlook, an item is the object that holds information (similar to files in other applications). Items include mail messages, appointments, contacts, tasks, journal entries, and notes.

When you have created a NameSpace object variable, you can access the top-level folder for any built-in Outlook item by using the NameSpace object's GetDefaultFolder method. For example, the following code sample returns a reference to the ContactItems folder:

```
Dim fldContacts As Outlook.MAPIFolder
Set fldContacts = gnspNameSpace.GetDefaultFolder(olFolderContacts)
```

You can also return a reference to any folder by using the name of the folder. For example, the following procedure returns a reference to the folder in the current user's mailbox whose name is specified in the strFolderName argument:

```
Function GetFolderByName(strFolderName As String) As Outlook.MAPIFolder
  ' This procedure illustrates how to return a MAPIFolder  object representing any folder in the mailbox folders
  ' collection whose name is specified by the strFolderName argument.
  Dim fldMain As Outlook.MAPIFolder

  On Error Resume Next
  ' Use the InitializeOutlook procedure to initialize global Application and NameSpace object variables, if necessary.
  If golApp Is Nothing Then
    If InitializeOutlook = False Then
      MsgBox "Unable to initialize Outlook Application or NameSpace object variables!"
      Exit Function
    End If
  End If

  Set fldMain = gnspNameSpace.Folders(GetMailboxName()).Folders(strFolderName)
  If Err = 0 Then
    Set GetFolderByName = fldMain
  Else
    ' Note: The most likely cause of an error here is that the folder specified in strFolderName could not be found.
    Set GetFolderByName = Nothing
  End If
End Function
```

The NameSpace object has at least two top-level folders representing all public folders and the user's mailbox. The preceding procedure uses the GetMailboxName procedure to return the name of the mailbox folder.

When you return a reference to a folder in the user's mailbox, that folder might contain additional folders, individual Outlook items, or both.

```
Sub GetFolderInfo(fldFolder As Outlook.MAPIFolder)
  ' This procedure prints to the Immediate window information about items contained in a folder.
  Dim objItem          As Object
  Dim dteCreateDate    As Date
  Dim strSubject       As String
  Dim strItemType      As String
  Dim intCounter       As Integer

  On Error Resume Next
  If fldFolder.Folders.Count > 0 Then
    For Each objItem In fldFolder.Folders
      Call GetFolderInfo(objItem)
    Next objItem
  End If

  Debug.Print "Folder '" & fldFolder.Name & "' (Contains " & fldFolder.Items.Count & " items):"

  For Each objItem In fldFolder.Items
    intCounter = intCounter + 1
    With objItem
      dteCreateDate = .CreationTime
      strSubject = .Subject
      strItemType = TypeName(objItem)
    End With

    Debug.Print vbTab & "Item #" & intCounter & " - " & strItemType & " - created on " _
      & Format(dteCreateDate, "mmmm dd, yyyy hh:mm am/pm") _
      & vbCrLf & vbTab & vbTab & "Subject: '" & strSubject & "'" & vbCrLf

  Next objItem
End Sub
```

The GetFolderInfo procedure examines a folder for subfolders and calls itself recursively until there are no subfolders remaining. It then prints information about the items contained in the folder or subfolder to the Immediate window. Note that the objItem object variable is declared by using the Object data type so that the procedure can work with any Outlook item.

To work with a single item or subset of items in a folder, you use the Restrict method, which returns a collection of objects that match the criteria specified in the method's single argument. For example, the following procedure uses the Restrict method to create a collection of Outlook ContactItem objects that match the name supplied in the strLastName argument:

```
Function GetItemFromName(strLastName As String, Optional strFirstName As String = "", _
                         Optional strCompany As String = "") As Boolean
  ' This procedure returns an Outlook ContactItem that matches the criteria specified in the arguments passed to the procedure.
  Dim fldFolder          As Outlook.MAPIFolder
```

```
  Dim objItemsCollection   As Object
  Dim objItem              As Object
  Dim strCriteria          As String
  Dim objMatchingItem      As Object

  On Error GoTo GetItem_Err
  ' Use the InitializeOutlook procedure to initialize global Application and NameSpace object variables, if necessary.
  If golApp Is Nothing Then
    If InitializeOutlook = False Then
      MsgBox "Unable to initialize Outlook Application or NameSpace object variables!"
      Exit Function
    End If
  End If

  Set fldFolder = gnspNameSpace.GetDefaultFolder(olFolderContacts)
  If Len(strLastName) = 0 And Len(strFirstName) = 0 Then
    If Len(strCompany) > 0 Then
      strCriteria = "[Company] = '" & strCompany & "'"
    End If
  Else
    strCriteria = IIf(Len(strFirstName) = 0, "[LastName] = '" & strLastName & "'", _
      "[LastName] = '" & strLastName & "' AND [FirstName] = '" & strFirstName & "'")
  End If

  Set objItemsCollection = fldFolder.Items.Restrict(strCriteria)
  If objItemsCollection.Count > 0 Then
    If objItemsCollection.Count = 1 Then
      For Each objItem In objItemsCollection
        Set objMatchingItem = gnspNameSpace.GetItemFromID(objItem.EntryId)
        objMatchingItem.Display
        GetItemFromName = True
        Exit Function
      Next objItem
    Else
      GetItemFromName = False
      Exit Function
    End If
  End If
  GetItemFromName = True

GetItem_End:
  Exit Function

GetItem_Err:
  GetItemFromName = False
  Resume GetItem_End
End Function
```

When you are using the Restrict method, you use Outlook field names within brackets to specify criteria for a search. You can join multiple criteria by using operators such as And, Or, and Not. For example, the following sample returns all the mail items sent in the last seven days that are unread and marked as highly important:

```
Dim fldMail      As Outlook.MAPIFolder
Dim itmItems     As Outlook.Items

strCriteria = "[SentOn] > '" & (Date - 7)  & "' And [UnRead] = True And [Importance] = High"
Set fldMail = gnspNameSpace.GetDefaultFolder(olFolderInbox)
Set itmItems = fldMail.Items.Restrict(strCriteria)
```

This line illustrates how to return all the Outlook ContactItem items that contain a value in the Business Address field:

```
Set objContacts = fldContacts.Items.Restrict("[BusinessAddress] <> '" & strZLS & "'")
```

The NorthwindContacts.dot sample file is a Microsoft®Word template that retrieves contacts from the Outlook Contacts folder and then displays the contacts in a UserForm. When the user selects a contact from the form, the contact name and address information is inserted in an address block in a letter.

Note
The NorthwindContacts.dot sample file also illustrates how to collect contact information from a database so that the user can insert name and address information into a letter.

Understanding the Explorer and Inspector Objects

The Explorer object represents what you would recognize as the Microsoft® Outlook® user interface. For example, when you open Outlook, you are working in the Outlook Explorer object. A window that contains a specific Outlook item, such as a mail message or a contact, is an Outlook Inspector object.

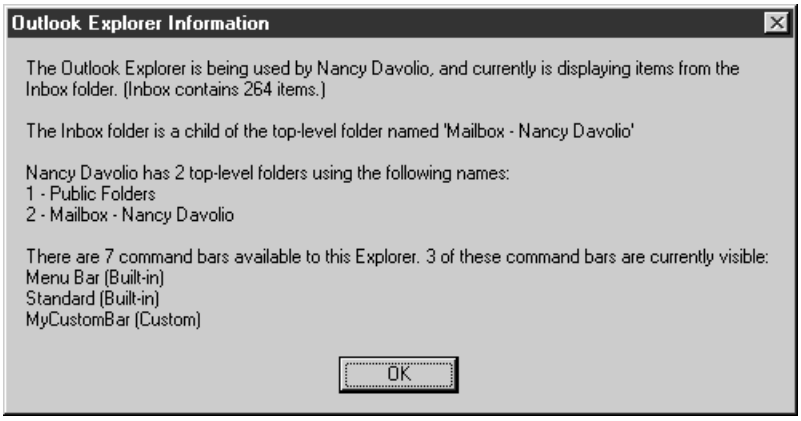
You can open these objects programmatically and display items for the user. You can also use the ActiveExplorer and ActiveInspector methods of the Application object to return a programmatic reference to the Explorer or Inspector object that the user is currently working with.

If you want to use Microsoft® Visual Basic® for Applications (VBA) to add, remove, or manipulate command bars in Outlook, you start with a reference to the Explorer or Inspector object that contains the command bar you want to use and then use the object's CommandBars property to return a reference to the object's CommandBars collection. For example, the following code illustrates how to get a reference to the CommandBars collection for the active Explorer object:

```
Dim cbrExplorerBars As CommandBars
Set cbrExplorerBars = ActiveExplorer.CommandBars
```

Note
You can use the GetExplorerInfo and GetInspectorInfo procedures to see sample code that uses the Explorer and Inspector objects to get information about what is displayed in the active Outlook Explorer and Inspector objects, including information about built-in and custom command bars. The following figure illustrates the kind of information you can get from an Explorer object.

Information Returned by the GetExplorerInfo Procedure



Understanding VBA in Outlook

Developers have wanted to use Microsoft® Visual Basic® for Applications (VBA) in Microsoft® Outlook® since Outlook was first released. Outlook supports both the VBA language and the Visual Basic Editor found in all other Office applications.

Outlook supports a single VBA project that is associated with a particular user and a running instance of the application. The other Microsoft® Office XP applications (except Microsoft® FrontPage®) let you associate a VBA project as code behind an individual Office document. Because Outlook has no document similar to the other Office applications, VBA code is associated only with the application.

Note
The closest thing to a "document" in Outlook is an Outlook item (for example, a mail message, an appointment item, or a task). As in previous versions of Outlook, you use Visual Basic Scripting Edition (VBScript) to write code behind an Outlook item.

The Outlook VBA project is stored in a file named VbaProject.OTM in the following locations:

- **Microsoft Windows.** If user profiles have been set up for multiple users, VbaProject.OTM is stored in the C:\Windows\Profiles\UserName\Application Data\Microsoft\Outlook subfolder. If user profiles have not been set up, VbaProject.OTM is stored in the C:\Windows\Application Data\Microsoft\Outlook subfolder.
- **Microsoft Windows NT Workstation and Microsoft Windows NT Server.** The VbaProject.OTM file is stored in the C:\Winnt\Profiles\UserName\Application Data\Microsoft\Outlook subfolder.
- **Microsoft Windows 2000.** The VbaProject.OTM file is stored in the C:\Documents and Settings\UserName\Application Data\Microsoft\Outlook subfolder.

You use VBA in Outlook to customize the application by working with the objects, methods, properties, and events available through the Outlook object model. For example, you can add code to application-level events to process messages, add custom command bar controls to call custom VBA procedures, or create Component Object Model (COM) add-ins by using the Visual Basic Editor to debug the add-in as it is being developed and tested. You can access the Visual Basic Editor just as you do in any other Office application, by pointing to Macro on the Tools menu, and then clicking Visual Basic Editor.

Understanding Events in Outlook

There are two classes of events in Microsoft® Outlook®, and you work with each class differently. The first class of events supported in Outlook represents application-level events. Because these events are associated with the application itself, or with top-level objects within the application, such as folders or the Outlook Bar, you can use Microsoft®Visual Basic® for Applications (VBA) code to handle these events.

The second class represents item-level events that are associated with a particular Outlook item. For example, an Outlook MailItem object has events such as Open, Close, Forward, and Send. As in previous versions of Outlook, you use Visual Basic Scripting Edition (VBScript) code within the item itself to handle these item-level events.

Application-Level Events

When you create a new Microsoft® Visual Basic® for Applications (VBA) project in Microsoft® Word or Microsoft®Excel, the project contains, by default, a class module bound to the application's current document. For example, Word creates a module for the ThisDocument object and Excel creates a module for the ThisWorkbook object. In Microsoft® Outlook®, because you use VBA to work with the application, the VBA project contains a class module called ThisOutlookSession, which is pre-bound to the Outlook Application object. As a result, all application-level events are available to you in the Visual Basic Editor Procedures drop-down list when you click the Application object in the Object drop-down list.

There are six events associated with the Application object that you can use to run custom VBA procedures. For example, you could use the Startup event to call custom procedures to customize the Outlook workspace or to create or display custom command bars or command bar controls. You could use the NewMail event procedure to call custom procedures that implement your own rules for handling incoming mail. These events are somewhat self-explanatory, and you can get complete documentation for each event by searching the Microsoft Outlook Visual Basic Reference Help index for the name of the event.

Item-Level Events

Working with the event procedures exposed by Microsoft®Outlook® objects (other than the Application object) is identical to creating event procedures in the other Office applications. First, you must declare an object variable by using the WithEvents keyword in the ThisOutlookSession module (or in another class module) for each object you want to work with. Second, you must add the Microsoft® Visual Basic® for Applications (VBA) code to the event procedure that you want to run when the event occurs. Finally, you must initialize the object variables that you have created.

For example, the following VBA code illustrates how to create an object variable that represents the Outlook Bar in the ThisOutlookSession module:

```
Dim WithEvents obpOutlookBar As Outlook.OutlookBarPane
```

When you declare an object variable as shown in the previous example, the variable name appears in the Object drop-down list in the class module's Code window. When you select this variable from the Object list, you can select the object's available event procedures by using the Procedure drop-down list. For example, the OutlookBarPane object shown earlier exposes the BeforeGroupSwitch and BeforeNavigate events.

```
Private Sub opbOutlookBar_BeforeNavigate(ByVal Shortcut As OutlookBarShortcut,  Cancel As Boolean)
  If Shortcut.Name <> "Inbox" Then
    Msgbox "Sorry, you only have permission to access the Inbox."
    Cancel = True
  End If
End Sub
```

Now you need to initialize the object variable. You can do this in two places: in the Application object's Startup event procedure, so that the variable is always available, or in a custom procedure you create for the purpose of initializing object variables. The following code shows how to initialize the object variable by using the Startup event procedure:

```
Private Sub Application_Startup()
  Set opbOutlookBar = Application.ActiveExplorer.Panes("OutlookBar")
End Sub
```

To determine how to instantiate an object variable, search the Microsoft Outlook Visual Basic Reference Help index for the name of the object you want to work with. For example, the Help topic for the OutlookBarPane object shows that the object is a member of the Panes collection and also that you use the string "OutlookBar" to identify the object within the collection.

Order of Events

Except for certain form events, your program cannot assume that events will occur in a particular order, even if they appear to be called in a consistent sequence. The order in which Outlook calls event handlers might change depending on other events that might occur, or the order might change in future versions of Outlook.

Note
You can get more information about the objects, methods, and properties in the Outlook object model by using Microsoft Outlook Visual Basic Reference Help.

Working with Microsoft PowerPoint Objects

As with other Microsoft® Office applications, you begin automating Microsoft® PowerPoint® by using the Application object. From the Application object, you can open an existing Presentation object or create a new presentation. Each Presentation object contains one or more Slide objects and each Slide object can contain Shape objects that represent text, graphics, tables, and other items found on a slide.

In This Section

Understanding the PowerPoint Application Object
Use the Application object to get started writing Microsoft® Visual Basic® for Applications (VBA) code to work with PowerPoint.

Working with the Presentation Object
Use a PowerPoint template, a presentation saved with a .pot extension that contains master slides and might contain regular slides, to apply a consistent look to an entire presentation.

Working with PowerPoint Slides
Use the Slides collection returned by the Slides property of the Presentation object to add new slides to or access an existing slide in a presentation.

Working with Shapes on Slides
Understand how to refer to a Shape object on a slide.

Understanding the PowerPoint Application Object

When you write Microsoft® Visual Basic® for Applications (VBA) code to work with Microsoft® PowerPoint®, you begin with the Application object. If you are writing VBA code within PowerPoint, the Application object is created for you. If you are automating PowerPoint from some other application, you first create a PowerPoint Application object variable and then create an instance of PowerPoint. Unlike the other Microsoft® Office applications (except Microsoft® Outlook®), there can be only one instance of PowerPoint running at a time. If an instance of PowerPoint is running and you use the New keyword or the CreateObject or GetObject function to instantiate a PowerPoint object variable, that object variable will point to the currently running instance of PowerPoint. This single instance of the Application object can contain any number of open Presentation objects.

Figure 1. Microsoft PowerPoint Object Model (page 1)

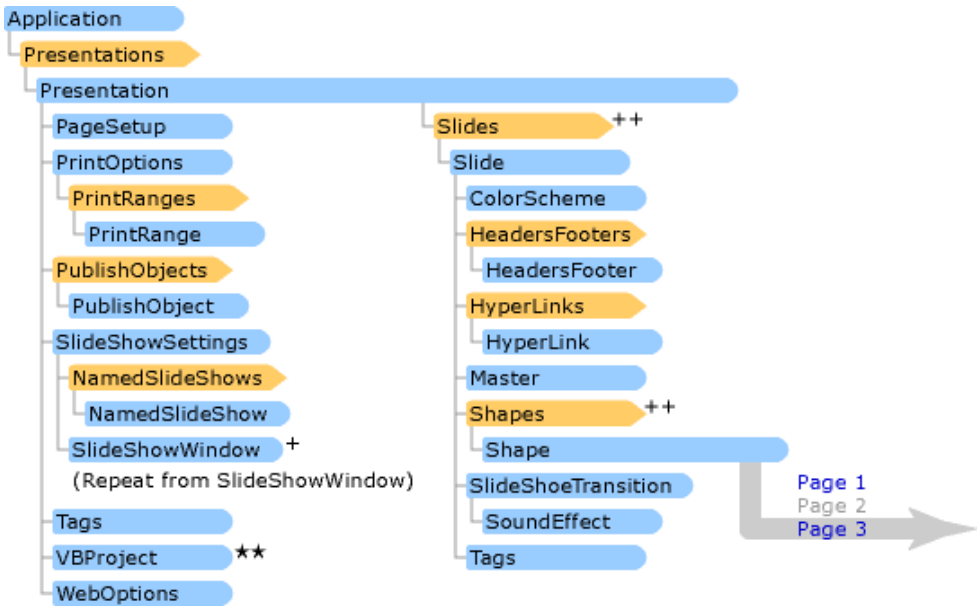


Figure 2. Microsoft PowerPoint Object Model (page 2)

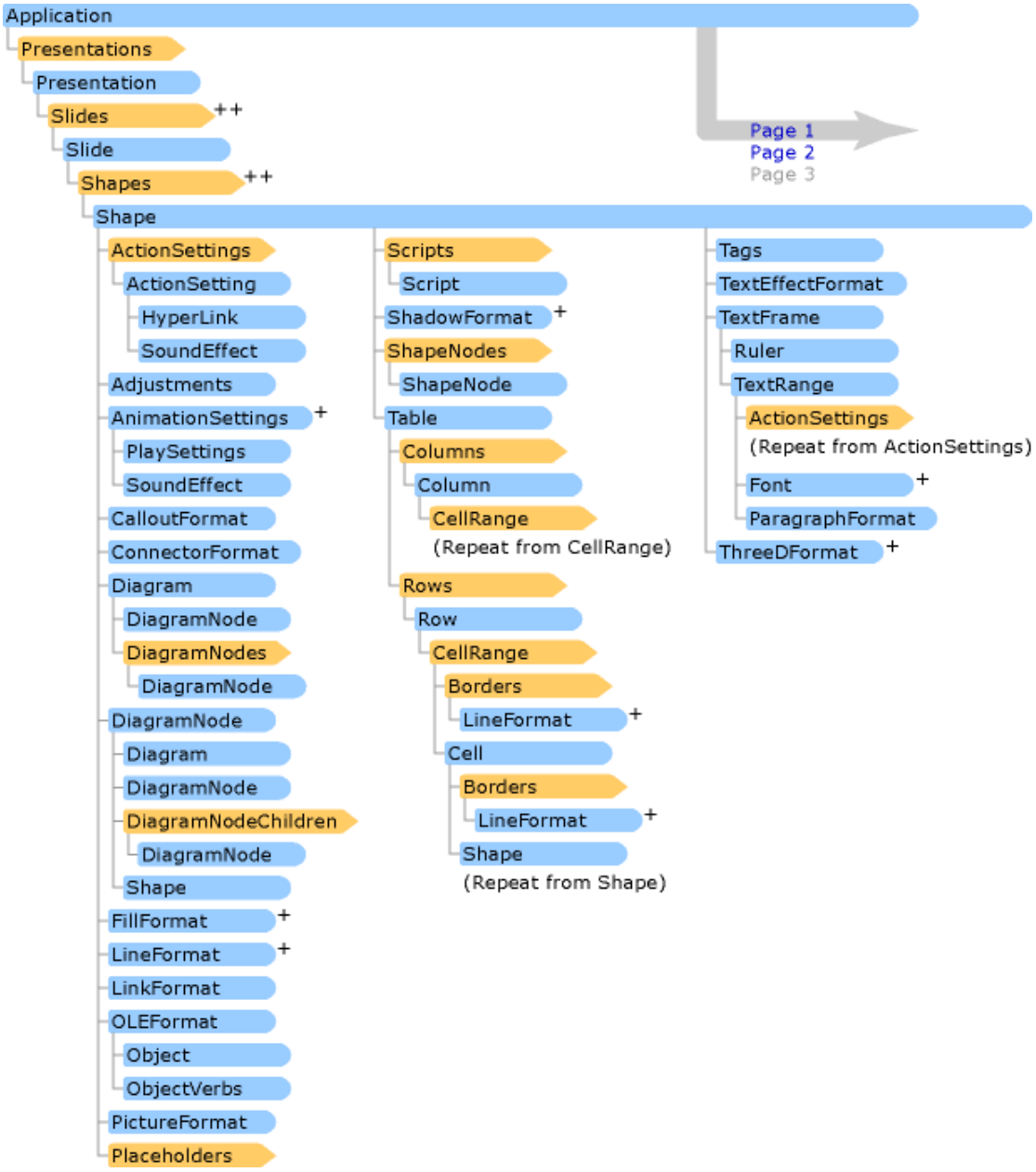
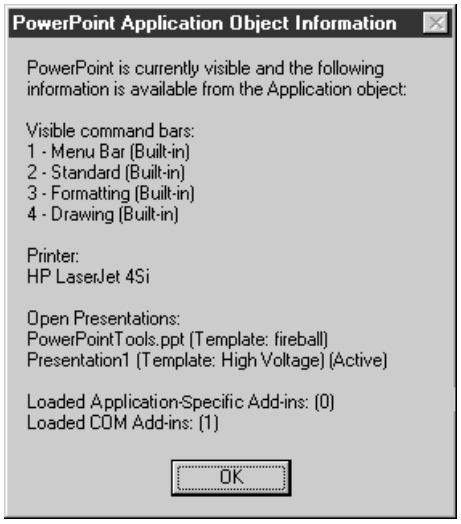


Figure 3. Microsoft PowerPoint Object Model (page 3)

Microsoft PowerPoint's Application object has properties you can use to access shared Office components such as command bars and the Office Assistant. In addition, the Application object has properties that return the currently active presentation or window, or information about the printer.

Information Returned by the Application object properties



Working with the Presentation Object

When you are working with Microsoft® PowerPoint® objects through Microsoft® Visual Basic®for Applications (VBA), you typically work with a Presentation object and the slides it contains. You use a PowerPoint template-a presentation saved with a .pot extension that contains master slides and might contain regular slides-to apply a consistent look to an entire presentation.

Working with Open Presentations

You create a reference to an open presentation in two ways: by using the Application object's ActivePresentation property or by accessing a Presentation object as a member of the Presentations collection. There are three ways you can access a Presentation object through the Presentations collection:

- By using the presentation's file name.
- By using the Caption property setting of the Window object that contains the presentation.
- By using the presentation's index value. Microsoft® PowerPoint® presentations are indexed in the order in which they are opened.

The following examples illustrate the different ways to set a reference to an open presentation:

```
Dim prsPres As PowerPoint.Presentation
' Use the ActivePresentation property.
Set prsPres = ActivePresentation
' Use the presentation's file name.
Set prsPres = Presentations("PowerPointTools.ppt")
' Use the Caption property setting of the Window  object that contains the presentation.
Set prsPres = Presentations("PowerPointTools")
' Use the presentation's index value in the collection.
Set prsPres = Presentations(1)
```

Working with Existing Presentations

You use the Presentations collection's Open method to open a presentation saved to disk and create a reference to that presentation at the same time. The following example opens the PowerPointTools.ppt presentation:

```
Dim ppApp      As PowerPoint.Application
Dim prsPres    As PowerPoint.Presentation
Set ppApp = New PowerPoint.Application
Set prsPres = ppApp.Presentations.Open("c:\opg\Samples\CH05\PowerPointTools.ppt")
With prsPres
  ' Code to manipulate presentation and its ' contents goes here.
End With
```

Creating a New Presentation

There are two ways you can create a Microsoft® PowerPoint® presentation:

- By using the Open method of the Application object. You can use any file format recognized by PowerPoint in the Open method's FileName argument. For example, if the FileName argument specifies a Microsoft® Word document in outline view, the outline will be converted to a new presentation with a slide representing each paragraph that has the Heading 1 style in the document.
- By using the Presentations collection's Add method. For example:

```
Dim ppApp     As PowerPoint.Application
Dim prsPres   As PowerPoint.Presentation
Set ppApp = New PowerPoint.Application
With ppApp
  Set prsPres = .Presentations.Add(msoFalse)
  With prsPres
    ' Code here to add and format slides in the new presentation.
  End With
End With
```

Note

The WithWindow argument of the Add and Open methods accepts a Boolean value that specifies whether the Window object that contains the presentation will be visible. (The default is True.) Although the Auto List Members drop-down list for the Add method's WithWindow argument contains five enumerated constants, you should use only the msoTrue or msoFalse constants.

When you use Microsoft® Visual Basic®for Applications (VBA) to create a new presentation, it exists in memory, but will not be saved to disk until you use the Presentation object's SaveAs method. (Use the Save method to save changes to a presentation that has already been saved to disk.) The following procedure creates a new Presentation object and immediately saves the presentation by using the name supplied in the strPresName argument. It then returns the new Presentation object to the calling procedure.

```
Function PPTCreatePresentation(ppApp As PowerPoint.Application,  strPresName As String) As PowerPoint.Presentation
  ' This procedure illustrates how to use the SaveAs method to  save a new presentation as soon as it is created.
  ' Note that in this example, the new Presentation object is not visible.
  On Error GoTo PPTCreate_Err
  Set PPTCreatePresentation = ppApp.Presentations.Add(msoFalse)
  If InStr(strPresName, "\") = 0 Then
    strPresName = "c:\" & strPresName
  End If
  PPTCreatePresentation.SaveAs strPresName
PPTCreate_End:
  Exit Function
PPTCreate_Err:
  Select Case Err
    Case Err <> 0
      Set PPTCreatePresentation = Nothing
  End Select
  Resume PPTCreate_End:
End Function
```

Formatting a Presentation

You use a Microsoft®PowerPoint® template to apply a consistent look to an entire presentation. A PowerPoint template is a presentation saved with a .pot extension that contains master slides and might contain regular slides. To see the difference, compare the master-slide-only templates found in C:\Program Files\Microsoft Office\Templates\Presentation Designs with the templates found in C:\Program Files\Microsoft Office\Templates\Presentations. Templates that contain slides typically include boilerplate text that you can replace with your own text to make a custom presentation.

Master slides specify the basic layout and formatting for the title slide in a presentation as well as regular slides, handouts, and notes. When you use the ApplyTemplate method, you specify the template that contains the master slides, which include the layout and formatting you want to apply to your presentation. For example, the following sample code applies the Fireball.pot template to the currently active presentation:

```
With ActivePresentation
  .ApplyTemplate FileName:="c:\program files\microsoft office\templates\presentation designs\fireball.pot"
End With
```

You can also use Microsoft® Visual Basic®for Applications (VBA) to create or manipulate master slides directly. Each Presentation object has a property that returns the available master slide that contains the formatting you want to use. You use the Presentation object's TitleMaster, SlideMaster, HandoutMaster, and NotesMaster properties to return a Slide object that represents the master slide you want to work with. Any changes you make to the layout or formatting of a master slide are applied to all slides of the specified type in the current presentation. For example, the following sample adds the CompanyLogo.bmp image to the background of the title master slide:

```
ActivePresentation.TitleMaster _  .Shapes.AddPicture(Filename:="c:\CompanyLogo.bmp", _
  Left:=100, Top:=200, Width:=400, Height:=300)
```

The master properties are useful when you want to apply changes to all slides based on a master, rather than applying changes one slide at a time. If you have an image or other formatting you want to appear on all slides in a presentation, make the change to the appropriate master slide.

Running a Slide Show from a Presentation

You use properties of the SlideShowSettings object to specify how you want a slide show to appear and which slides to include in the show. You use the SlideShowSettings object's Run method to start the slide show. You access the SlideShowSettings object by using the Presentation object's SlideShowSettings property. These objects and properties are most useful if you want to create and run a Microsoft® PowerPoint®presentation from another Microsoft® Office application. For example, the following code is from the PresentationView sample procedure, and it is used in Microsoft® Word to take a Word outline and display it as a PowerPoint presentation. The slide show runs automatically, and when it is finished, it returns the focus to the Word document from which the macro was run.

```
Set prsPres = ppApp.Presentations.Open(strOutlineFileName)
' Format the presentation and set the slide show timings.
With prsPres
  .ApplyTemplate strTemplate
  With .Slides.Range.SlideShowTransition
    .AdvanceTime = intShowSlide
    .AdvanceOnTime = msoTrue
  End With
  ' Run the slide show, showing each slide once,  and then end the show and close the presentation.
  With .SlideShowSettings
```

```
    .AdvanceMode = ppSlideShowUseSlideTimings
    .ShowType = ppShowTypeSpeaker
    .StartingSlide = 1
    .EndingSlide = prsPres.Slides.Count
    Set objCurrentShow = .Run.View
    Do Until objCurrentShow.State = ppSlideShowDone
      DoEvents
    Loop
  End With
End With
```

Working with PowerPoint Slides

Every Microsoft® PowerPoint®presentation (with the exception of some templates) is a collection of slides. Each slide can contain text or graphics and might include animation effects. The Presentation object has a Slides property that returns the Slides collection. The Slides collection is used to add new slides to or access an existing slide in a presentation. Each slide is represented in the collection by a Slide object.

Working with the Slides Collection

You primarily use the Slides collection to add new slides to a presentation or to access a specific slide within a presentation. You use the Slides collection's Add method to add a new slide to a collection. You use arguments of the Add method to specify the location of the slide in the Slides collection and to specify the slide's layout. The following example shows how you would add a new blank slide to the end of the current Slides collection:

```
Dim sldNewSlide          As PowerPoint.Slide
Dim lngLastSlideAdded    As Long

With ActivePresentation
  Set sldNewSlide = .Slides.Add(.Slides.Count + 1, ppLayoutBlank)
  With sldNewSlide
    ' Add code to set properties of the slide here.
    lngLastSlideAdded = .SlideID
  End With
End With
```

You can add existing slides, or data that can be converted to slides, to a presentation by using the Slides collection's InsertFromFile method. For example, you could create a new presentation that used the opening and closing slides from a company presentation template and then used a Word outline to create the slides that make up the body of the presentation:

```
Dim ppApp     As New PowerPoint.Application
Dim prsPres   As PowerPoint.Presentation

With ppApp
  Set prsPres = .Presentations.Add
  With prsPres
    .ApplyTemplate "c:\corp\corpPresentations.pot"
    .Slides.InsertFromFile "c:\PPTOutline.doc", 1
  End With
End With
```

To locate a slide within the collection, you use the Slides collection's FindBySlideID method. Each slide in a PowerPoint presentation has a SlideID property that is a Long Integer value that uniquely identifies the slide regardless of its location in the Slides collection. When you add to or delete slides from a collection, a slide's index value might change, but its SlideID property will always be the same. The first code sample in this section illustrates how to save the SlideID property to a variable so that it might be used again to locate the slide. The following sample shows how to locate a slide by using the Long Integer value representing the SlideID property:

```
Function FindSlide(lngID As Long) As PowerPoint.Slide
  ' This procedure returns the slide whose SlideID property value  matches lngID. If no match is found, the return value of the
  ' procedure is = Nothing.
  On Error Resume Next
  Set FindSlide = ActivePresentation.Slides.FindBySlideID(lngID)
End Function
```

Working with Slide Objects

By default, Microsoft® PowerPoint® names slides by using the convention Sliden, where n is a number representing the location of the slide at the time it was added to the Slides collection. You can specify your own name for a slide by setting the Slide object's Name property.

There are four ways to access a Slide object in the Slides collection:

- By using an index value representing the location of the slide in the Slides collection.
- By using the slide's name.
- By using the slide's SlideID property with the Slides collection's FindBySlideID method.
- By using the SlideIndex property of the SlideRange object from the PowerPoint Selection object to return the currently selected slide; however, using the SlideIndex property in this manner might return an error, if more than one slide is selected.

The following code sample illustrates three ways to return the third Slide object in the current presentation and a way to return the currently selected slide:

```
Dim sldCurrentSlide As PowerPoint.Slide
' Using the slide's index value.
Set sldCurrentSlide = ActivePresentation.Slides(3)
' Using the slide's name.
Set sldCurrentSlide = ActivePresentation.Slides("Slide3")
' Using the FindBySlideID method, where lngSlide3 contains the SlideID  property for the third slide.
Set sldCurrentSlide = ActivePresentation.Slides.FindBySlideID(lngSlide3)
' Using the SlideIndex property to return the currently selected slide.
' This sample shows how to determine if a single slide is currently selected.
If ActiveWindow.Selection.SlideRange.Count = 1 Then
   Set sldCurrentSlide = ActivePresentation _
     .Slides(ActiveWindow.Selection.SlideRange.SlideIndex)
End If
```

If you want to work with a group of Slide objects, perhaps to apply consistent formatting to the slides, you can use the Slides collection's Range method. The Range method returns a SlideRange object representing one or more Slide objects in a presentation.

If you use the Range method without an argument, the method returns a SlideRange object that contains all the Slide objects in a presentation.

You use the Range method's Index argument to specify one or more Slide objects to include in the SlideRange object returned by the method. If the argument is a single integer, the method returns a SlideRange object for the Slide object whose index value matches the integer. For example, the following sample returns a SlideRange object representing the third slide in the current presentation:

```
Dim sldCurrSlide As PowerPoint.SlideRange
Set sldCurrSlide = ActivePresentation.Slides.Range(3)
```

In addition to using the Index argument, you can also use the Microsoft® Visual Basic® for Applications (VBA) Array function as an argument to the Range method in order to return a SlideRange object containing multiple Slide objects. The Array function uses a comma-delimited list of values to be included in the array. When used as an argument to the Range method, the comma-delimited list should contain the index values or names of the slides you want to include in the SlideRange object returned by the method. The following sample shows how to use the Array function to return a SlideRange object containing the first four slides with even-numbered index values:

```
Dim sldCurrSlides As PowerPoint.SlideRange
Set sldCurrSlides = ActivePresentation.Slides.Range(Array(2,4,6,8))
```

The next sample illustrates how to use the Array function to return a SlideRange object that is a collection of specific named slides in a presentation:

```
Dim sldCurrSlides As PowerPoint.SlideRange
Set sldCurrSlides = ActivePresentation.Slides.Range(Array("CostOfGoods", "SalesTotals", "Benefits", "Forecast"))
With sldCurrSlides
   ' Set properties common to all slides in this collection.
End With
```

Note
You can also use the Array function as an argument to the Range method for a Shapes collection in order to return a collection of specified Shape objects as a ShapeRange object.

Working with Shapes on Slides

Just as a Microsoft® PowerPoint® presentation consists of a collection of slides, a PowerPoint slide typically consists of one or more Shape objects. Whether a slide contains a picture, a title, text, an OLE object, an AutoShape, a diagram, or other content, everything on the slide is a Shape object.

You can refer to a Shape object on a slide in two ways:

- By using the value of the shape's index in the collection of shapes on the slide. A shape will have an index value equal to its position in the Shapes collection at the time it was added to the collection.
- By using the name of the shape. You can specify the name of a Shape object by setting its Name property. By default, PowerPoint sets the name of a shape at the time it is added to a slide. The naming convention is shapetype n, where shapetype is the type of shape added and n is a number representing 1 plus the number of shapes on the slide when the current shape was added. For the first shape added to a slide, n = 2. To find out more about the types of shapes available in PowerPoint, search the Microsoft PowerPoint Visual Basic Reference Help index for "Shapes collection object."

To work with multiple shapes on a slide, you use the Range method of the Shapes collection. The Range method returns a ShapeRange object containing the shapes specified in the method's argument. If no Index argument is supplied, the Range method returns a ShapeRange object containing all the shapes on a slide. To specify multiple shapes, you can use the Microsoft® Visual Basic®for Applications (VBA) Array function.

Adding Shapes to Slides

Typically, you use the Add method of a collection object to add an item to the collection. For example, to add a slide to a Microsoft® PowerPoint® presentation, you use the Presentation object's Slides collection's Add method. However, adding shapes to a slide is a little different. The PowerPoint object model provides a different method for each shape you can add to a slide. For example, the following sample inserts a new slide at the end of the current presentation and uses two methods of the Shapes collection to add shapes to the slide. The AddTextEffect method is used to add a WordArt shape and the AddTextbox method is used to add a text box shape:

```
Sub AddTestSlideAndShapes()
  ' Illustrate how to add shapes to a slide and then  center the shapes in relation to the slide and  each other.
  Dim sldNewSlide        As PowerPoint.Slide
  Dim shpCurrShape       As PowerPoint.Shape
  Dim lngSlideHeight     As Long
  Dim lngSlideWidth      As Long

  With ActivePresentation
    ' Determine height and width of slide.
    With .PageSetup
      lngSlideHeight = .SlideHeight
      lngSlideWidth = .SlideWidth
    End With

    ' Add new slide to end of presentation.
    Set sldNewSlide = .Slides.Add(.Slides.Count + 1, ppLayoutBlank)
    With sldNewSlide
      ' Specify a background color for the slide.
      .ColorScheme = ActivePresentation.ColorSchemes(3)
      ' Add a WordArt shape by using the AddTextEffect method.
      Set shpCurrShape = .Shapes.AddTextEffect(msoTextEffect16, _
        "Familiar Quotations", "Tahoma", 42, msoFalse, msoFalse, 100, 100)
      ' Locate the WordArt shape at the middle of the slide, near the top.
      With shpCurrShape
        .Left = (lngSlideWidth - .Width) / 2
        .Top = (lngSlideHeight - .Height) / 8
      End With

      ' Add a Textbox shape to the slide and add text to the shape.
      Set shpCurrShape = .Shapes _
        .AddTextbox(msoTextOrientationHorizontal, 100, 100, 500, 500)
      With shpCurrShape
        With .TextFrame.TextRange
          .Text = "'If not now, when? If not us, who?'" & vbCrLf & "'There is no time like the present.'" _
            & vbCrLf & "'Ask not what your country can do for you, " & "ask what you can do for your country.'"
          With .ParagraphFormat
            .Alignment = ppAlignLeft
            .Bullet = msoTrue
          End With
          With .Font
            .Bold = msoTrue
            .Name = "Tahoma"
            .Size = 24
          End With
        End With
        ' Shrink the Textbox to match the text it now contains.
        .Width = .TextFrame.TextRange.BoundWidth
        .Height = .TextFrame.TextRange.BoundHeight
```

```
        .Left = (lngSlideWidth - .Width) / 2
        .Top = (lngSlideHeight - .Height) / 2
      End With
    End With
  End With
End Sub
```

Positioning Shapes on Slides

When you add a shape to a slide, the method you use typically requires you to specify values to establish the dimensions of the shape. In some cases, as with the AddTextEffect method (illustrated in the AddTestSlideAndShapes procedure shown earlier in "Adding Shapes to Slides"), you specify values for the Left and Top properties of the shape (the height and width of the shape is determined by the text it contains). In other cases (as with the AddTextbox method, also illustrated in the AddTestSlideAndShapes procedure), you must specify values for the Shape object's Left, Top, Width, and Height properties.

The height and width of shapes are specified in pixels. The default slide size is 720 pixels wide and 540 pixels high. The center of a slide is 360 pixels from the left edge of the slide and 270 pixels from the top of the slide. You can center any shape horizontally by using the formula (SlideWidth - ShapeWidth) / 2. You can center any shape vertically by using the formula (SlideHeight - ShapeHeight) / 2. You can programmatically specify or determine the height and width setting for the slides in a presentation by using the Presentation object's PageSetup property to return a PageSetup object, and then use the PageSetup object's SlideHeight and SlideWidth properties. This technique is also illustrated in the AddTestSlideAndShapes procedure shown earlier.

To position one or more shapes on a slide either in relation to the slide or to other shapes on the slide, you can use the Align or Distribute methods of a ShapeRange object.

Working with Text in a Shape

Much of what you do with shapes on slides involves adding or modifying text. In addition to the Textbox shape, many other Shape objects can contain text. For example, you can add text to many of the AutoShape Shape objects.

All shapes that support text have a TextFrame property you can use to return a TextFrame object. You can determine if a shape supports the use of a text frame by using the Shape object's HasTextFrame property. Each TextFrame object has a HasText property you can use to determine if the text frame contains text.

The TextFrame object has a TextRange property you use to return a TextRange object. You use the TextFrame object's Text property to specify or determine the text within a frame. You use the properties and methods of the TextRange object to work with the text associated with a Microsoft® PowerPoint® shape.

Note
Placeholder shapes contain default text that is visible from the PowerPoint user interface, but is not available programmatically. When you set the Text property of a Placeholder shape's TextRange object, the default text is replaced with the text you specify.

There is one Shape object that contains text but does not use the TextFrame or TextRange objects. The TextEffect property returns a TextEffectFormat object that contains the properties and methods used to work with WordArt shapes. You add WordArt shapes to a slide by using the Shapes collection's AddTextEffect method. The text of the WordArt shape and the location of the shape are specified in the arguments to the AddTextEffect method. You use the TextEffectFormat object's Text property to read or change the text in a WordArt shape. For example, the following code changes the text of an existing WordArt shape on the first slide of the current presentation:

```
With ActivePresentation
  strExistingText =    .Slides(1).TextEffect.Text
  If Len(strNewText) <= Len(strExistingText) Then
    .Slides(1).TextEffect.Text = strNewText
  End If
End With
```

Note that this code checks to make sure the new text is not longer than the existing text. This step is required because a WordArt shape does not automatically resize itself to accommodate new text. Alternatively, you could capture the properties of the existing WordArt shape, then delete it and replace it with a new WordArt shape that uses the same properties as the old shape.

Working with Microsoft Project Objects

With the Microsoft® Project 2000 object model, you can build powerful custom applications easily. Microsoft® Visual Basic® Applications Edition programming system extends the Visual Basic programming style to access Project project-planning software!Vsupplied objects. In addition, the Visual Basic programming style is extended to access a Microsoft® Excel spreadsheet, Microsoft® Word word processing, the Microsoft® Access database management system, and the Microsoft® PowerPoint® presentation graphics program. This common macro language across applications makes it possible for access to schedule data stored in a project, as well as all the interface commands in the macro language found in Project.

In This Section

Understanding the Project Object Model
Each Project object contains summary information, tasks, and resources. The Project object represents an individual project or a collection of projects.

Understanding the Project Application Object
The Application object is the top of the hierarchy. All project objects are accessed through the Application object.

Understanding the Project Object Model

The Application object contains all Microsoft® Project objects. Each Project object contains summary information, tasks, and resources. The Project object represents an individual project or a collection of projects. In addition, Project objects are parent objects of Windows, Tasks, Resources, and Calendars collections.

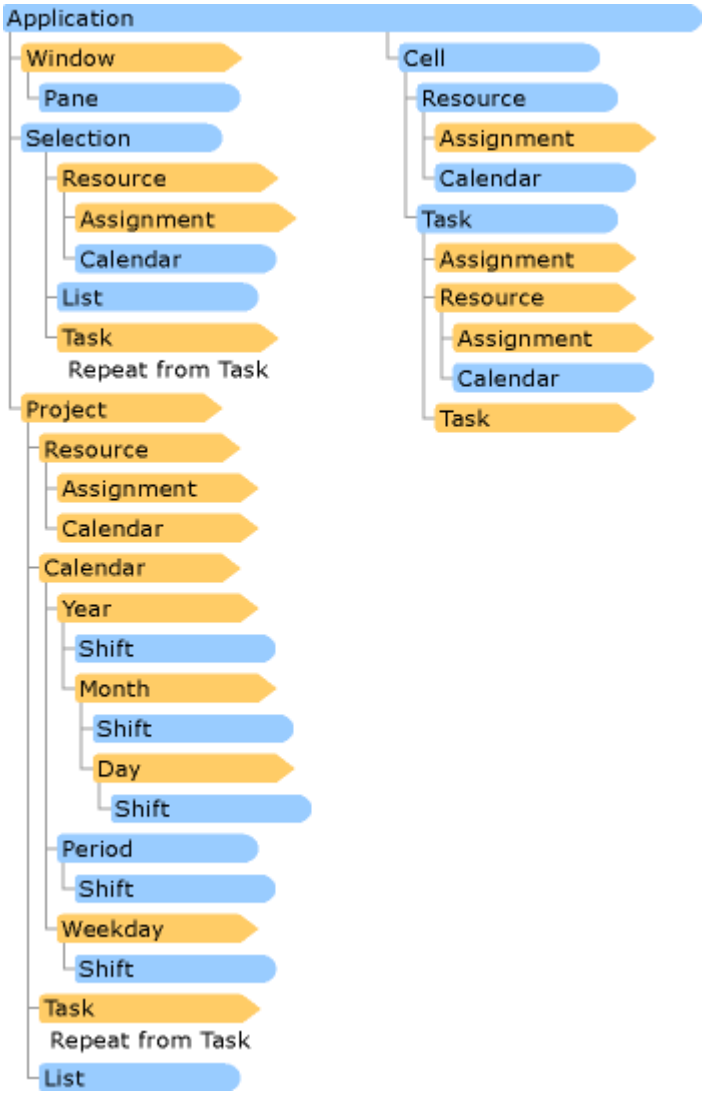


Figure 1. Microsoft Project Object Model

Project objects can be referenced using the Application property ActiveProject, by referring to the project's index value or by naming the project:

```
Sub ProjectRefs()
MsgBox ActiveProject
MsgBox Projects(1).Name
MsgBox Projects("Project1").ProjectStart
End Sub
```

Project Object Properties

Fields in the project summary task record can be browsed or edited using Project object properties. The following examples return information from these fields. After running the macro, review the information on the Schedule tab in the Tools Options dialog box.

```
Sub ProjectProperties()
MsgBox ActiveProject.ActualCost
MsgBox ActiveProject.Author
```

```
MsgBox ActiveProject.AutoLinkTasks
ActiveProject.AutoLinkTasks = False
End Sub
```

Other properties are useful for returning active views, tables, and filters. Make sure to run the following procedure from a task or resource view:

```
Sub ProjectCurrentProperties()
MsgBox ActiveProject.CurrentView
MsgBox ActiveProject.CurrentTable
MsgBox ActiveProject.CurrentFilter
End Sub
```

Project Object Methods

The Tasks method returns a task or tasks in a Project object. To iterate through tasks try the following two routines:

```
Sub TaskDisplay()
Dim i As Integer
Dim t As Variant
For i = 1 To ActiveProject.Tasks.Count
MsgBox ActiveProject.Tasks(i).Name
Next i
For Each t In ActiveProject.Tasks
MsgBox t.Name
Next t
End Sub
```

The Resources method returns a resource or resources in a Project object:

```
Sub ResourceNameDisplay()
Dim r as Variant
For Each r in ActiveProject.Resources
MsgBox r.Name
Next r
End Sub
```

The TaskViewList, TaskTableList, TaskFilterList, ResourceViewList, ResourceTableList, and ResourceFilterList methods return available views, tables, and filters in a project. The following example displays a list of available task views:

```
Sub ListViews()
Dim strViewList As String
Dim i As Integer
NL = Chr$(13) & Chr$(10)
strViewList = "Task Views:" & NL
For i = 1 To ActiveProject.TaskViewList.Count
strViewList = strViewList & ActiveProject.TaskViewList(i) & NL
Next i
MsgBox strViewList
End Sub
```

Understanding the Project Application Object

The Application object is the top of the hierarchy. All Microsoft® Project objects are accessed through the Application object. Application object methods represent the common command functionality of the user interface. These methods are used for the basic Project commands. The Application object is the parent of the Cell, Project, Selection, and Window objects, as well as the Projects and Windows collections.

For example, to open a new project from a template file:

```
Sub Tester()
Application.FileOpen Name:="LANSUB.MPT"
End Sub
```

Using the Application object to refer to Project objects is optional when writing macros. This returns the same result as the previous example:

```
Sub Tester()
FileOpen Name:="LANSUB.MPT"
End Sub
```

To move the active cell, use one of the following options, SelectCellDown, SelectCellRight, SelectCellLeft, SelectCellUp, or SelectColumn.

```
Sub Tester()
For I = 1 To 10
SelectCellRight I
SelectCellDown I
SelectCellLeft I
SelectCellUp I
Next I
SelectColumn
End Sub
```

Application object properties describe the Project environment. These properties are used to manage settings in the Options and Leveling dialog boxes. Although not all properties are editable, many are:

```
Sub TitleBarExample()
MsgBox Application.ActiveWindow
Application.ActiveWindow.Caption = "CPM is Cool"
End Sub
```

Working with Microsoft Publisher Objects

When you write Microsoft® Visual Basic® for Applications (VBA) code that calls into Microsoft® Publisher, you treat Publisher as a collection of Visual Basic objects, where each object has methods and properties that either return Publisher's state or cause Publisher to do something. The object model is the entire interface you see – all of the Publisher functionality is somewhere in the tree of exposed objects. This nicely encapsulates the functionality of Publisher in an easy-to-browse, standard way, so anyone who has programmed (for example) Microsoft® Word has a good chance of understanding Publisher. It provides an object-oriented face to Publisher. A number of concepts and technologies make it possible for Publisher to respond to Visual Basic, including: IDispatch, reference counting, object creation, type library interpreting, error handling, lifetime management, and collection enumeration.

In This Section

Understanding the Publisher Application Object
Use properties or methods of the Application object to control or return the Microsoft® Publisher application-wide attributes, to control the appearance of the application window, and to get to the rest of the Publisher object model.

Understanding the Publisher Application Object

Use properties or methods of the Application object to control or return the Microsoft® Publisher application-wide attributes, to control the appearance of the application window, and to get to the rest of the Publisher object model. In addition, COM add-ins can attach to the Application object.

Every time you write Microsoft® Visual Basic® for Applications (VBA) code in Publisher or write code to automate Publisher from some other application, you begin with the Application object. From the Application object, you can access all the other objects exposed by the application, as well as properties and methods unique to the Application object itself.

Application Object

Represents the Microsoft Publisher application. The **Application** object includes properties and methods that return top-level objects. For example, the **ActiveDocument** property returns a **Document** object.

Using the Application object

Use the **Application** property to return the **Application** object. The following example displays the application name.

```
Sub ShowAppName()
   MsgBox Application.Name
End Sub
```

Remarks
When using Visual Basic for Applications in Microsoft Publisher, all of the properties and methods of the **Application** object can be used without the **Application** object qualifier. For example, instead of typing Application.ActiveDocument.PrintOut, you can type ActiveDocument.PrintOut. Properties and methods that can be used without the **Application** object qualifier are considered "global." To view the global properties and methods in the Object Browser, click **<globals>** at the top of the list in the **Classes** box. When accessing the Publisher object model from a non-Publisher project, all properties and methods must be fully qualified.

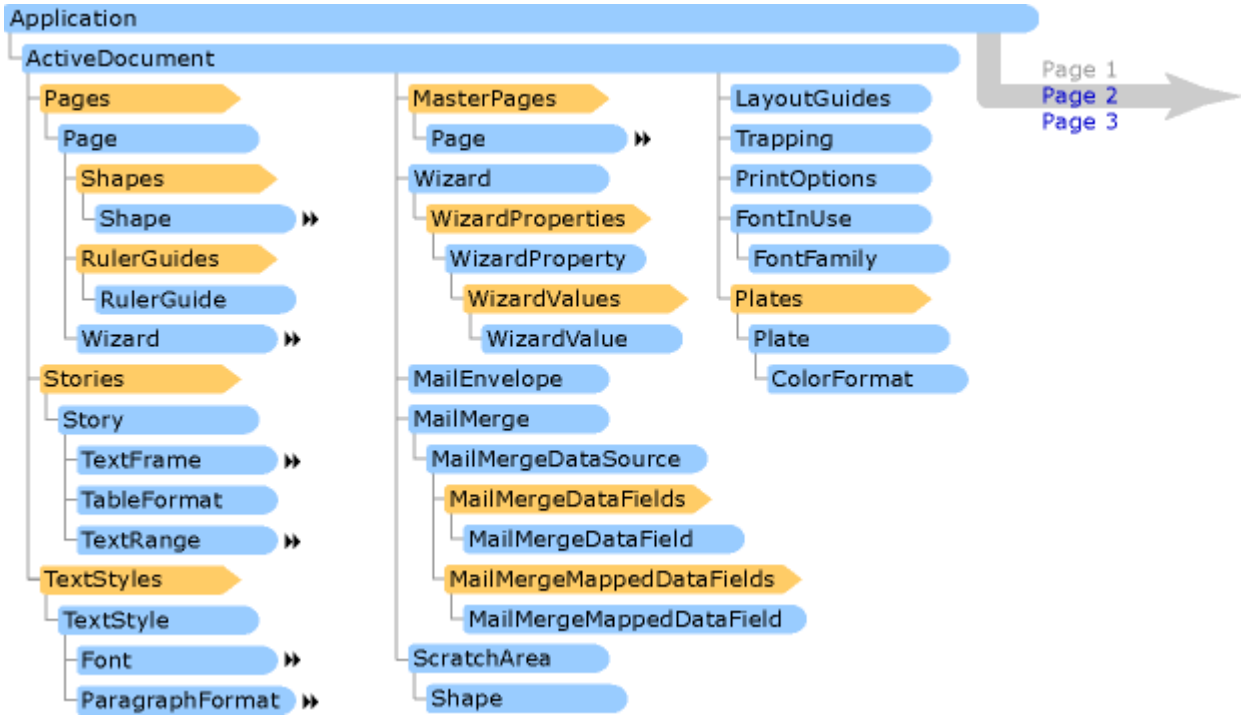


Figure 1. Microsoft Publisher Object Model (page 1)

Figure 2. Microsoft Publisher Object Model (page 2)

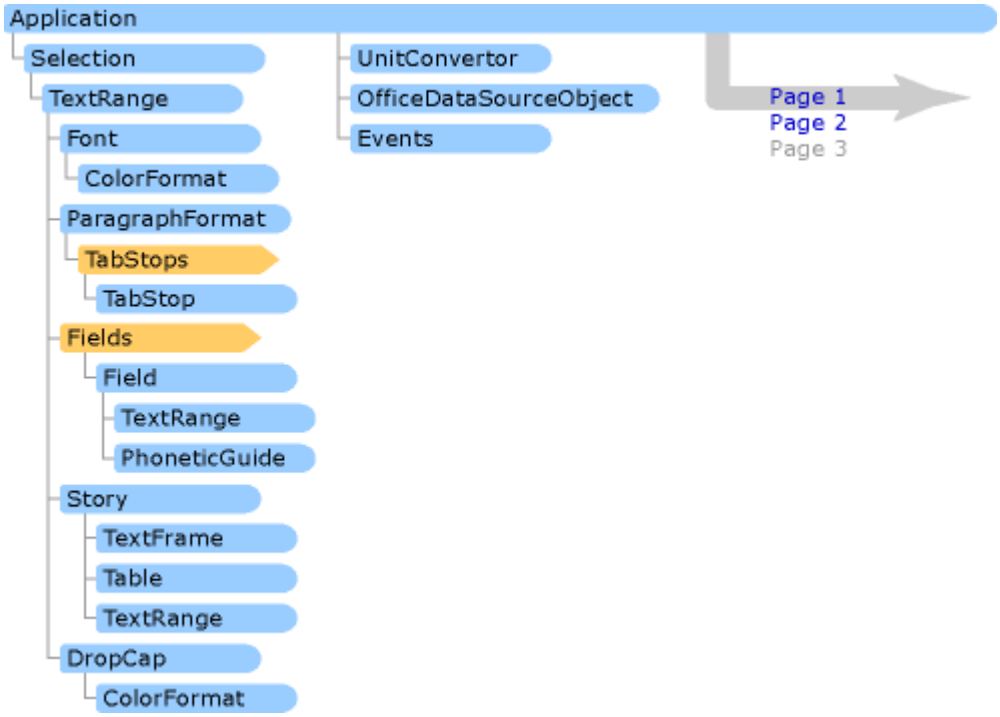


Figure 3. Microsoft Publisher Object Model (page 3)

Assistant Object

Represents the Microsoft Office Assistant.

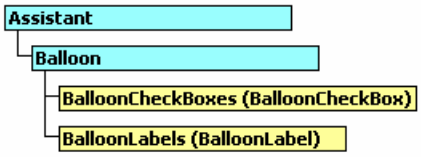


Figure 4. Assistent Object

Using the Assistant Object

Use the **Assistant** property to return the **Assistant** object. There isn't a collection for the **Assistant** object; only one **Assistant** object can be active at a time. Use the **Visible** property to display the Assistant, and use the **On** property to enable the Assistant.

Remarks

The default Assistant is Rocky. To select a different Assistant programmatically, use the **FileName** property. The following example displays and animates the Assistant.

```
With Assistant
    .Visible = True
    .Animation = msoAnimationGreeting
End With
```

Balloon Object

Represents the balloon where the Office Assistant displays information. A balloon can contain controls such as check boxes and labels.

Using the Balloon Object

Use the **NewBalloon** property to return a **Balloon** object. There isn't a collection for the **Balloon** object; only one balloon can be visible at a time. However, it's possible to define several balloons and display any one of them when needed. For more information, see "Defining and Reusing Balloons" later in this topic.

Use the **Show** method to make the specified balloon visible. Use the **Callback** property to run procedures based on selections from modeless balloons (balloons that remain visible while a user works in the application). Use the **Close** method to close modeless balloons.

The following example creates a balloon that contains tips for saving entered data.

```
With Assistant.NewBalloon
    .BalloonType = msoBalloonTypeBullets
    .Icon = msoIconTip
    .Button = msoButtonSetOk
    .Heading = "Tips for Saving Information."
    .Labels(1).Text = "Save your work often."
    .Labels(2).Text = "Install a surge protector."
```

```
    .Labels(3).Text = "Exit your application properly."
    .Show
End With
```

Defining and Reusing Balloons

You can reuse balloon objects you've already created by assigning the object to a variable and displaying the variable when you need it. This example defines balloon1 and balloon2 separately so that they can be reused.

```
Set balloon1 = Assistant.NewBalloon
balloon1.Heading = "First balloon"
Set balloon2 = Assistant.NewBalloon
balloon2.Heading = "Second balloon"
balloon1.Show
balloon2.Show
balloon1.Heading = "First balloon, new heading"
balloon1.Show
```

BalloonCheckBoxes Object

Represents a check box in the Office Assistant balloon. The **BalloonCheckBox** object is a member of the **BalloonCheckBoxes** collection.

Using the BalloonCheckBox

Use the **CheckBoxes** property to return the **BalloonCheckBoxes** collection.

Use **CheckBoxes**(*index*), where *index* is a number from 1 through 5, to return a single **BalloonCheckBox** object. You can specify up to five check boxes (and five labels) per balloon; each check box appears when a value is assigned to its **Text** property. If you specify more than five check boxes, a run-time error occurs.

The following example creates a balloon with a heading, text, and three region choices. When the user selects one or more check boxes and then clicks **OK**, the specified procedure or procedures are called.

```
With Assistant.NewBalloon
    .Heading = "Regional Sales Data"
    .Text = "Select your region"
    For i = 1 To 3
        .CheckBoxes(i).Text = "Region " & i
    Next
    .Button = msoButtonSetOkCancel
    .Show
    If .CheckBoxes(1).Checked Then
        runregion1
    End If
    If .CheckBoxes(2).Checked Then
        runregion2
    End If
    If .CheckBoxes(3).Checked Then
        runregion3
    End If
End With
```

You cannot add check boxes to or remove check boxes from the **BalloonCheckBoxes** collection after the balloon has been displayed.

BalloonLabel Object

Represents a label in the Office Assistant balloon. The **BalloonLabel** object is a member of the **BalloonLabels** collection.

Using the BalloonLabel Object

Use **Labels**(*index*), where *index* is a number from 1 through 5, to return a **BalloonLabel** object. There can be up to five labels on one balloon; each label appears when a value is assisgned to its **Text** property.

The following example creates a balloon that asks the user to click the label corresponding to his or her age.

```
With Assistant.NewBalloon
    .Heading = "Check Your Age Group."
    .Labels(1).Text = "Under 30."
    .Labels(2).Text = "30 to 50."
    .Labels(3).Text = "Over 50."
    .Text = "Which of the following " & .Labels.Count & " choices apply to you?"
```

```
    .Show
End With
```

Remarks
Balloon check boxes display the user's choices until he or she dismisses the balloon. You can use balloon labels to return a number corresponding to the user's choice in the **Select** method as soon as the user clicks the button beside the label. To pass values to the **Select** method based on the user's choice, you must have the balloon type be set to **msoBalloonTypeButtons**.

Document Object

Represents a publication. Since Microsoft Publisher works with only one publication at a time, there is no **Documents** collection.

Using ActiveDocument

Use the **ActiveDocument** property to refer to the current publication. This example adds a table to the first page of the active publication.

```
Sub NewTable()
   With ActiveDocument.Pages(1).Shapes
      .AddTable NumRows:=3, NumColumns:=3, Left:=72, Top:=300, Width:=488, Height:=36
      With .Item(1).Table.Rows(1)
         .Cells(1).TextRange.Text = "Column1"
         .Cells(2).TextRange.Text = "Column2"
         .Cells(3).TextRange.Text = "Column3"
      End With
   End With
End Sub
```

You could also write the above routine using a reference to the **ThisDocument** module. This example uses a **ThisDocument** reference instead of **ActiveDocument**.

```
Sub PrintPublication()
   With ThisDocument.Pages(1).Shapes
      .AddTable NumRows:=3, NumColumns:=3, Left:=72, Top:=300, Width:=488, Height:=36
      With .Item(1).Table.Rows(1)
         .Cells(1).TextRange.Text = "Column1"
         .Cells(2).TextRange.Text = "Column2"
         .Cells(3).TextRange.Text = "Column3"
      End With
   End With
End Sub
```

ActiveDocument Property

Returns a **Document** object that represents the active publication. If there are no documents open, an error occurs.

expression.**ActiveDocument**
expression Required. An expression that returns one of the objects in the Applies To list.

This example allows the user to assign a file name to the active publication and save it with the new file name. The file name, along with other text, is then inserted after the currently selected text.

```
Sub NewsLetterSave()
   Dim strFileName As String

   ' Assign the explicit file name to a variable.
   strFileName = "NewsLetter3.pub"
   Publisher.ActiveDocument.SaveAs strFileName
   ' Insert the file name and supporting text after selected text.
   Selection.TextRange.Collapse pbCollapseEnd
   Selection.TextRange = " This publication has been saved as " & strFileName
End Sub
```

Selection Object

Represents the current selection in a window or pane. A selection represents either a selected (or highlighted) area in the publication, or it represents the insertion point if nothing in the publication is selected. There can only be one **Selection** object per publication window pane, and only one **Selection** object in the entire application can be active.

Using the Selection Object

Use the **Selection** property to return the **Selection** object. If no object qualifier is used with the **Selection** property, Microsoft Publisher returns the selection from the active pane of the active publication window. The following example copies the current selection from the active publication.

```
Sub CopySelection()
   Selection.ShapeRange.Copy
End Sub
```

The following example determines what type of item is selected and if it is an autoshape, fills the first shape in the selection with color. This example assumes there is at least one item selected in the active pubication.

```
Sub SelectedShape()
   If Selection.Type = pbSelectionShape Then
      Selection.ShapeRange.Item(1).Fill.ForeColor _
         .RGB = RGB(Red:=200, Green:=20, Blue:=255)
   End If
End Sub
```

The following example copies the selection and pastes it into the first shape on the second page of the active publication.

```
Sub CopyPasteSelection()
   Selection.TextRange.Copy
   With ActiveDocument.Pages(2).Shapes(1).TextFrame.TextRange
      .Collapse Direction:=pbCollapseEnd
      .InsertAfter NewText:=vbLf
      .Paste
   End With
End Sub
```

Working with Microsoft Visio Objects

Microsoft® Visio® products provide a sophisticated toolset for information technology professionals who design, model, and manage complex enterprise-level systems.

Automation is a means by which a program written in Microsoft® Visual Basic® for Applications (VBA), or other computer languages that support automation, can incorporate the functionality of the Visio applications simply by using its objects. VBA is incorporated in Visio, so you are not required use a separate development environment to write your programs.

The way objects in an application are related to each other, along with each object's properties (data), methods (behavior), and events, is called the program's object model. In the Visio object model, most objects correspond to items you can see and select in the Visio user interface. For example, a Shape object represents a shape in a drawing.

In automation, the application that provides the objects (such as the Visio application) makes the objects accessible to other applications, and provides the properties and methods that control them.

The application that uses the objects (such as your program) creates instances of the objects and sets their properties or invokes their methods to make the objects serve the application.

In This Section

Understanding the Visio Object Model
The Microsoft® Visio® object model represents the objects, properties, methods, and events that the Visio engine exposes through automation.

Understanding the Visio Application Object
The Application object is a property of the Microsoft® Visio® global objects, so you can access any of the Application object's properties by referencing directly the Application property of the Visio global object.

Understanding the Visio Object Model

The Microsoft® Visio® object model represents the objects, properties, methods, and events that the Visio engine exposes through automation. More important, it describes how the objects are related to each other.

Most objects in the model correspond to items you can see and select in the Visio user interface. For example, a Shape object can represent anything on a Visio drawing page that you can select with the pointer tool-a shape, a group, a guide, or an object from another application that is linked, embedded, or imported into a Visio drawing.

Visio objects reside in an instance of the Visio application. Microsoft® Visual Basic®for Applications (VBA) code runs within an instance of Visio and accesses the objects it requires. An external program runs outside an instance of the Visio application, so it starts the Visio application or accesses an instance of Visio that is running, and then it accesses the Visio object it needs.

Some objects represent a collection of other objects. A collection contains zero or more objects of a specified type. For example, a Document object represents one open document in an instance of Visio; the Documents collection represents all of the documents that are open in the instance.

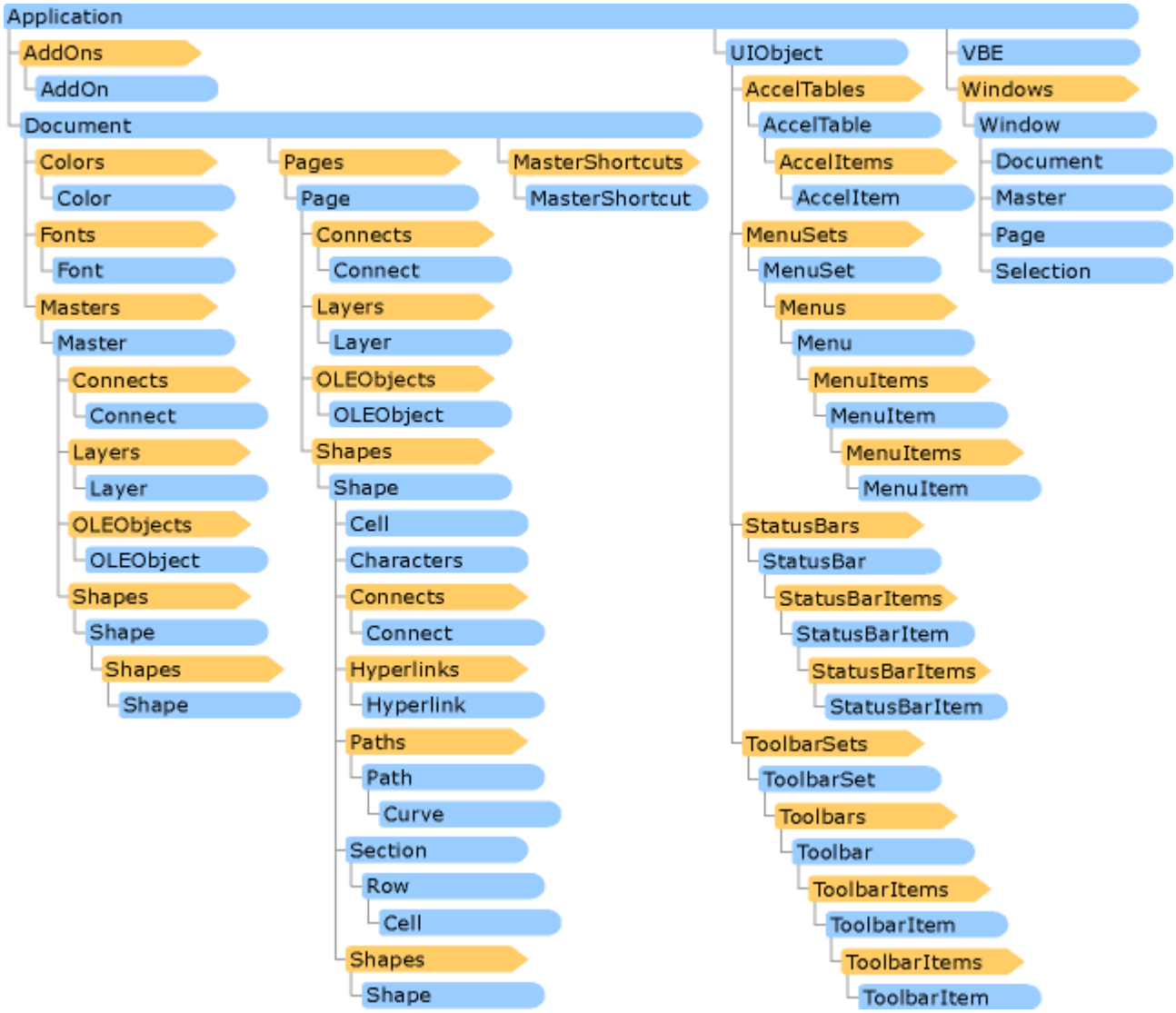


Figure 1. Microsoft Visio Object Model

Using Visio Object Types

You can take advantage of the Visio type library to write code more effectively. By using Visio object types declared in the type library, you can declare variables as specific types, such as Visio.Page, which is illustrated in the following code:

```
Dim pagObj As Visio.Page
```

This example uses Visio to inform the program that it is referencing Visio object types in the Visio type library, and it uses Page to inform the program that the pagObj variable is a Page object. Here are a few more object types:

```
Dim docsObj As Visio.Documents  'A Documents collection
Dim docObj As Visio.Document    'A Document object
Dim shpsObj As Visio.Shapes     'A Shapes collection
Dim shpObj As Visio.Shape       'A Shape object
Dim mastObj As Visio.Master     'A Master object
```

Getting and Releasing Visio Objects

You get an object by declaring an object variable, navigating through the object model to get a reference to the object you want to control, and assigning the reference to the object variable. After you have a reference to an object, you can get and set the values of its properties or use methods that cause the object to perform actions.
The following are some guidelines for getting and releasing Visio objects:

Declare object variables Declare a Visio object type as defined in the Visio type library. Use the Set statement to assign the reference to the object variable.

Access Visio objects through properties Most Visio objects have properties whose values refer to other objects. You can use these properties to navigate up and down the layers of the object model to get to the object you want to control.

Refer to an object in a collection A collection is an object that represents objects of a particular type. You can get a reference to a particular object in the collection. The Item property returns a reference to an object in the collection.

Iterate through a collection A collection's Count property returns the number of objects in the collection. Most often, you will use the Count property to set the limit for an iteration loop.

Release an object An object in a collection is released automatically when the program finishes running or when all object variables referring to that object go out of scope.

Use compound object references You can concatenate Visio object references, properties, and methods in single statements as you can with Microsoft® Visual Basic® for Applications (VBA) objects. However, simple references are sometimes more efficient, even if they require more lines of code.

Restrict the scope and lifetime of object variables You can prevent invalid references by restricting the scope and lifetime of an object variable. For example, when your program resumes execution after giving control to the user, you can release certain objects and retrieve them again to make sure that the objects still are available and your program has references to the objects in their current state.

Understanding the Visio Application Object

Unlike a stand-alone program, which must obtain a reference to the Microsoft® Visio® Application object by creating it or getting it, code in a Microsoft® Visual Basic® for Applications (VBA) project executes in a running instance of Visio. Therefore, you are not required to obtain a reference to the Application object. The Visio engine provides the global object, which represents the Visio instance. In addition, the Visio engine provides the ThisDocument object, which represents the Visio document associated with your project.

The global object represents the instance and provides more direct access to certain properties. The properties of the Visio global object are not prefixed with a reference to an object.

The Application object is a property of the Visio global objects, so you can access any of the Application object's properties by referencing the Application property of the Visio global object directly.

The following are three examples of code that get the first document in a Documents collection-all three use different syntax.

Example 1 creates an Application object. Typically, this code is used when writing an external program:

Example 1
Dim appVisio As Visio.Application
Dim docsObj As Visio.Documents
Dim docObj As Visio.Document
Set appVisio = CreateObject("visio.application")
Set docsObj = appVisio.Documents
Set docObj + docsObj.Item(1)

Example 2 uses the Application property of the Visio global object:

Example 2
Dim docsObj As Visio.Documents
Dim docObj As Visio.Document
Set docsObj = Application.Documents
Set docObj = docsObj.Item(1)

Example 3 directly accesses the Documents property of the Visio global object:

Example 3
Dim docObj As Visio.Document
Set dicObj + Documents.Item(1)

You might have noticed in examples 2 and 3 that Application and Documents are not preceded by an object. When you are referencing any property or method of the Visio global object, you are not required to declare a variable for the global object or reference it as the preceding object of a property-the global object is implied. The third example is the most direct method of accessing the Documents collection from a VBA Project.

The following are some of examples of code for commonly used properties of the Visio global object.

Set docObj = ActiveDocument
Set pagObj = ActivePage
Set WinObj = ActiveWindow

Note
The Visio global object is available only when you are writing code in the VBA project of a Visio document.

Working with Microsoft Word Objects

In Microsoft® Word, the fundamental working object is a document and everything is part of the document. When you are using VBA to work with Word, a Document object represents an open document, and all Document objects are contained in the Application object's Documents collection. Because each Document object is based on a template, each document has an AttachedTemplate property.

A document is a collection of characters arranged into words, words are arranged into sentences, sentences are arranged into paragraphs, and so on. Therefore, each Document object has a Characters collection, a Words collection, a Sentences collection, and a Paragraphs collection. Furthermore, each document has a Sections collection of one or more sections, and each section has a HeadersFooters collection that contains the headers and footers for the section. In addition, some or all of the text in the document might have certain formatting attributes set, and paragraphs might have built-in or custom styles applied.

In This Section

Understanding the Word Application Object
Understanding Application-Level Objects
Learn about the Application object and other objects that affect Word.

The Document Object
Create or open documents, and add them to the Documents collection.

Working with Document Content
Work with objects contained in the Document object.

Understanding the Word Application Object

Figure 1. Microsoft Word Object Model (page 1)

Figure 2. Microsoft Word Object Model (page 2)

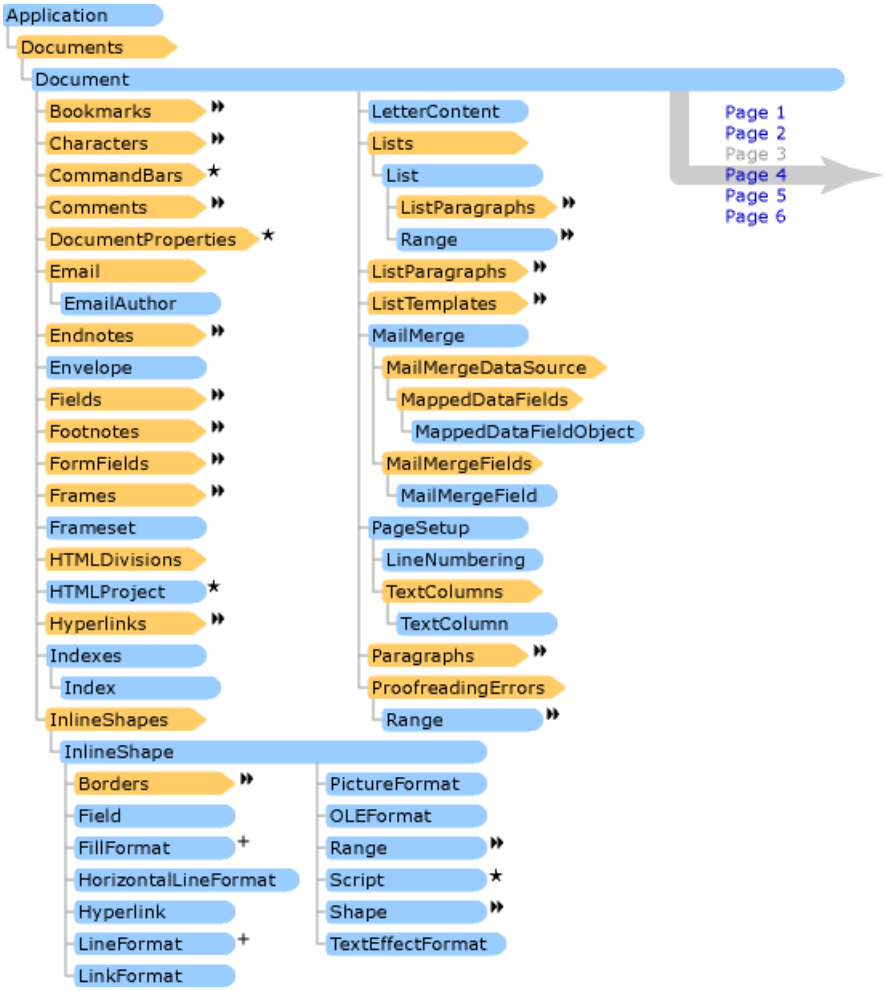


Figure 3. Microsoft Word Object Model (page 3)

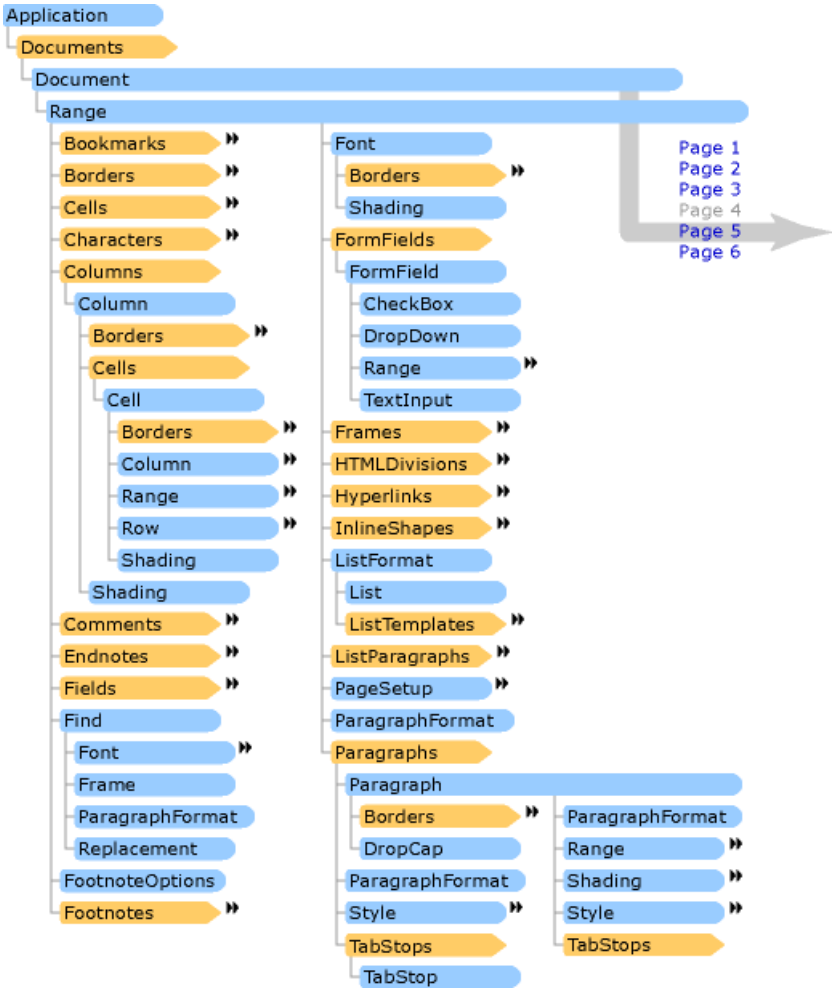


Figure 4. Microsoft Word Object Model (page 4)

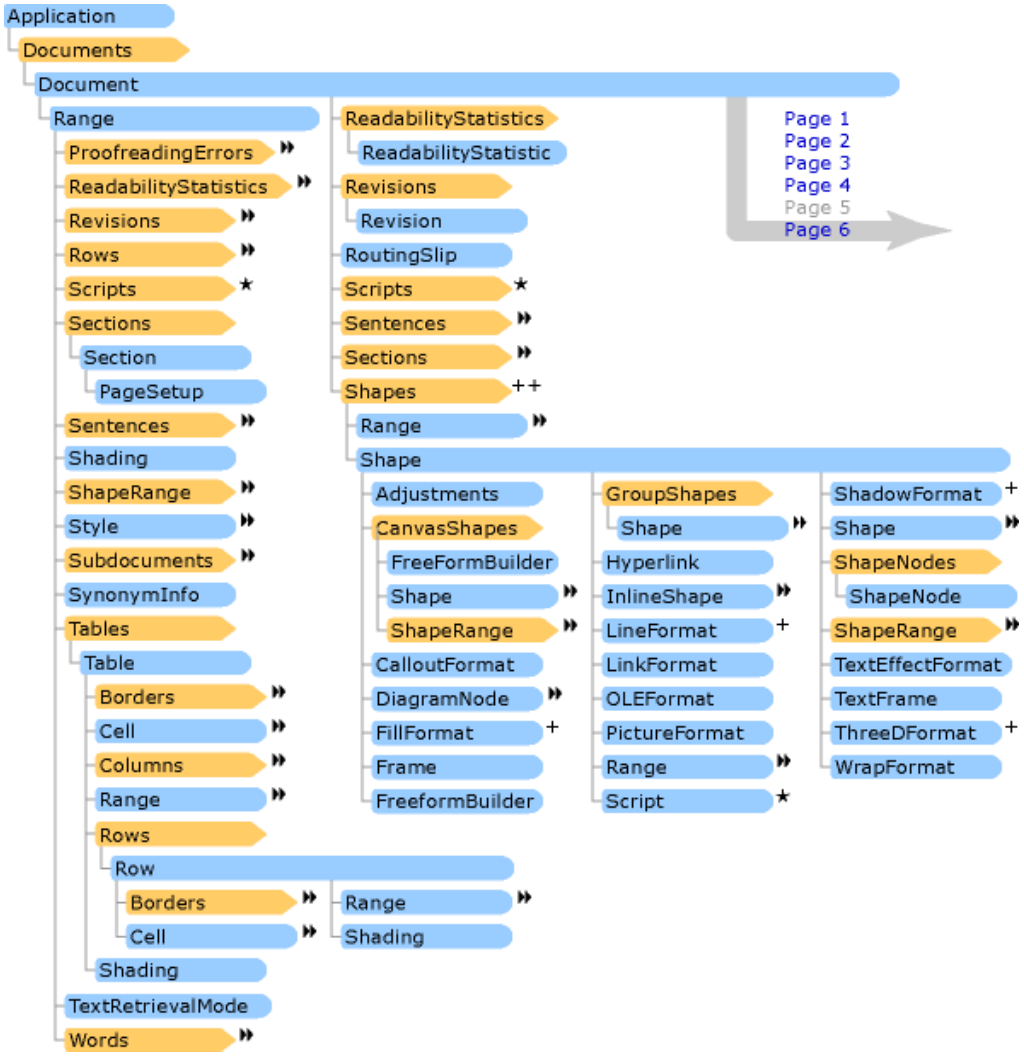


Figure 5. Microsoft Word Object Model (page 5)

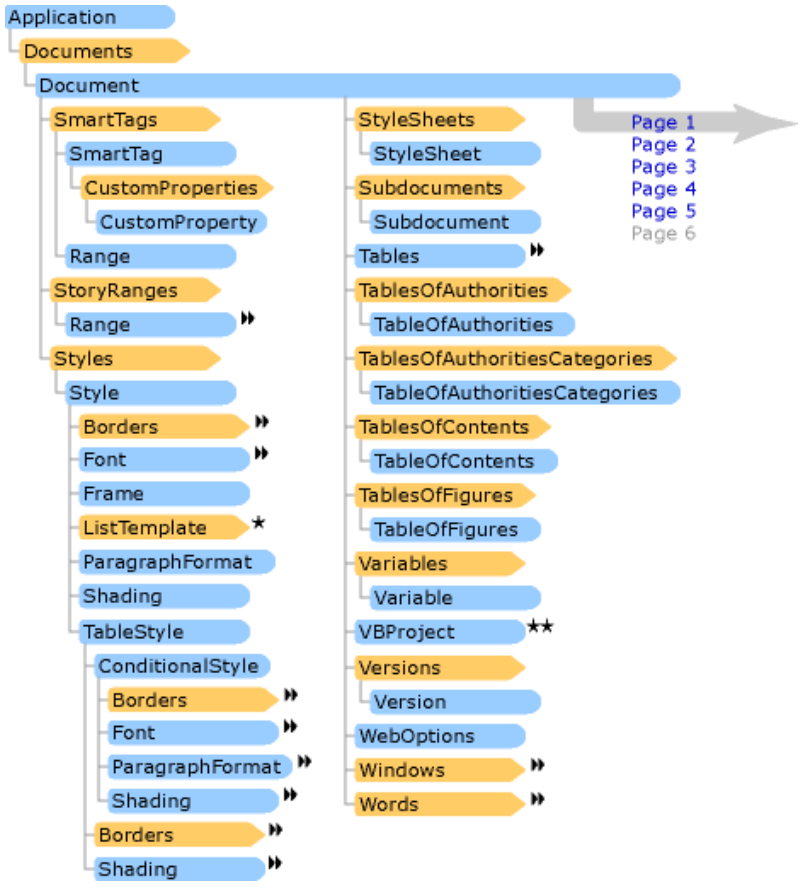


Figure 6. Microsoft Word Object Model (page 6)

Understanding Application-Level Objects

Application-level objects are the Application object itself and its properties, methods, options (displayed in the Options dialog box), as well as the built-in dialog boxes in Word-in other words, these are objects that can affect more than one document at a time or are accessed and manipulated independent of the currently active Document object.

In This Section

Working with the Application Object
Access all the other objects exposed by the application as well as properties and methods unique to the Application object itself.

Working with the Settings in the Options Dialog Box
Learn how to use the Options, View, and Dialog options.

Working with Word Dialog Boxes
Create your own custom dialog boxes by using the functionality of one of the Word built-in dialog boxes.

Modifying Built-in Commands
Customize the behavior of Word by running your own VBA procedure in place of a built-in procedure.

Working with the Application Object

Every time you write VBA code in Word, or write code to automate Word from some other application, you begin with the Application object. From the Application object, you can access all the other objects exposed by the application as well as properties and methods unique to the Application object itself.

Note
If you are working in Word, the Application object is created for you, and you can use the Application property to return a reference to the Word Application object. If you are automating Word from some other application, you must create a Word Application object variable and then create an instance of Word.

To access properties and methods of the Application object, you use the following syntax:

Application.PropertyName
Application.MethodName (arg1, arg2, argN)

You can access child objects of the Application object by using the following syntax:

Application.ObjectName
-or-
ObjectName

Note
You do not have to use the Application property in this context because these objects are global.

Working with the Settings in the Options Dialog Box

The Options dialog box contains many settings that let you customize the way Word looks and behaves. You can view this dialog box by clicking Options on the Tools menu. To programmatically access the settings in this dialog box, you use the Options object or the View object of a Window object. You can also access these settings through the Dialog object that represents the tab in the Options dialog box that contains the setting you want to manipulate.

Using the Options Object

The Options object contains many properties that represent items in the Options dialog box. For example, the Options object's ReplaceSelection property setting is equivalent to the Typing replaces selection setting on the Edit tab of the Options dialog box. The easiest way to identify which properties of the Options object represent settings on a tab in the Options dialog box is to record a macro that changes a setting and then examine the property settings Word records. For example, the Print tab of the Options dialog box contains fourteen settings. Two settings (Print PostScript over text and Print data only for forms) apply only to the active document and are therefore properties of the Document object. The remaining settings represent properties of the Options object. A macro that records a change to a setting on the Print tab would create, in part, the following list of Options object properties:

```
With Options
  .UpdateFieldsAtPrint = False
  .UpdateLinksAtPrint = False
  .DefaultTray = "Use printer settings"
  .PrintBackground = True
  .PrintProperties = False
  .PrintFieldCodes = False
  .PrintComments = True
  .PrintHiddenText = True
  .PrintDrawingObjects = True
  .PrintDraft = False
  .PrintReverse = False
  .MapPaperSize = True
End With
```

If you change the settings of Options object properties, make sure you return each setting to its original value when you are finished. Many of these properties are global application-level settings, and you might be making changes that the user would not want persisted. The following example illustrates how to return Options object properties to their original settings when a procedure that changes those properties ends:

```
Sub PrintAllDocInfo()
  ' This procedure illustrates how to use the Options object  to change certain settings, print a document, and then
  ' return the settings to their original state.
  Dim blnProps        As Boolean
  Dim blnFields       As Boolean
  Dim blnComments     As Boolean
  Dim blnHidden       As Boolean
    With Options
    ' Save the existing property settings.
    blnProps = .PrintProperties
    blnFields = .PrintFieldCodes
    blnComments = .PrintComments
    blnHidden = .PrintHiddenText
    ' Set properties to True and print document.
    .PrintProperties = True
    .PrintFieldCodes = True
    .PrintComments = True
    .PrintHiddenText = True
    Application.PrintOut
    ' Return properties to original settings.
    .PrintProperties = blnProps
    .PrintFieldCodes = blnFields
    .PrintComments = blnComments
    .PrintHiddenText = blnHidden
  End With
End Sub
```

Using the View Object and the Dialog Object

The View object lets you determine or specify all the attributes of a Window object. For example, you can run the following code sample from the Immediate window in the Microsoft® Visual Basic® Editor to determine whether hidden text is displayed in the current document:

```
? ActiveWindow.View.ShowHiddenText
```

Note that although many of the View object properties map directly to settings in the Options dialog box, they do not necessarily map to settings on the View tab of that dialog box.

You use the Dialogs collection to access a Dialog object that represents a tab in the Options dialog box. For example, if you execute the following code from the Immediate window, it prints the current setting for the Typing replaces selection check box on the Edit tab of the Options dialog box:

```
? CBool(Dialogs(wdDialogToolsOptionsEdit).ReplaceSelection)
```

Working with Word Dialog Boxes

You can create your own custom dialog boxes by using UserForms, but before you do, you should determine whether you could simply appropriate the functionality of one of Word's more than two hundred built-in dialog boxes.

From VBA, you access any of Word's built-in dialog boxes through the Dialogs collection. The Dialogs collection is a global object, so you can reference it without specifying the Application property. For example, you can run the following code from the Immediate window to return the number of dialog boxes in the Dialogs collection:

```
? Dialogs.Count
```

To work with a particular dialog box, you create an object variable declared As Dialog and use one of the wdWordDialog constants to specify the dialog box you want to reference. For example, the following code creates a reference to the Spelling and Grammar dialog box:

```
Dim dlgSpell As Dialog

Set dlgSpell = Dialogs(wdDialogToolsSpellingandGrammar)
```

When you instantiate a Dialog object variable in this way, you can easily determine or specify the various dialog box settings. When you refer to one of these settings in VBA, you can reference it as a property of the dialog box. For example, you can refer to the All setting on the View tab of the Options dialog box by using the ShowAll property:

```
MsgBox "The 'All' setting on the View tab is currently set to "  & CBool(Dialogs(wdDialogtoolsOptionsView).ShowAll)
```

Dialog box properties are typically set from the user interface by using check box controls, combo box controls, or text box controls. Check box controls contain the value 1 when they are selected and 0 when they are not selected. Combo box controls

contain the index value of the item selected, beginning at 0 for the first item in the control. Text box controls contain a String value representing the text in the control.

You also have control over how a dialog box is displayed and when changes to settings take effect. When you display a dialog box from the Word user interface and change a setting, the change usually takes effect as soon as you click the dialog box's OK button, although some dialog box settings take effect immediately. When you use VBA to display a dialog box, you can control how the dialog box behaves by using either the Dialog object's Show method or its Display method. If you use the Show method, the dialog box behaves just as it does when Word displays it. The Display method simply displays the dialog box and you must use additional VBA code to take further action in response to any selections made in the dialog box by the user. Both methods also return a value representing whether the user clicked OK, Cancel, Close, or some other button in the dialog box.

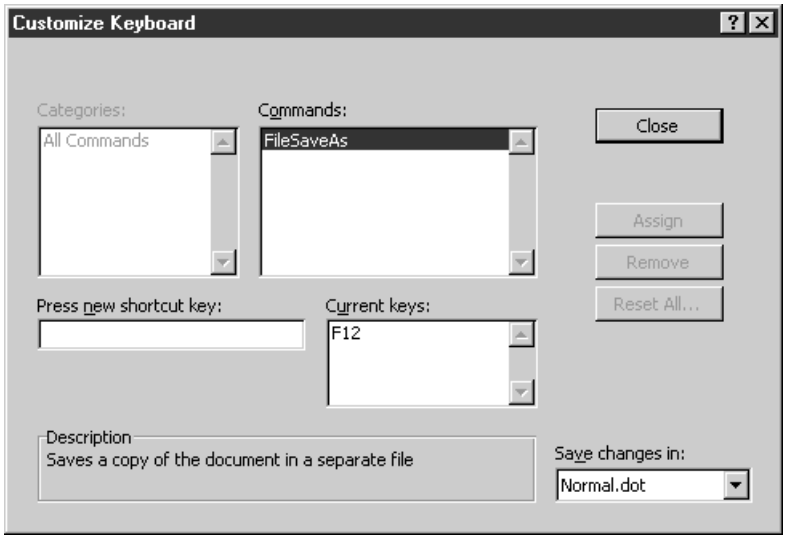
Modifying Built-in Commands

One simple but very powerful method you can use to customize the way Microsoft® Word works is to run your own VBA procedure in place of a built-in procedure. Doing this lets you customize the behavior of Word in any way you can imagine.

There is no limit to the kinds of things you can do and the kinds of built-in behaviors you can change. You could save documents created by using your custom template to a different directory than documents created by using Normal.dot. You could modify the File New command to create custom document properties for every new document. You could display your own custom dialog box instead of the Word dialog box normally displayed in response to a menu command. You could also let the built-in command run and then detect whether a user made certain selections from a Word dialog box.

The first thing you have to do is figure out which procedure Word runs to perform a built-in action. This is easy to do for all built-in menu commands. If you press ALT+CTRL+PLUS SIGN (+) on the numeric keypad (not the PLUS SIGN on the keyboard) and then click the menu item you want to investigate, Word displays the Customize Keyboard dialog box, which shows the name of the built-in procedure in the Commands list. For example, in the following figure, you can see that Word runs the FileSaveAs procedure whenever a user clicks Save As on the File menu.

The Customize Keyboard Dialog Box



There are three ways you can substitute your own procedure for a built-in Word procedure:

- In any standard module, create a VBA procedure that uses the same name as the procedure you want to replace. For example, if you create a procedure named FileSaveAs, Word will run your procedure instead of the built-in FileSaveAs procedure whenever the built-in procedure would normally be called.
- Create a module and name it by using the name of the built-in command you want to replace. Then add a subroutine named Main( ) to the module and add your custom code to that procedure.
- Create a new procedure by using the Macros dialog box. To do this, point to Macro on the Tools menu, and then click Macros. In the Macros dialog box, click Word Commands in the Macros in list. The Macro name list will then display the hundreds of built-in Word procedures. You can learn something about what these procedures do by clicking a procedure name in the list and reading its description in the Description box at the bottom of the dialog box. When you locate the command you want to modify, click it and then use the Macros in list to select the template or document in which to save the procedure. Then click the Create button to create a new VBA procedure that uses the same name as the built-in command.

The Document Object

The Document object is just below the Application object in Word's object model and is at the heart of Word programming. When you open a new document from the user interface, you create a new Document object. Each document you create or open is added to the Documents collection, and the document that has the focus is called the active document.

In This Section

Working with the Document Object
Create or open documents and add them to the Documents collection.

Opening, Creating, Saving, and Closing New Documents
Create and manipulate new documents by using the Documents collection's methods.

Working with the Document Object

You can reference a Document object as a member of the Documents collection by using either its index value (where 1 is the first document in the collection) or its name. In addition, you can use the ActiveDocument property to return a reference to the document that currently has the focus. For example, if a document named Policies.doc is the only open document, the following three object variables will all point to Policies.doc:

```
Dim docOne     As Word.Document
Dim docTwo     As Word.Document
Dim docThree   As Word.Document

Set docOne = Documents(1)
Set docTwo = Documents("Policies.doc")
Set docThree = ActiveDocument
```

You will rarely refer to a document by using its index value in the Documents collection because this value can change for a given document as other documents are opened and closed. Typically, you will use the ActiveDocument property or a Document object variable created by using the Documents collection's Add method or Open method. The following example shows how you can use the ActiveDocument property to add an address to the document that currently has the focus:

```
Sub AddOPGAddress()
  With ActiveDocument
    .Envelope.Insert Address:="The MOD Team"  & vbCrLf & "One Microsoft Way" & vbCrLf _
      & "Redmond, WA 98052", ReturnAddress:= "One Happy Customer" & vbCrLf & _
      "77 Pine Bough Lane" & vbCrLf & "Any Town, USA 12345"
  End With
End Sub
```

The next example illustrates how to create an instance of a Document object variable by using the Documents collection's Open method. After the Document object variable is set, the code calls the procedure from the prior example to add an envelope and then the envelope and the document are printed. Finally, the document is closed and all changes are saved.

```
Dim docPolicy As Word.Document
Set docPolicy = Documents.Open("c:\my documents\policies.doc")
With docPolicy
  Call AddOPGAddress
  .Envelope.PrintOut
  .PrintOut
  .Close SaveChanges:=True
End With
```

Note
The document opened by using the Open method or the document created by using the Add method will also be the currently active document represented by the ActiveDocument property. If you want to make some other document the active document, use the Document object's Activate method.

Opening, Creating, Saving, and Closing New Documents

You use the Documents collection's Open method to open an existing document. The FileName argument can include the full path to the file or the file name alone. If the file specified in the FileName argument does not include the full path to the document, Word looks for the document in the current directory. If Word can't find the file by using the file path and file name specified in the FileName argument, an error occurs.

Instead of using a hard-coded path and file name, you can use the FileSearch object to make sure the file exists before trying to open it. You can also create a Dialog object that represents the File Open dialog box and use it to let the user select the file name to use as the FileName argument of the Open method.

You create a new document by using the Documents collection's Add method. The Add method can accept up to two optional arguments. You use the Template argument to specify the template on which to base the new document. If you leave this argument blank, the new document is based on the Normal.dot template. The NewTemplate argument is a Boolean value that specifies whether to create the new document as a template. The following example creates a new document based on the Normal.dot template:

```
Dim docNew As Word.Document

Set docNew = Documents.Add
With docNew
  ' Add code here to work with the new document.
End With
```

The method you use to save a document depends on whether the document is new or has already been saved. To save an existing document, you use the Document object's Save method. To save a new document, you use the Document object's SaveAs method and specify a file name in the method's FileName argument. If you use the Save method on a new document, Word displays the Save As dialog box to prompt the user to give the document a name.

You can also save a new document as soon as it is created by using the Add method and the SaveAs method together as follows:

```
Documents.Add.SaveAs FileName:="c:\my documents\fastsave.doc"
Set docNew = Documents("fastsave.doc")
```

If you use the Documents.Add.SaveAs syntax, you will not be able to set a Document object variable at the same time you use the Add method. Instead, you can refer to the newly created document by using the ActiveDocument property or by using the document's name in the Documents collection, as shown in the preceding example.

To close a document, you use the Document object's Close method. If there are changes to the document and you do not specify the SaveChanges argument, Word prompts the user to save changes. To prevent this prompt from appearing, use either the wdDoNotSaveChanges or the wdSaveChanges built-in constant in the Close method's SaveChanges argument. To close all open documents at once, use the Documents collection's Close method and either the wdDoNotSaveChanges or the wdSaveChanges constant in the SaveChanges argument.

Working with Document Content

When you have a document to work with, most of the tasks you'll want to perform with VBA will involve working with the text in the document or manipulating the objects contained in the document. Documents contain words, sentences, paragraphs, sections, headers and footers, tables, fields, controls, images, shapes, hyperlinks and more. All of these objects are available to you through VBA.

The starting point for much of what you do to the contents of a document will be to specify a part of the document and then to do something to it. This might involve, for example, adding or removing text or formatting words or characters. The two objects you will use to accomplish much of this work are the Range object and the Selection object.

In This Section

The Range Object
Understand the Range object, including creating, defining, determining the location of, and working with text in a Range object.

The Selection Object
Learn about the Selection object, including how to use its Type property to get information about the state of the current selection.

The Selection Object vs. the Range Object
Compare and contrast the Selection and Range objects.

Working with Bookmarks
Use bookmarks to mark a location in a document or as a container for text in a document.

The Find and Replacement Objects
Loop through a document looking for some specific text, formatting, or style, and specify what you want to use to replace the item you found.

The Range Object

A Range object represents a contiguous area in a document, defined by a starting character position and an ending character position. The contiguous area can be as small as the insertion point or as large as the entire document. It can also be, but does not have to be, the area represented by the current selection. You can define a Range object that represents a different area than the current selection. You can also define multiple Range objects in a single document. The characters in a Range object include nonprinting characters, such as spaces, carriage returns, and paragraph marks.

Note
The area represented by the current selection is contained in the Selection object.

A Range object is similar to a Word bookmark in that they both define a specific area within a document. However, unlike a bookmark, a Range object exists only so long as the code that creates it is running. In addition, when you insert text at the end of a range, Word automatically expands the range to include the new text. When you insert text at the end of a bookmark, Word does not expand the bookmark to include the new text.

Creating, Defining, and Redefining a Range

You typically create a Range object by declaring an object variable of type Range and then instantiating that variable by using either the Document object's Range method or the Range property of another object, such as a Character, Word, Sentence, or Selection object. For example, the following code creates two Range objects that both represent the second sentence in the active document.

```
Public Sub GetRangeExample()
  ' This example shows how the Range method and the Range property both return the same characters.
  Dim rngRangeMethod      As Word.Range
  Dim rngRangeProperty    As Word.Range

  With ActiveDocument
    If .Sentences.Count >= 2 Then
       Set rngRangeMethod = .Range(.Sentences(2).Start, .Sentences(2).End)
       Set rngRangeProperty = .Sentences(2)
    End If
  End With
  Debug.Print rngRangeMethod.Text
  Debug.Print rngRangeProperty.Text
End Sub
```

When you use the Range method to specify a specific area of a document, you use the method's Start argument to specify the character position where the range should begin and you use the End argument to specify where the range should end. The first character in a document is at character position 0. The last character position is equal to the total number of characters in the document. You can determine the number of characters in a document by using the Characters collection's Count property. As shown in the preceding example, you can also use the Start and End properties of a Bookmark, Selection, or Range object to specify the Range method's Start and End arguments. You can set the Start and End arguments to the same number. In this case, you create a range that does not include any characters.

You can set or redefine the contents of a Range object by using the object's SetRange method. You can specify or redefine the start of a range by using the Range object's Start property or its MoveStart method. Likewise, you can specify or redefine the end of a range by using the Range object's End property or its MoveEnd method.

The following example begins by using the Content property to create a Range object that covers the entire contents of a document. It then changes the End property to specify that the end of the range will be at the end of the first sentence in the document. It then uses the SetRange method to redefine the range to cover the first paragraph in the document. Finally, it uses the MoveEnd method to extend the end of the range to the end of the second paragraph in the document. At each step in the example, the number of characters contained in the range is printed to the Immediate window.

```
Public Sub RedefineRangeExample1()
  ' This procedure illustrates how to use various properties  and methods to redefine the contents of a Range object.
  ' See also the RedefineRangeExample2 procedure.
  Dim rngSample As Range

  Set rngSample = ActiveDocument.Content
  With rngSample
    Debug.Print "The range now contains " & .Characters.Count & " characters."
    .End = ActiveDocument.Sentences(1).End
    Debug.Print "The range now contains " & .Characters.Count & " characters."
    .SetRange Start:=0, End:=ActiveDocument.Paragraphs(1).Range.End
    Debug.Print "The range now contains " & .Characters.Count & " characters."
    .MoveEnd Unit:=wdParagraph, Count:=1
    Debug.Print "The range now contains " & .Characters.Count & " characters."
  End With
End Sub
```

You can also redefine a Range object by using the object's Find property to return a Find object. The following example illustrates the use of the Find property to locate text within the active document. If the text is found, the Range object is automatically redefined to contain the text that matched the search criteria.

```
With rngRangeText.Find
  .ClearFormatting
  If .Execute(FindText:=strTextToFind) Then
    Set RedefineRangeExample2 = rngRangeText
  Else
    Set RedefineRangeExample2 = Nothing
  End If
End With
```

Many Word objects have a Range property that returns a Range object. You use an object's Range property to return a Range object under circumstances where you must work with properties or methods of the Range object that are not available from the object itself. For example, the following code uses the Range property of a Paragraph object to return a Range object that is used to format the text in the first paragraph in a document:

```
Dim rngPara As Range

Set rngPara = ActiveDocument.Paragraphs(1).Range
With rngPara
  .Bold = True
```

```
    .ParagraphFormat.Alignment = wdAlignParagraphCenter
    .Font.Name = "Arial"
End With
```

After you identify the Range object, you can apply methods and properties of the object to modify the contents of the range or get information about the range. You use the Range object's StoryType property to determine where in the document the Range is located.

Working with Text in a Range Object

You use a Range object's Text property to specify or determine the text the range contains. For example, the following code first displays the text within a Range object, then changes it and displays the new text, and finally restores the original text:

```
Public Sub ChangeTextSample()
    ' This procedure illustrates how to use the Range object's Text  property to copy and paste text into a document
   ‘ while  maintaining the original paragraphs.
   ' When the rngText variable is instantiated, it includes all of the text in the first paragraph in the active document plus
   ‘ the paragraph mark at the end of the paragraph. Note how the new  text in the strNewText variable includes a paragraph
   ‘mark  (vbCrLf) to replace the mark removed when the orginal text was replaced.

    Dim rngText             As Range
    Dim strOriginalText     As String
    Dim strNewText          As String

  strNewText = "Now is the time to harness the power of VBA in Word." & "This text is replacing the original text in the first " _
      & "paragraph. This is all done using only the Text property " & "of the Range object!" & vbCrLf

    Set rngText = ActiveDocument.Paragraphs(1).Range
    With rngText
      MsgBox .Text, vbOKOnly, "This is the original text."
      strOriginalText = .Text
      .Text = strNewText
      MsgBox .Text, vbOKOnly, "This is the new text inserted in paragraph 1."
      .Text = strOriginalText
      MsgBox "The original text is restored."
    End With
End Sub
```

In this example, the Range object's Text property is used to specify the text that appears in the document.

Determining Where the Range Is Located

You can use the Range object's StoryType property to determine where the range is located. Stories are distinct areas of a document that contain text. You can have up to 11 story type areas in a document, representing areas such as document text, headers, footers, footnotes, comments, and more. You use the StoryRanges property to return a StoryRanges collection. The StoryRanges collection contains Range objects representing each story in a document.

A new Word document contains a single story, called the Main Text story, which represents the text in the main part of the document. Even a blank document contains a character, a word, a sentence, and a paragraph.

You do not expressly add new stories to a document, but rather, Word adds them for you when you add text to a portion of the document represented by one of the 11 story types. For example, if you add footnotes, Word adds a Footnotes story. If you add comments, Word adds a Comments story to the document.

You use the Range property to return a Range object representing each story in a document. For example, the following code prints the text associated with the Main Text story and the Comments story:

```
Dim rngMainText         As Word.Range
Dim rngCommentsText     As Word.Range

Set rngMainText = ActiveDocument.StoryRanges(wdMainTextStory)
Set rngComments = ActiveDocument.StoryRanges(wdCommentsStory)
Debug.Print rngMainText.Text
Debug.Print rngComments.Text
```

Inserting Text in a Range

You use the Range object's InsertBefore or InsertAfter methods to add text to an existing Range object. In fact, there is an entire class of methods, with names that begin with "Insert," that you can use to manipulate a Range object.

It's useful to have a procedure that combines the Range object's InsertBefore and InsertAfter methods with the Text property. Having such a procedure creates a single place to handle much of the work you will do when manipulating text programmatically.

You can call the InsertTextInRange procedure when you must add text to a Range object. In other words, the procedure is useful when you want to programmatically make any changes to existing text in a Word document.

The InsertTextInRange procedure uses one required arguments and two optional argument. The strNewText argument contains the text you want to add to the Range object specified in the rngRange argument. The intInsertMode optional argument specifies how the new text will be added to the range. The values for this argument are one of three custom enumerated constants that specify whether to use the InsertBefore method, the InsertAfter method, or the Text property to replace the existing range text.

```
Public Function InsertTextInRange(strNewText As String, Optional rngRange As Word.Range, Optional intInsertMode _
        As opgTextInsertMode =  Replace) As Boolean
  ' This procedure inserts text specified by the strNewText  argument into the Range object specified by the rngRange
  ' argument. It calls the IsLastCharParagraph procedure to' strip off trailing paragraph marks from the rngRange object.
Call IsLastCharParagraph(rngRange, True)
  With rngRange
    Select Case intInsertMode
      Case 0 ' Insert before text in range.
        .InsertBefore strNewText
      Case 1 ' Insert after text in range.
        .InsertAfter strNewText
      Case 2 ' Replace text in range.
        .Text = strNewText
      Case Else
    End Select
    InsertTextInRange = True
  End With
End Function
```

Note

The IsLastCharParagraph procedure is used to strip off any final paragraph marks before inserting text in the range. The IsLastCharParagraph procedure is discussed earlier.

Understanding Paragraph Marks

When you create a Range object that represents a Character, Word, or Sentence object, and that object falls at the end of a paragraph, the paragraph mark is automatically included within the range. The Range object also includes all additional subsequent empty paragraph marks. For example, in a document where the first paragraph consists of three sentences, the following code creates a Range object that represents the last sentence in the first paragraph:

```
Set rngCurrentSentence = ActiveDocument.Sentences(3)
```

Because the rngCurrentSentence Range object refers to the last sentence in the first paragraph, that paragraph mark (and any additional empty paragraph marks) will be included in the range. If you then set the Text property of this object to a text string that didn't end with a paragraph mark, the first and second paragraphs in the document would be deleted.

When you write VBA code that manipulates text in a Word document, you must account for the presence of a paragraph mark in your text. There are two basic techniques you can use to account for paragraph marks when you are cutting and pasting text in Range objects:

- Include a new paragraph mark (represented by the vbCrLf constant) in the text to be inserted in the document. This technique is illustrated in the ChangeTextSample procedure shown in "Working with Text in a Range Object" .
- Exclude the final paragraph mark from a Range object. The following code sample shows how to change the contents of a Range object to exclude the final paragraph mark. The example uses the Chr$() function with character code 13 to represent a paragraph mark.

```
Function IsLastCharParagraph(ByRef rngTextRange As Word.Range, Optional blnTrimParaMark As Boolean = _
        False) As Boolean
  ' This procedure accepts a character, word, sentence, or paragraph Range object as the first argument and returns True
  ' if the last character in the range is a paragraph mark, and False if it is not. The procedure also accepts an optional
  ' Boolean argument that specifies whether the Range object should be changed to eliminate the paragraph mark if it
  ' exists. When the blnTrimParaMark argument is True, this procedure calls itself recursively to strip off all trailing
  ' paragraph marks.
  Dim strLastChar As String
  strLastChar = Right$(rngTextRange.Text, 1)
  If InStr(strLastChar, Chr$(13)) = 0 Then
    IsLastCharParagraph = False
    Exit Function
  Else
    IsLastCharParagraph = True
    If Not blnTrimParaMark = True Then
        Exit Function
    Else
      Do
        rngTextRange.SetRange rngTextRange.Start, rngTextRange.Start + rngTextRange.Characters.Count - 1
```

```
        Call IsLastCharParagraph(rngTextRange, True)
      Loop While InStr(rngTextRange.Text, Chr$(13)) <> 0
    End If
  End If
End Function
```

In this example, the Count property of the Range object's Characters collection is used to redefine the Range object's end point.

The Selection Object

When you use the Word user interface to work with a document, you typically select (highlight) text and then do something to the text, such as formatting it, typing new text, or moving it to another location. The Selection object represents the currently selected text in a Word document. The Selection object is always present in a document; if no text is selected, it represents the insertion point. Unlike the Range object, there can only be one Selection object at a time. You can use the Selection object's Type property to get information about the state of the current selection. For example, if there is no current selection, the Selection object's Type property returns wdSelectionIP. The Type property will return one of nine different values represented by the wdSelectionType enumerated constants.

You access a Selection object by using the Selection property. This property is available from the Application, Window, and Pane objects. However, because the Selection property is global, you can refer to it without referencing another object first. For example, the following sample code illustrates how you use the Selection property to get information about the currently selected text:

```
Sub SelectionCurrentInfo()
  Dim strMessage As String
  With Selection
    If .Characters.Count > 1 Then
   strMessage = "The Selection object in '" & ActiveDocument.Name  & "' contains " & .Characters.Count & " characters, " _
        & .Words.Count & " words, " & .Sentences.Count & " sentences, and " & .Paragraphs.Count & " paragraphs."
      MsgBox strMessage
    End If
  End With
End Sub
```

The Selection Object vs. the Range Object

In many ways, the Selection object is similar to a Range object. The Selection object represents an arbitrary portion of a document. It has properties that represent characters, words, sentences, paragraphs, and other objects in a Word document. The main difference is that when you use the Range object, it's not necessary to first select the text. In addition, there can only be one Selection object at a time, but the number of Range objects you can create is unlimited.

The Selection object and the Range object have many common methods and properties, and it is easy to return a Range object from a Selection object or to create a Selection object from a Range object. However, most things you can do with a Selection object, you can do even faster with a Range object. There are two main reasons for this:

- The Range object typically requires fewer lines of code to accomplish a task.
- Manipulating a Range object does not incur the overhead associated with Word having to move or change the selection "highlight" in the active document.

In addition, you can do much more with a Range object than you can with a Selection object:

- You can manipulate a Range object without changing what the user has selected in the document. Practically speaking, you could save the original selection by using a Range object variable, manipulate the Selection object programmatically, and then use the saved Range object's Select method to display the original selection, but there is rarely a good reason to show the user that the selection is changing. Some WordBasic developers relied on changing the selection to indicate to the user that the code is still running (and the machine has not locked up). But this is not the right way to convey information to a user. An operation that takes a long time to execute should signal its progress by using a progress meter or by posting status messages to the status bar.
- You can maintain multiple Range objects in your code, and, where necessary, store those objects in a custom Collection object. You cannot do these two things by using only the Selection object.

When it comes to manipulating text, the Selection and Range objects have many methods and properties in common-for example, all the InsertName methods and the Text property discussed in "Working with Text in a Range Object"193 . However, the Selection object has a unique set of methods for manipulating text. These are the TypeText, TypeParagraph, and TypeBackspace methods. You use these methods to enter or remove text and insert paragraph marks in a Selection object. To get the results you expect, there are a few things you must understand about the TypeName methods.

With the exception of the InsertParagraph and InsertFile methods, which remove selected text, the InsertName methods let you work with a selection without deleting existing text. In contrast, the TypeName methods might delete existing text, depending on the value of the Options object's ReplaceSelection property.

When the ReplaceSelection property is True, using any of the TypeName methods results in the currently selected text being replaced. When the ReplaceSelection property is False, the TypeText and TypeParagraph methods behave just as the

InsertBefore method: The text or paragraph mark is inserted at the beginning of the current selection. When the ReplaceSelection property is False, the TypeBackspace method behaves the same as the Collapse method of a Range or Selection object when the wdCollapseStart constant is specified in the Direction argument. The Collapse method collapses a range or selection so that its starting point and ending point are the same.

Working with Bookmarks

In many ways, a Bookmark object is similar to a Selection or Range object in that it represents a contiguous area in a document. It has a starting position and an ending position, and it can be as small as the insertion point or as large as the entire document. However, a Bookmark object differs from a Selection or Range object because you can give the Bookmark object a name and it does not go away when your code stops running or when the document is closed. In addition, although bookmarks are normally hidden, you can make them visible by setting the View object's ShowBookmarks property to True.

You use bookmarks to mark a location in a document or as a container for text in a document. The following examples illustrate these uses:

- You could use bookmarks to mark areas in a document that will contain data supplied by the user or obtained from an outside data source. For example, a business letter template might have bookmarks marking the locations for name and address information. Your VBA code could obtain the data from the user or from a database and then insert it in the correct locations marked by bookmarks. When a location is marked, navigating to that location is as simple as navigating to the bookmark. You can determine if a document contains a specific bookmark by using the Bookmarks collection's Exists method. You display a location marked by a bookmark by using the Bookmark object's Select method. When a bookmark is selected, the Selection object and the Bookmark object represent the same location in the document.
- If you have a document that contains boilerplate text that you must modify in certain circumstances, you could use VBA code to insert different text in these specified locations depending on whether certain conditions were met. You can use a Bookmark object's Range property to create a Range object, and then use the Range object's InsertBefore method, InsertAfter method, or Text property to add or modify the text within a bookmark.

When you understand the subtleties associated with adding or changing text through VBA code, working with bookmarks can be a powerful way to enhance your custom applications created in Word.

You add a bookmark by using the Bookmarks collection's Add method. You specify where you want the bookmark to be located by specifying a Range or Selection object in the Add method's Range argument. When you use the InsertBefore method, the InsertAfter method, or the Text property, a Range object automatically expands to incorporate the new text. As you will see in the next few examples, a bookmark does not adjust itself as easily, but making a bookmark as dynamic as a range is a simple exercise.

When you use the Range object's InsertBefore method to add text to a bookmark, the text is added to the start of the bookmark and the bookmark expands to include the new text. For example, if you had a bookmark named CustomerAddress on the following text (the brackets appear when the ShowBookmarks property is set to True)

[Seattle, WA 12345]

you could add the street address to this bookmark by using the following VBA code:

```
Dim rngRange As Word.Range
Set rngRange = ActiveDocument.Bookmarks("CustomerAddress").Range
rngRange.InsertBefore "1234 Elm Drive #233" & vbCrLf
```

As you might expect, the bookmark expands to include the additional address information:

[1234 Elm Drive #233
Seattle, WA 12345]

Now suppose you want to use the InsertAfter method to add text to the end of a bookmark that contains the street address, and you want to add the city, state, and zip code information by using this code:

```
Dim rngRange As Word.Range
Set rngRange = ActiveDocument.Bookmarks("CustomerAddress").Range
rngRange.InsertAfter vbCrLf & "Seattle, WA 12345"
```

Note that when you use the InsertAfter method to add text to the end of a bookmark, the bookmark does not automatically expand to include the new text:

[1234 Elm Drive #233]
Seattle, WA 12345

This behavior could create problems if you were unaware of it. But now you are aware of it, and the solution is quite easy. The first part of the solution results from the benefits achieved when you use the Selection and Range objects together. The second part results from another aspect of bookmarks that you must know: When you add a bookmark to a document in which the bookmark already exists, the original bookmark is deleted (but not the text it contained) when the new bookmark is created.

The following sample code uses the InsertAfter method to add text to the end of the CustomerAddress bookmark. It then uses the Range object's Select method to create a Selection object covering all the text you want to bookmark. Finally, it uses the Bookmarks collection's Add method to add a new bookmark that has the same name as the original bookmark and then uses the Selection object's Range property to specify the location of the bookmark:

```
Dim rngRange As Word.Range

Set rngRange = ActiveDocument.Bookmarks("CustomerAddress").Range
With rngRange
  .InsertAfter vbCrLf & "Seattle, WA 12345"
  .Select
End With
ActiveDocument.Bookmarks.Add "CustomerAddress", Selection.Range
```

If you use the Range object's Text property to replace the entire contents of a bookmark, you run into a similar problem: The text in the bookmark is replaced, but in the process, the bookmark itself is deleted. The solution to this problem is the same solution we used for the InsertAfter method in the preceding example. You insert the new text, use the Range object's Select method to select the text, and then create a new bookmark that has the same name as the original bookmark.

The Find and Replacement Objects

Among the most frequently used commands in the Word user interface are the Find and Replace commands on the Edit menu. These commands let you specify the criteria for what you want to locate. They are both really the same thing, with the Replace command's functionality being just an extension of the Find command's functionality. In fact, you might have noticed that Find, Replace, and Go To appear on different tabs of the same dialog box-the Find and Replace dialog box.

Much of the VBA code you write in Word involves finding or replacing something in a document. There are several techniques you can use to locate text or other elements in a document, for example, using the GoTo method or the Select method. Typically, you use the Find object to loop through a document looking for some specific text, formatting, or style. To specify what you want to use to replace the item you found, you use the Replacement object, which you can access by using the Replacement property of the Find object.

The Find object is available from both the Selection object and the Range object; however, it behaves differently depending on whether it is used from the Selection object or the Range object. Searching for text by using the Find object is one of those situations where the Selection and Range objects can be used together to accomplish more than either object can its own.

The following list describes differences between the behavior of the Range object and the Selection object when you are searching for an item in a document:

- When you are using the Selection object, your search criteria are applied only against the currently selected text.
- When you are using the Selection object, if an item matching the search criteria is found, the selection changes to highlight the found item, as illustrated by the following example, which uses the Find object to search within the currently selected text:

```
With Selection.Find
  .ClearFormatting
  strFindText = InputBox("Enter the text you want to find.", "Find Text")
  If Len(strFindText) = 0 Then Exit Sub
  .Text = strFindText
  If .Execute = True Then
    MsgBox """ & Selection.Text & """ & " was found and is now highlighted."
  Else
    MsgBox "The text could not be located."
  End If
End With
```

- When you are using the Find object off of the Range object, the definition of the Range object changes when an item matching the search criteria is found. Failing to account for this change in the definition of the Range object can cause all kinds of debugging headaches. The following code sample illustrates how the Range object is redefined:

```
Dim rngText      As Word.Range
Dim strToFind    As String

Set rngText = ActiveDocument.Paragraphs(3).Range
With rngText.Find
  .ClearFormatting
  strToFind = InputBox("Enter the text you want to find.",  "Find Text")
  If Len(strToFind) = 0 Then Exit Sub
  .Text = strToFind
  If .Execute = True Then
    MsgBox """ & strToFind & """ & " was found. As a result, the Range object has been redefined and now covers the text: "
& rngText.Text
  Else
    MsgBox "The text could not be located."
  End If
End With
```

Regardless of whether you are using the Find object with the Range object or the Selection object, you must account for the changes that occur to the object when the search is successful. Because the object itself might point to different text each time the search is successful, you might have to account for this and you might also have to keep track of your original object so that you can return to it when the search has been completed.

Specifying and Clearing Search Criteria

You specify the criteria for a search by setting properties of the Find object. There are two ways to set these properties. You can set individual properties of the Find object and then use the Execute method without arguments. You can also set the properties of the Find object by using the arguments of the Execute method. The following two examples execute identical searches:

Example 1: Using properties to specify search criteria.

```
With Selection.Find
  .ClearFormatting
  .Forward = True
  .Wrap = wdFindContinue
  .Text = strToFind
  .Execute
End With
```

Example 2: Using Execute method arguments to specify search criteria.

```
With Selection.Find
  .ClearFormatting
  .Execute FindText:=strToFind, Forward:=True, Wrap:=wdFindContinue
End With
```

The Find object's search criteria are cumulative, which means that unless you clear out the criteria from a previous search, new criteria are added to the criteria used in the previous search. You should get in the habit of always using the ClearFormatting method to remove formatting from the criteria from a previous search before specifying the criteria for a new search. The Find object and the Replacement object each has its own ClearFormatting method. When you are performing a find and replace operation, you must use the ClearFormatting method of both objects, as illustrated in the following example:

```
With Selection.Find
  .ClearFormatting
  .Text = strToFind
  With .Replacement
    .ClearFormatting
    .Text = strReplaceWith
  End With
  .Execute Replace:=wdReplaceAll
End With
```

Finding All Instances of the Search Criteria

When you use the Execute method as shown in the preceding examples, the search stops at the first item that matches the specified criteria. To locate all items that match the specified criteria, use the Execute method inside a loop, as shown in the following example:

```
Public Sub SearchAndReturnExample()
  ' This procedure shows how to use the Execute method inside  a loop to locate multiple instances of specified text.
  Dim rngOriginalSelection    As Word.Range
  Dim colFoundItems           As New Collection
  Dim rngCurrent              As Word.Range
  Dim strSearchFor            As String
  Dim intFindCounter          As Integer

  If (Selection.Words.Count > 1) = True Or _
      (Selection.Type = wdSelectionIP) = True Then
    MsgBox "Please select a single word or part or a word. " & "This procedure will search the active document for " _
      & "additional instances of the selected text."
      Exit Sub
  End If
  Set rngOriginalSelection = Selection.Range
  strSearchFor = Selection.Text
' Call custom procedure that moves the insertion point to the start of the document.
  Call GoToStartOfDoc
With Selection.Find
    .ClearFormatting
    .Forward = True
    .Wrap = wdFindContinue
    .Text = strSearchFor
    .Execute
```

```
    Do While .Found = True
      intFindCounter = intFindCounter + 1
      colFoundItems.Add Selection.Range, CStr(intFindCounter)
      .Execute
    Loop
End With
  rngOriginalSelection.Select
If MsgBox("There are " & intFindCounter & " instances of '" & rngOriginalSelection & "' in this document." & vbCrLf & _
          vbCrLf  & "Would you like to loop through and display all instances?", vbYesNo) = vbYes Then
    intFindCounter = 1
    For Each rngCurrent In colFoundItems
      rngCurrent.Select
      MsgBox "This is instance #" & intFindCounter
      intFindCounter = intFindCounter + 1
    Next rngCurrent
End If
  rngOriginalSelection.Select
End Sub
```

The preceding example also illustrates how to use a Collection object to store the matching items as Range objects. In this example, the user is given the option of viewing all matching items, but you could use the same technique to work with the found items as a group.

Replacing Text or Other Items

To replace one item with another, you must specify a setting for the Replace argument of the Execute method. You can specify the replacement item by using either the Text property of the Replacement object or the ReplaceWith argument of the Execute method. To delete an item by using this technique, use a zero-length string ("") as the replacement item. The following example replaces all instances of the text specified by the strFind argument with the text specified in the strReplace argument:

```
Sub ReplaceText(strFind As String, strReplace As String)
  Application.ScreenUpdating = False
  ActiveDocument.Content.Select
  With Selection.Find
    .ClearFormatting
    .Forward = True
    .Wrap = wdFindContinue
    .Execute FindText:=strFind, Replace:=wdReplaceAll, ReplaceWith:=strReplace
  End With
End Sub
```

Restoring the User's Selection After a Search

In most cases, when you finish a search operation, you should return the selection (or the insertion point if there was no previous selection) to where it was when the search began. You do this by saving the state of the Selection object before you begin a search, and then restoring it when the search is completed, as shown in the following example:

```
Sub SimpleRestoreSelectionExample()
  Dim rngStartMarker      As Word.Range
  Dim strToFind           As String

  Set rngStartMarker = Selection.Range
  strToFind = InputBox("Enter the text to find.", "Find Text")
  With Selection.Find
    .ClearFormatting
    .Text = strToFind
    If .Execute = True Then
      MsgBox """ & strToFind & """ & " was found and is currently highlighted. Click OK to restore your original selection."
    Else
      MsgBox """ & strToFind & """ & " was not found."
    End If
  End With
  rngStartMarker.Select
End Sub
```

3. WORKING WITH SHARED OFFICE COMPONENTS

Microsoft® Office includes a set of shared objects available in all Office applications that help you search for files, use the Office Assistant, manipulate command bars, read and write document properties, read and write script, and hook add-ins to your Office application. Because these objects are shared among all Office applications, it is easy to write code that uses these objects and that will run without modification from within any Office application or custom Office application.

You can use these objects to customize the appearance of your application, create custom toolbars and menu bars in code, perform custom file searches, or customize the Office Assistant to respond to the user's actions.

In This Section

Referencing Shared Office Components
Return a reference to a shared component object by using the appropriate properties.

Working with the FileSearch Object
Programmatically access the functionality of the Office File Open dialog box.

Working with the Office Assistant
Use the objects, methods, and properties of the Office Assistant object to programmatically control the Office Assistant.

Working with Command Bars
Write code to manipulate command bars that can be used in any Microsoft® Office application or custom application you develop.

Working with Document Properties
Use document properties to create, maintain, and track information about a Microsoft® Office document.

Working with Scripts
Access script, or insert script into a cell or range in a Microsoft® Excel worksheet, a Microsoft® PowerPoint® slide, a Microsoft® Word document, or Word Selection object.

Referencing Shared Office Components

Every Microsoft® Office application includes accessor properties that provide access to the shared Office components. For example, an Office application's Assistant property returns a reference to the Assistant object, the FileSearch property returns a reference to the FileSearch object, and the Scripts property returns a reference to the Scripts collection. From within any Office application, you can return a reference to a shared component object by using the appropriate accessor property; you do not have to use the New keyword to create an object variable that references the shared Office component.

Note
All Office applications, except Microsoft® Access, Microsoft®FrontPage®, and Microsoft® Outlook®, include a reference to the Microsoft Office XP object library by default. Before you can work with shared Office components in Access, FrontPage, or Outlook, you must first manually set a reference to the Microsoft Office XP object library.

As with any object model, before you can work with an object, you must either set an object variable to the object you want to work with or use the host application's accessor property. For example, the following code fragments illustrate using the accessor property (in these cases, the FileSearch, Assistant, and CommandBars accessor properties are used) to access various shared Office components.

```
With Application.FileSearch
  .NewSearch
  .LookIn = "C:\My Documents"
  .FileName = "*.doc"
  If .Execute() > 0 Then
    ' Work with found files here.
  End If
End With
```

```
Dim objAssistant As Assistant
Set objAssistant = Application.Assistant
With objAssistant
  .On = True
  .Visible = True
  .Animation = msoAnimationCharacterSuccessMajor
End With
```

```
Dim cbrCustomBar As CommandBar
Set cbrCustomBar = Application.CommandBars(strCBName)
With cbrCustomBar.Controls(strCtlName)
  .Enabled = Not .Enabled
End With
```

Note
To set a reference to a shared Office component from outside an Office application, you must still use the accessor property of an Office application. For example, to set a reference to the FileSearch object from a Microsoft® Visual Basic® application, you could set a reference to the Microsoft® Word Application object and then use the Word FileSearch property to return a reference to the FileSearch object. For example:

```
Dim wdApp As Word.Application
Set wdApp = New Word.Application
With wdApp.FileSearch
…..
End With
```

Working with the FileSearch Object

The FileSearch object exposes a programmatic interface to all the functionality of the Office File Open dialog box, including the features found in the Advanced Find dialog box, which is available from the Open dialog box. You can use the objects, methods, and properties of the FileSearch object to search for files or collections of files based on criteria you supply.

Note
If Microsoft Fast Find is enabled, the FileSearch object can use Fast Find indexes to speed up its searching capabilities.

In This Section

The Basics of File Searching
Understand the methods and properties of file searching.

Using Advanced File-Searching Features
Learn about some more complex file-searching features.

Creating Reusable File-Search Code
Learn why the FileSearch object is a great candidate for encapsulation in a class module.

The Basics of File Searching

The following code fragment illustrates how to use an application's FileSearch property to return a reference to the FileSearch object. Because the FileSearch object is shared among all Microsoft® Office applications, this code will work without modification from within any Office application:

```
Function FindFile(strFileSpec As String)
  Dim fsoFileSearch As FileSearch
  Set fsoFileSearch = Application.FileSearch
  With fsoFileSearch
    .NewSearch
    .LookIn = "c:\"
    .FileName = strFileSpec
    .SearchSubFolders = False
    If .Execute() > 0 Then
      For Each varFile In .FoundFiles
        strFileList = strFileList & varFile & vbCrLf
      Next varFile
    End If
  End With
  MsgBox strFileList
End Function
```

The FileSearch object has two methods and several properties you can use to build custom file-searching functionality into your custom Office applications. The previous example uses the NewSearch method to clear any previous search criteria and the Execute method to carry out the search for the specified files. The Execute method returns the number of files found, and also supports optional parameters that make it possible for you specify the sort order, the sort type, and whether to use only saved Fast Find indexes to perform the search. You use the FoundFiles property to return a reference to the FoundFiles object that contains the names of all matching files found in your search.

Note
You must use the NewSearch method to clear any search criteria from previous searches; otherwise, the new search criteria will be added to the existing search criteria.

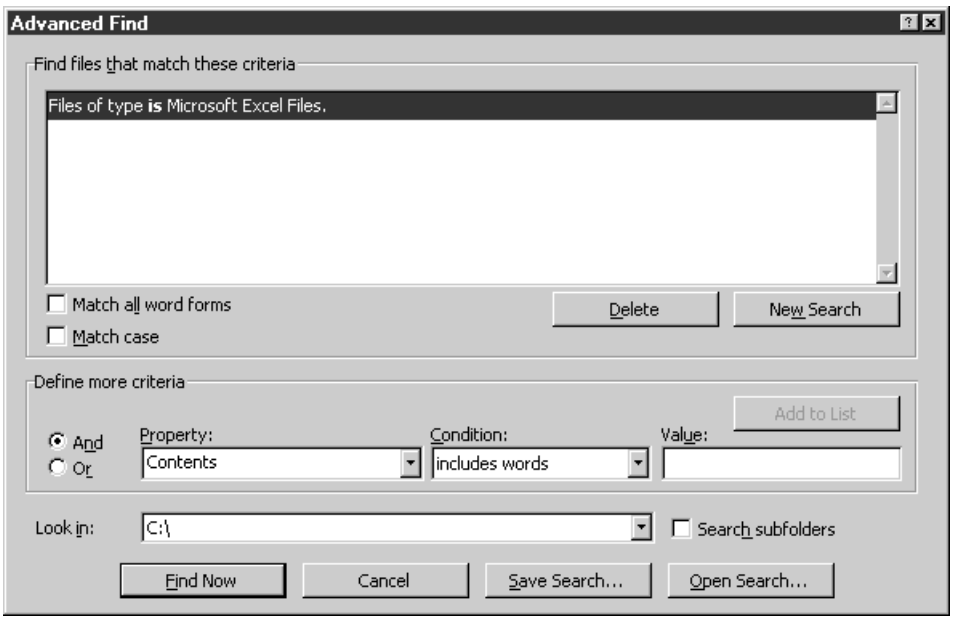
You use the LookIn property to specify what directory to begin searching in and the SearchSubFolders property to specify whether the search should extend to subfolders of the directory specified in the LookIn property. The FileName property supports wildcard characters and a semicolon-delimited list of file names or file-type specifications.

For more information about using the methods and properties of the FileSearch object, search the Microsoft Office Visual Basic Reference Help index for "FileSearch object."

Using Advanced File-Searching Features

You get programmatic access to the advanced features of the FileSearch object by using its PropertyTests collection. These features correspond to the options available in the Advanced Find dialog box, which is available through the Office File Open dialog box.

The Advanced Find Dialog Box



The PropertyTests collection contains the criteria for a file search. Some of these criteria might have been specified by properties of the FileSearch object itself while others must be added to the PropertyTests collection by using its Add method.

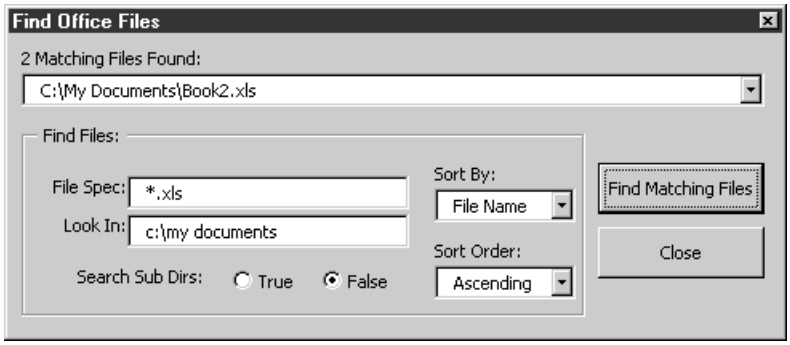
In the following example, one of the file-search criteria added to the PropertyTests collection corresponds to the Contents setting in the Property box and another corresponds to the includes words setting in the Condition box in the preceding figure.

```
Set fsoFileSearch = Application.FileSearch
With fsoFileSearch
  .NewSearch
  .FileName = strFileName
  .LookIn = strLookIn
  .SearchSubFolders = blnSearchSubDir
  .PropertyTests.Add "Contents", msoConditionIncludes, strFindThisText
  If .Execute(msoSortByFileName, msoSortOrderAscending, True) > 0 Then
    For Each varFile In .FoundFiles
      cboFoundCombo.AddItem varFile
    Next varFile
  Else
    cboFoundCombo.AddItem "No Matching Files Located!"
  End If
  cboFoundCombo.ListIndex = 0
End With
```

Creating Reusable File-Search Code

Searching for files is something you might do repeatedly in any number of different Office applications. This makes the FileSearch object a great candidate for encapsulation in a class module that could be used in any Microsoft®Office application that requires file-searching capabilities.

Dialog Box Used to Gather Custom Search Criteria



The Find Office Files dialog box is shown immediately after executing a search for files with an ".xls" extension in the "c:\my documents" directory. Code behind the Find Matching Files command button uses a global variable named objFileInfo to call the GetFileList method of the custom clsGetFileInfo class as follows:

```
Sub UpdateFileList()
' If the file-search specifications are valid, update  the files contained in the form's combo box with a current list of matching files.
  Dim varFoundFiles   As Variant
  Dim varFile         As Variant

  varFoundFiles = objFileInfo.GetFileList
  If IsArray(varFoundFiles) Then
    With Me
      .cboMatchingFiles.Clear
      For Each varFile In varFoundFiles
        .cboMatchingFiles.AddItem varFile
      Next varFile
      .cboMatchingFiles.ListIndex = 0
      .lblFilesFound.Caption = CStr(objFileInfo.MatchingFilesFound)  & " Matching Files Found:"
    End With
  Else
    MsgBox "No files matched the specification: '" & Me.txtFileSpec & "'"
  End If
End Sub
```

The class contains several properties used to set the properties of the FileSearch object. It also exposes the GetFileList method that returns an array containing all files that match the specified criteria.

```
Public Function GetFileList() As Variant
  ' This function returns an array of files that match the criteria specified by the SearchPath and SearchName properties. If the
  ' SearchSubDirs property is set to True, the search includes subdirectories of SearchPath.
  Dim intFoundFiles    As Integer
  Dim astrFiles()      As String
  Dim fsoFileSearch    As FileSearch

  Set fsoFileSearch = Application.FileSearch
  With fsoFileSearch
    .NewSearch
    .LookIn = p_strPath
    .FileName = p_strName
    .FileType = msoFileTypeAllFiles
    .SearchSubFolders = p_blnSearchSubs
    If .Execute(p_intSortBy, p_intSortOrder) > 0 Then
      p_intFoundFiles = .FoundFiles.Count
      ReDim astrFiles(1 To .FoundFiles.Count)
        For intFoundFiles = 1 To .FoundFiles.Count
          astrFiles(intFoundFiles) = .FoundFiles(intFoundFiles)
        Next intFoundFiles
        GetFileList = astrFiles
    Else
      GetFileList = ""
    End If
  End With
End Function
```

Working with the Office Assistant

You can use the Office Assistant to animate characters that interact with your users, provide context-sensitive help, highlight parts of your user interface, collect information from users, or otherwise provide a "social" interface to your application that many users find interesting and fun to use. The Office Assistant character is drawn onscreen without an enclosing window that can interact with other elements of the application interface, pointing out controls or directing the user's attention to specific sections of a document.

You use the objects, methods, and properties of the Assistant object to programmatically control the Office Assistant, the Office Assistant balloon, and all the items inside the balloon.

In This Section

Microsoft Agent ActiveX Control vs. the Office Assistant
Learn when to use features of the Agent that are available only through the control.

Programming the Office Assistant
Make the Assistant visible, move it to different locations on the screen, specify the animation you want to run, and display Assistant balloons containing text and controls.

Working with Balloon Objects
Use the Office Assistant Balloon object to enables the Assistant to communicate with and get feedback from users.

Using Balloon Controls
Understand how to use Balloon controls.

Modeless Balloons and the Callback Property
Create a modeless balloon, and set its Callback property.

Microsoft Agent ActiveX Control vs. the Office Assistant

The Office Assistant is based on the Microsoft Agent ActiveX control. Many of the Agent control's methods and properties are exposed through the Assistant's object model. You can use the Agent control in Microsoft® Office applications, on Web pages, or in any environment that supports Microsoft® ActiveX® controls.

There are some circumstances where you would use the Agent control instead of the Assistant object to provide Office Assistant services:

You want to use features of the Agent that are available only through the control-for example, the Agent's speech-recognition capabilities.

You want to use the Agent control in an Office application where the Assistant object is not available. For example, if you have an Access run-time application on a machine that does not have Office installed, you can use the Agent control to provide the full range of Assistant services without accessing the Assistant's object model.

You want to use the Agent to provide Assistant-like services on a Web page. The Agent control is added to HTML pages by using the <OBJECT> tag and is manipulated by using script.

Programming the Office Assistant

Programming the Assistant is a matter of setting an object variable to the Assistant object and then accessing the properties and methods you must have to make the assistant do what you want. You can make the Assistant visible, move it to different locations on the screen, specify the animation you want to run, and display Assistant balloons containing text and controls.

One important thing to remember when you programmatically manipulate the properties of the Assistant is that the user might have set various Assistant properties that you should preserve. Any time you manipulate the Assistant, you should save the properties that existed before you began and then restore those properties when you are finished. For example, if the user normally has the Assistant turned off and you programmatically turn it on to perform some task, you should make sure you turn it off when you are finished using it.

In previous versions of Microsoft® Office, the Assistant is either visible and available or not visible and unavailable, but the Assistant can never be completely turned off. In Office XP, the Assistant has an On property that affects whether the Assistant is available at all.

You use the Assistant's On and Visible properties to determine its initial state. When the On property is set to False, the Visible property is False and any attempt to programmatically manipulate the Assistant (except for a call to the Assistant's Help method) is ignored and no error is raised. When the On property is set to True, the Assistant will be either visible or hidden depending on the Visible property's setting.

Note
When the On property's value is changed from False to True, the Visible property is set to True.

The following example demonstrates how to save the initial settings for the On and Visible properties, how to make the Assistant visible, and a simple animation technique:

```
Sub SimpleAnimation()
  ' This procedure shows simple Assistant animation  techniques. It calls the Wait procedure between  animations to give the animation time to complete.
  Dim blnAssistantVisible      As Boolean
  Dim blnAssistantOn           As Boolean

  With Application.Assistant
    blnAssistantOn = .On
    blnAssistantVisible = .Visible
   If Not blnAssistantOn Then
       .On = True
    ElseIf Not blnAssistantVisible Then
      .Visible = True
    End If
    .Animation = msoAnimationCheckingSomething
    Call Wait(5000)
    .Animation = msoAnimationEmptyTrash
    Call Wait(7000)
    .Animation = msoAnimationCharacterSuccessMajor
    Call Wait(5000)
    If (Not blnAssistantOn) Or (Not blnAssistantVisible) Then
      .On = blnAssistantOn
      .Visible = blnAssistantVisible
    End If
  End With
End Sub
```

Note

The Wait procedure used in the previous example is a subroutine that uses a custom class object to wait the number of milliseconds specified in the procedure's argument. When you are stringing Assistant animations together, this procedure is required to give one animation time to finish before another animation begins.

The Assistant's FileName property specifies the animated character that is displayed when the Assistant is visible. Character files use an ".acs" extension, and several characters are supplied with Office. In addition, you can create your own character files by using the Microsoft Agent ActiveX control's character editor.

Characters you create should be stored in the host application's folder or in the C:\Windows\Application Data\Microsoft\Office\Actors subfolder; if multiple users work on the same machine and user profiles have been set up on the machine, store your characters in the host application's folder or in the C:\Windows\Profiles\UserName\Application Data\Microsoft\Office\Actors subfolder. For more information about setting up user profiles, search the Microsoft® Windows® Help index for "user profiles."

You specify which character is displayed by setting the Assistant's FileName property to the name of the .acs file for the character you want to use. For example, the following procedure changes the character to the name of the character specified in the strCharName argument:

```
Function ChangeCharacter(strCharName As String) As Integer
  ' This procedure changes the existing Assistant  character to the character specified in the strCharName argument. The procedure's return values are set by using constants defined in the Declarations section of this module.

  With Application.Assistant
    If UCase(.FileName) = UCase(strCharName) Then
      ChangeCharacter = ASST_CHAR_SAMECHAR
      Exit Function
    End If
    .FileName = strCharName
    ChangeCharacter = ASST_CHAR_CHANGED
  End With
End Function
```

Note

If you are using a character supplied by Office, you are not required to include the full path to the .acs file you want to use. When you set the FileName property, Visual Basic for Applications (VBA) will look for the file in the host application's folder and then in the C:\Windows\Application Data\Microsoft\Office\Actors subfolder and, if it exists, in the C:\Windows\Profiles\UserName\Application Data\Microsoft\Office\Actors subfolder. If your character files are located somewhere other than the three locations discussed here, you must set the FileName property by using the full path to the file. If the file cannot be found, a message is displayed. If the user clicks OK, the same file-search sequence is executed again. If the user clicks Cancel, the attempt to change the FileName property is ignored and no error occurs.

Working with Balloon Objects

The Assistant's Balloon object enables the Assistant to communicate with and get feedback from your users. Balloon objects are designed to make it possible for you create a simple interface for user interaction. They are not designed to replace complex dialog boxes.

Balloon objects can contain text that can be plain, underlined, or displayed in different colors. In addition, Balloon objects can contain labels or check boxes, certain icons, and bitmaps. Only one Balloon object can be visible at a time, but you can create multiple Balloon objects in code and use them when required.

You create a Balloon object by using the Assistant's NewBalloon property. When you have created the new Balloon object, you can set its properties and then display it by using the Balloon object's Show method. The following two simple procedures illustrate many of the features of Balloon objects discussed so far. The TestCreateSimpleBalloon procedure creates two formatted strings used to specify the balloon Heading and Text properties. The procedure then creates the balSimple Balloon object by calling the CreateSimpleBalloon procedure and passing in the heading and text strings. CreateSimpleBalloon sets several other "default" properties for this balloon and then returns the new Balloon object to the calling procedure where it is displayed by using the Show method.

```
Sub TestCreateSimpleBalloon()
  Dim balSimple         As Balloon
  Dim strMessage        As String
  Dim strHeading        As String
  Dim blnAssistVisible  As Boolean

  strHeading = "This is a simple balloon."
  strMessage = "When you have finished reading this message, click OK to proceed." _
    & vbCrLf & "{cf 249}This text is red." & vbCrLf & "{cf 252}This text is blue." _
    & vbCrLf & "{cf 0}This text has a {ul 1}word{ul 0} that is underlined." _
    & vbCrLf & "This text is plain."
  blnAssistVisible = Application.Assistant.Visible
  Set balSimple = CreateSimpleBalloon(strMessage, strHeading)
If Not blnAssistVisible Then
    Call ShowAssistant
  End If
  With balSimple
    .Show
  End With
  Application.Assistant.Visible = blnAssistVisible
End Sub

Function CreateSimpleBalloon(strText As String, strHeading As String) As Office.Balloon
  Dim balBalloon As Balloon

  With Application.Assistant
    Set balBalloon = .NewBalloon
    With balBalloon
      .BalloonType = msoBalloonTypeButtons
      .Button = msoButtonSetOK
      .Heading = strHeading
      .Icon = msoIconTip
      .Mode = msoModeModal
      .Text = strText
    End With
    Set CreateSimpleBalloon = balBalloon
  End With
End Function
```

Note that the strMessage variable contains a string that includes embedded brackets such as {cf 252}, {cf 0}, {ul 1}, and {ul 0}. You use the {cf value} brackets to specify the color of the text that follows the bracket. You use the {ul value} brackets to specify where text underlining begins and ends. For more information about specifying text color and underlining text in Balloon objects, search the Microsoft® Office Visual Basic Reference Help index for "Text property."

If you run the sample code, you will notice that the code stops executing while the Balloon object is displayed. This is because the balloon's Mode property specifies that the balloon is modal. In addition, you can display modeless Balloon objects.

Using Balloon Controls

You add labels or check box controls to a Balloon object by using the Balloon object's Labels or Checkboxes property, respectively. (Note that label controls in Balloon objects are similar to option button controls.) You specify the text associated

with a label or check box by using the control's Text property. You specify a single control by using an index number between 1 and 5, which represents the number of the label or check box control in the balloon. For example, the following sample shows one way to use label controls in a Balloon object:

```
With balBalloon
  .Button = msoButtonSetNone
  .Heading = "Balloon Object Example One"
  .Labels(1).Text = "VBA is a powerful programming language."
  .Labels(2).Text = "Office is a great development environment."
  .Labels(3).Text = "The Assistant is cool!"
  .Labels(4).Text = "Balloon objects are easy to use."
  .Text = "Select one of the following " & .Labels.Count & "Options:"
  ' Show the balloon.
  intRetVal = .Show

  ' Save the selection made by the user.
  If intRetVal > 0 Then
    strChoice = "{cf 4}" & .Labels(intRetVal).Text & "{cf 0}"
  Else
    strChoice = ""
  End If
End With
Set balBalloon = Assistant.NewBalloon
With balBalloon
  .Text = "You selected option " & CStr(intRetVal) & ": '" & strChoice & "'"
  .Show
End With
```

Note that when the balloon is displaying label controls, you are not required to have OK or Cancel buttons, because the balloon is dismissed as soon as any label control is selected, and the user can select only one control at a time. This is not the case when you use check box controls. The user can select more than one check box before dismissing the balloon, so your code should account for multiple selections. The next example shows one way to display check box controls and then identify the selections made by the user:

```
With balBalloon
  .Button = msoButtonSetOK
  .Heading = "Balloon Object Example Two"
  .Checkboxes(1).Text = "VBA is a powerful programming language."
  .Checkboxes(2).Text = "Office is a great development environment."
  .Checkboxes(3).Text = "The Assistant is cool!"
  .Checkboxes(4).Text = "Balloon objects are easy to use."
  .Text = "How many of the following "  & .Checkboxes.Count & " statements do you agree with?"
' Save the selection made by the user.
  intRetVal = .Show
' Construct the string to display to the user based on the  user's selections.
  For Each chkBox In .Checkboxes
    If chkBox.Checked = True Then
      strChoice = strChoice & "{cf 4}" & chkBox.Text & "{cf 0}" & "' and '"
    End If
  Next chkBox
  ' Remove the trailing "' and '" from strChoice.
  If Len(strChoice) <> 0 Then
    strChoice = Left(strChoice, Len(strChoice) - 7)
  End If
End With
' Create new Balloon object and display the user's choices.
Set balBalloon = Assistant.NewBalloon
With balBalloon
  If intRetVal > 0 Or Len(strChoice) > 0 Then
    .Text = "You selected '" & strChoice & "'."
  Else
    .Text = "You didn't make a selection."
  End If
  .Show
End With
```

Modeless Balloons and the Callback Property

When you display a modeless balloon, the user is able to use your application while the balloon is displayed. You specify that a balloon is modeless by setting the Mode property to the built-in constant msoModeModeless.

When you create a modeless balloon you must also set its Button property to something other than msoButtonSetNone and its Callback property to the name of a procedure to call when the user clicks a button in the modeless balloon. The procedure named in the Callback property must accept three arguments: a Balloon object, a long integer representing the button selected (msoBalloonButtonType values or a number representing the button clicked when the BalloonType property is set to msoBalloonTypeButtons), and a long integer representing the Balloon object's Private property.

You use the Private property to assign a value to a Balloon object that uniquely identifies it to the procedure named in the Callback property. You could use this property in a single generic callback procedure that is called from multiple modeless balloons. For example, in the following code, the sample contains a five-step tour of a Northwind Company spreadsheet that uses a collection of modeless balloons representing each step in the tour. All five Balloon objects name the BalloonCallBackProc procedure in their Callback property setting. Each Balloon object uses a unique value in its Private property setting, and the BalloonCallBackProc procedure uses this value and the value of the button clicked by the user (lngBtnRetVal) to identify which balloon has called the procedure and which button was clicked. The Balloon objects that call this procedure all specify a Private property by using a module-level constant (BALLOON_ONE, BALLOON_TWO, and so on) that indicates in which step of the tour they are called. Each balloon has a Close button and either a Next button, a Back button, or both, depending on the balloon's location in the tour. This single procedure is designed to handle all selections made in all balloons:

```
Function BalloonCallBackProc(balBalloon As Balloon, lngBtnRetVal As Long, lngPrivateBalloonID As Long)
  ' This procedure is specified in the Callback property  setting for all five balloons used in the Modeless
  ' Balloon Demo. These balloons are created in the AddBalloon procedure and stored in the mcolModelessBalloons collection.
  Const BUTTON_BACK As Long = -5
  Const BUTTON_NEXT As Long = -6

 ' Close current balloon.
  balBalloon.Close
  Select Case lngPrivateBalloonID + lngBtnRetVal
    Case BALLOON_ONE + BUTTON_NEXT
      ' User clicked first balloon, Next button.
      Call ShowModelessBalloon(CStr(BALLOON_TWO))
    Case BALLOON_TWO + BUTTON_NEXT
      Call ShowModelessBalloon(CStr(BALLOON_THREE))
    Case BALLOON_TWO + BUTTON_BACK
      Call ShowModelessBalloon(CStr(BALLOON_ONE))
    Case BALLOON_THREE + BUTTON_NEXT
      Call ShowModelessBalloon(CStr(BALLOON_FOUR))
    Case BALLOON_THREE + BUTTON_BACK
      Call ShowModelessBalloon(CStr(BALLOON_TWO))
    Case BALLOON_FOUR + BUTTON_NEXT
      Call ShowModelessBalloon(CStr(BALLOON_FIVE))
    Case BALLOON_FOUR + BUTTON_BACK
      Call ShowModelessBalloon(CStr(BALLOON_THREE))
    Case BALLOON_FIVE + BUTTON_BACK
      Call ShowModelessBalloon(CStr(BALLOON_FOUR))
    Case Else
      ' User clicked Close button.
      Set mcolModelessBalloons = Nothing
  End Select
End Function
```

This is just one example of the kinds of things you can do with modeless Balloon objects. You have a great deal of flexibility over what you can do with a balloon and a great deal of programmatic control over how your users interact with the balloons you create. You can use the Object Browser to get a complete listing of all the Balloon object's properties and methods. You can use the Microsoft® Office Visual Basic Reference Help index to get more information about these properties and methods.

Working with Command Bars

Microsoft®Office applications all share the same technology for creating menus and toolbars, and this technology is available to you through the command bars object model. In Office applications, there are three kinds of CommandBar objects: toolbars, menu bars, and pop-up menus. Pop-up menus are displayed in three ways: as menus that drop down from menu bars, as submenus that cascade off menu commands, and as shortcut menus. Shortcut menus (also called "right-click menus") are menus that appear when you right-click something.

Because the command bars object model is shared by all Office applications, you can write code to manipulate command bars that can be used in any Office application or custom application you develop. Everything you can do in a host application by

using the Customize dialog box you also can do by using Microsoft® Visual Basic® for Applications (VBA) code. In addition, there are some things you can do only by using VBA code.

Understanding how to work with command bars in Office applications requires that you understand not only what they have in common across all applications (the command bars object model) but also how they differ within each application.

In This Section

Understanding Application-Specific Command Bar Information
Learn how each Microsoft® Office application stores command bar information in a different location and, in some cases, implements command bars in a different way.

Manipulating Command Bars and Command Bar Controls with VBA Code
Use objects, collections, properties, and methods to show, hide, and modify existing command bars and command bar controls, as well as create new ones.

Understanding Application-Specific Command Bar Information

Despite sharing a common object model, each Microsoft® Office application stores command bar information in a different location and, in some cases, implements command bars in a different way. The primary difference is how and where each Office application stores custom command bars.

Note
When you make changes to any of the built-in command bars in an Office application, information about those changes is stored in the Windows registry on a per-user basis. Information about the visibility and location of built-in and custom command bars is stored in the registry as well.

Each Office application stores its command bars either with the Office document that contains the command bars, or in an application-specific file. One important result of this is that command bars cannot be shared between Office documents of different types although they can be shared among documents of the same type. You cannot create a command bar in Microsoft® Word and then copy that command bar to a Microsoft® Access application and use it there.

With the exception of Access, all Office applications store command bar information in specific locations, the path to which depends on whether user profiles have been set up for multiple users on the computer where the command bars are created. For more information about setting up user profiles, search the Microsoft® Windows® Help index for "user profiles."

Microsoft Access Command Bars

The command bars you create in Microsoft® Access are stored with the database in which they are created. If you want to create command bars that are available to more than one database, you must create them in an add-in database and reference that database from each database application where you want the command bars to be available.

Built-in command bars and information about them, for example their visibility and location, is stored in the Windows registry.

The location of information about command bars in an Access database is not dependant upon whether user profiles have been set up for multiple users.

Microsoft Excel Command Bars

Microsoft® Excel makes it possible for you store command bars with an individual workbook or in the Excel workspace. Workspace command bars are saved in a file named Excel.xlb. If user profiles have been set up for multiple users, Excel.xlb is stored in the C:\Windows\Profiles\UserName\Application Data\Microsoft\Excel subfolder. If user profiles have not been set up, Excel.xlb is stored in the C:\Windows\Application Data\Microsoft\Excel subfolder.

You can copy command bars from the workspace to a workbook by using the Attach Toolbars dialog box (click Attach on the Toolbars tab of the Customize dialog box). You cannot copy command bars to a workbook by using Microsoft® Visual Basic® for Applications (VBA) code. After you have copied a command bar to a workbook, you can delete it from the workspace by clicking Delete on the Toolbars tab of the Customize dialog box or by using the Delete method of the CommandBars collection.

Note
When you open a workbook, Excel copies the workbook's custom command bars that do not already exist in the workspace to the workspace. These copied command bars are not deleted from the workspace when you close your workbook. If you want custom command bars to be available only when your workbook is open, you must programmatically delete them from the workspace when your workbook closes. When you delete a command bar in this fashion, you are removing only the workspace copy, not the workbook copy. The workbook copy will be copied again to the workspace the next time your custom application opens. If you do not delete the workspace copy and the workspace copy of the command bar is modified by the user, the workbook copy will not be recopied to the workspace when your workbook is reopened.

Note
You cannot use VBA to copy a workspace command bar to a workbook or to delete a workbook command bar from a workbook. The only way to delete a custom command bar from a workbook is to use the Delete button in the Attach Toolbars dialog box.

Note
If a control on the workspace copy of the command bar calls code that exists in a workbook and the workbook is not open when the control is used, the workbook is immediately opened and made visible.

Command bars that you create to distribute with a custom application should be stored in the application's workbook or template.

Microsoft FrontPage Command Bars

In Microsoft® FrontPage®, you can create custom command bars that will be available in the FrontPage workspace. In other words, custom command bars are generally available and are not linked to a specific FrontPage-based web. FrontPage command bars are saved in a file named CmdUI.PRF. If user profiles have been set up for multiple users, CmdUI.PRF is stored in the C:\Windows\Profiles\UserName\Application Data\Microsoft\FrontPage\State subfolder. If user profiles have not been set up, CmdUI.PRF is stored in the C:\Windows\Application Data\Microsoft\FrontPage\State subfolder.

Microsoft Outlook Command Bars

In Microsoft® Outlook®, command bars are stored in the Outlook workspace in the Outcmd.dat file. If user profiles have been set up for multiple users, the Outcmd.dat file is stored in the C:\Windows\Profiles\UserName\Application Data\Microsoft\Outlook subfolder. If user profiles have not been set up, the Outcmd.dat file is stored in the C:\Windows\Application Data\Microsoft\Outlook subfolder.

Microsoft PowerPoint Command Bars

In Microsoft® PowerPoint®, custom command bars are stored only in the application workspace in a file named PPT.pcb. If user profiles have been set up for multiple users, the PPT.pcb file is stored in the C:\Windows\Profiles\UserName\Application Data\Microsoft\PowerPoint subfolder. If user profiles have not been set up, the PPT.pcb file is stored in the C:\Windows\Application Data\Microsoft\PowerPoint subfolder.

Note
Command bars are not visible while a PowerPoint presentation is running. Therefore, changes you make to PowerPoint command bars are limited to those that are available in the design-time environment.

Microsoft Word Command Bars

When you create a command bar in Microsoft® Word, you have the option of storing that command bar in the Normal.dot template, in a separate template, or in the currently active document. If the command bar is stored with the Normal.dot template, it will be available to any document, even if the document is based on a different template. If the command bar is stored with the currently active document and that document is a template, the command bar will be available for any document created based on that template. If the command bar is stored with a document, it will be available only when that document is open.

In Word, custom command bars are stored in the Normal.dot file by default. If user profiles have been set up for multiple users, this file is stored in the C:\Windows\Profiles\UserName\Application Data\Microsoft\Templates subfolder. If user profiles have not been set up, the Normal.dot file is stored in the C:\Windows\Application Data\Microsoft\Templates subfolder. Command bars created in other documents or in document templates are stored with that document or template.

When you create custom applications based on Word, it is typical to store your code in a custom document template so that the code is available to documents created based on your template. You should also store any custom command bars in the template on which your custom application documents are based. If you must have your command bars available to documents based on more than one template, you can store them in a global template or add-in.

Note
It is not a good practice to store your code or command bars in a user's Normal.dot file. Many users or system administrators protect the Normal.dot file from modifications to prevent the file from being infected by a virus or to keep the file from growing to an unreasonable size. Because you can never be sure that Normal.dot will be available for modifications, you should use your own custom template or add-in to distribute your code.

When you create custom command bars in Word by using the Customize dialog box, you specify where the command bar is stored by using the Save In box on the Commands tab of the Customize dialog box. When you create a custom command bar in Word by using Microsoft®Visual Basic® for Applications (VBA) code, you specify where it is stored by using the CustomizationContext property of the Application object.

Manipulating Command Bars and Command Bar Controls with VBA Code

The command bars object model exposes a wealth of objects, collections, properties, and methods that you can use to show, hide, and modify existing command bars and command bar controls, and create new ones. In addition, you can specify a Microsoft® Visual Basic® for Applications (VBA) procedure to run when a user clicks a command bar button or to respond to events triggered by a command bar or command bar control. The following sections provide a broad overview of the kinds of things you can do in your custom Microsoft® Office applications and how to accomplish them.

Note

Many of the examples in this section refer to the "Menu Bar" CommandBar object. This is the name of the main menu bar in Microsoft® Word, Microsoft® PowerPoint® and Microsoft® Access. The main menu bar in Microsoft® Excel is called "Worksheet Menu Bar." To experiment with sample code that refers to the "Menu Bar" CommandBar object in Excel, simply change the reference from "Menu Bar" to "Worksheet Menu Bar."

Getting Information About Command Bars and Controls

Each Microsoft® Office application contains dozens of built-in command bars and can contain as many custom command bars as you choose to add. Each command bar can be one of three types: menu bar, toolbar, or pop-up menu. All of these command bar types can contain additional command bars and any number of controls. To get a good understanding of the command bars object model, it is often best to start by examining the various command bars and controls in an existing application.

You can use the following procedure to print (to the Debug window) information about any command bar and its controls:

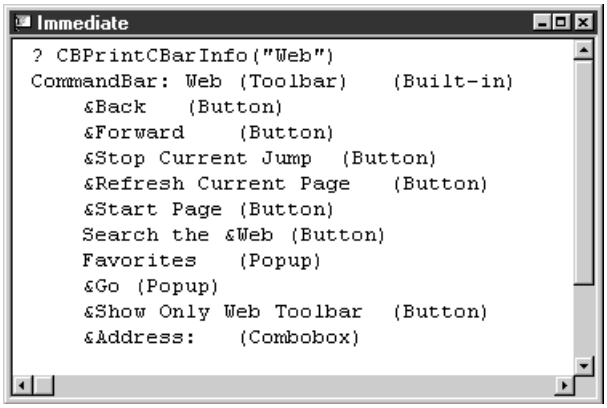
```
Function CBPrintCBarInfo(strCBarName As String) As Variant
  ' This procedure prints (to the Debug window) information  about the command bar specified in the strCBarName argument
  ' and information about each control on that command bar.
  Dim cbrBar                As CommandBar
  Dim ctlCBarControl        As CommandBarControl
  Const ERR_INVALID_CMDBARNAME   As Long = 5

  On Error GoTo CBPrintCBarInfo_Err
  Set cbrBar = Application.CommandBars(strCBarName)
  Debug.Print "CommandBar: " & cbrBar.Name & vbTab & "("  & CBGetCBType(cbrBar) & ")" & vbTab & "(" _
    & IIf(cbrBar.BuiltIn, "Built-in", "Custom") & ")"
  For Each ctlCBarControl In cbrBar.Controls
    Debug.Print vbTab & ctlCBarControl.Caption & vbTab & "(" & CBGetCBCtlType(ctlCBarControl) & ")"
  Next ctlCBarControl
CBPrintCBarInfo_End:
  Exit Function
CBPrintCBarInfo_Err:
  Select Case Err.Number
    Case ERR_INVALID_CMDBARNAME
      CBPrintCBarInfo = """ & strCBarName &  "' is not a valid command bar name!"
    Case Else
      CBPrintCBarInfo = "Error: " & Err.Number & " - " & Err.Description
  End Select
  Resume CBPrintCBarInfo_End
End Function
```

You call this procedure in the Visual Basic Editor's Immediate window by using the name of a command bar as the only argument. For example, if you execute the following command from the Immediate window:

```
? CBPrintCBarInfo("Web")
```

You will see a listing of all the controls and their control types on the Office Web built-in toolbar, as shown in the following figure.

Listing of Web Toolbar Controls



When a control type is shown as "Popup," as with the Favorites control above, the control itself is a command bar. You can get a listing of the controls on a pop-up menu command bar by calling the CBPrintCBarInfo procedure and passing in the name of the pop-up menu as the strCBarName argument. For example:

```
? CBPrintCBarInfo("Favorites")
```

Note that the CBPrintCBarInfo procedure calls two other custom procedures to get the command bar type and the control type. To get information about every command bar of any type in an application, you can use the PrintAllCBarInfo procedure.

Note
To refer to a member of the CommandBars collection, use the name of the CommandBar object or an index value that represents the object's location in the collection. The controls on a command bar are members of the CommandBar object's Controls collection. To refer to a control in the Controls collection, use the control's Caption property or an index value that represents the control's location within the collection. All collections are indexed beginning with 1.

Creating a Command Bar

You can create toolbars by using the Customize dialog box or by using Microsoft® Visual Basic® for Applications (VBA) code in any Microsoft® Office application. In Microsoft® Access, you also can create menu bars and pop-up menus by using the Customize dialog box. However, in all other Office applications, you must use VBA code to create menu bars or pop-up menus.

You create a custom command bar by using the CommandBars collection's Add method. The Add method creates a toolbar by default. To create a menu bar or pop-up menu, use the msoBarMenuBar or msoBarPopup constant in the Add method's Position argument. The following code sample illustrates how to create all three types of CommandBar objects:

```
Dim cbrCmdBar       As CommandBar
Dim strCBarName     As String

' Create a toolbar.
strCBarName = "MyNewToolbar"
Set cbrCmdBar = Application.CommandBars.Add(Name:=strCBarName)
' Create a menu bar.
strCBarName = "MyNewMenuBar"
Set cbrCmdBar = Application.CommandBars.Add(Name:=strCBarName, Position:=msoBarMenuBar)
' Create a pop-up menu.
strCBarName = "MyNewPopupMenu"
Set cbrCmdBar = Application.CommandBars.Add(Name:=strCBarName, Position:=msoBarPopup)
```

After you have created a command bar, you still must add any controls that you want and set the command bar's Visible property to True.

Hiding and Showing a Command Bar

You hide or show a toolbar by using the CommandBar object's Visible property. When you display a toolbar, you can specify where it will appear on the screen by using the Position property. For example, the following code sample takes three arguments: the name of a toolbar, a Boolean value indicating whether it should be visible or hidden, and a value matching an msoBarPosition constant specifying where on the screen the toolbar should be displayed. The sample code also illustrates how to use the CommandBar object's Type property to make sure the specified command bar is a toolbar:

```
Function CBToolbarShow(strCBarName As String, blnVisible As Boolean, Optional lngPosition As Long = msoBarTop) As Boolean
  ' This procedure displays or hides the command bar specified in the strCBarName argument according to the value of the blnVisible  argument. The optional lngPosition argument specifies where the  command bar will appear on the screen.
  Dim cbrCmdBar As CommandBar

  On Error GoTo CBToolbarShow_Err
  Set cbrCmdBar = Application.CommandBars(strCBarName)
  ' Show only toolbars.
  If cbrCmdBar.Type > msoBarTypeNormal Then
    CBToolbarShow = False
    Exit Function
  End If
  ' If Position argument is invalid, set to the default  msoBarTop position.
  If lngPosition < msoBarLeft Or lngPosition > msoBarMenuBar Then
    lngPosition = msoBarTop
  End If
  With cbrCmdBar
    .Visible = blnVisible
    .Position = lngPosition
  End With
  CBToolbarShow = True
CBToolbarShow_End:
  Exit Function

CBToolbarShow_Err:
  CBToolbarShow = False
  Resume CBToolbarShow_End
End Function
```

You display a custom menu bar by setting its Visible property to True and setting the existing menu bar's Visible property to False.

Copying a Command Bar

You must use Microsoft® Visual Basic® for Applications (VBA) code to copy an existing command bar. You create a copy of a command bar by creating a new command bar of the same type as the one you want to copy, and then use the CommandBarControl object's Copy method to copy each control from the original command bar to the new command bar. The following procedure illustrates how to use VBA to copy an existing command bar:

```
Function CBCopyCommandBar(strOrigCBName As String, strNewCBName As String, Optional blnShowBar As Boolean =
False) As Boolean
  ' This procedure copies the command bar named in the strOrigCBName argument to a new command bar specified in the
strNewCBName argument.
  Dim cbrOriginal        As CommandBar
  Dim cbrCopy            As CommandBar
  Dim ctlCBarControl     As CommandBarControl
  Dim lngBarType         As Long

  On Error GoTo CBCopy_Err
  Set cbrOriginal = CommandBars(strOrigCBName)
  lngBarType = cbrOriginal.Type
  Select Case lngBarType
    Case msoBarTypeMenuBar
      Set cbrCopy = CommandBars.Add(Name:=strNewCBName, Position:=msoBarMenuBar)
    Case msoBarTypePopup
      Set cbrCopy = CommandBars.Add(Name:=strNewCBName, Position:=msoBarPopup)
    Case Else
      Set cbrCopy = CommandBars.Add(Name:=strNewCBName)
  End Select
  ' Copy controls to new command bar.
  For Each ctlCBarControl In cbrOriginal.Controls
    ctlCBarControl.Copy cbrCopy
  Next ctlCBarControl
  ' Show new command bar.
  If blnShowBar = True Then
    If cbrCopy.Type = msoBarTypePopup Then
      cbrCopy.ShowPopup
    Else
      cbrCopy.Visible = True
    End If
  End If
  CBCopyCommandBar = True
CBCopy_End:
  Exit Function
CBCopy_Err:
  CBCopyCommandBar = False
  Resume CBCopy_End
End Function
```

This procedure will not work if you pass in the name of an existing command bar in the strNewCBName argument, because that argument represents the name of the new command bar.

Note
If you copy a pop-up menu and set the blnShowBar argument to True, the pop-up menu will be displayed at the current location of the mouse pointer. For more information about displaying pop-up menus, search the Microsoft Office Visual Basic Reference Help index for "ShowPopup method."

Deleting a Command Bar

You can delete toolbars and menu bars from the Customize dialog box or by using Microsoft® Visual Basic® for Applications (VBA). You can delete pop-up menus only by using VBA. Use the Delete method of the CommandBars collection to remove an existing command bar from the collection. The following procedure illustrates one way to delete a CommandBar object:

```
Function CBDeleteCommandBar(strCBarName As String) As Boolean
  On Error Resume Next
  Application.CommandBars(strCBarName).Delete
End Function
```

An error will occur if strCBarName is not the name of an existing command bar. The procedure uses the On Error Resume Next statement to ignore this error because, if an error occurs, it means there is nothing to delete. In addition, it could mean you tried to delete a built-in command bar, such as Standard, which cannot be deleted.

Preventing Users from Modifying Custom Command Bars

There might be circumstances when you want to make sure that users of your custom application cannot delete or disable your custom command bars by using the Customize dialog box. The easiest, but least secure, way to keep users from modifying your custom command bars is to disable the command bars and make sure they are visible only when absolutely necessary. You disable a command bar by setting its Enabled property to False. You hide a command bar by setting its Visible property to False. However, hiding a command bar does nothing to prevent users from getting to the bar through the Customize dialog box.

To completely restrict access to your custom command bars, you must restrict all access to the Customize dialog box. This dialog box can be accessed in three ways: by pointing to Toolbars on the View menu and then clicking Customize; by right-clicking any command bar and then clicking Customize on the shortcut menu; and by clicking Customize on the Tools menu.

All Microsoft® Office applications use the Toolbar List pop-up command bar to provide access to built-in and custom command bars. The Toolbar List command bar appears when you click Toolbars on the View menu or when you right-click any command bar. If you set the Enabled property of the Toolbar List command bar to False as shown in the following line of code, a user will not be able to open the Customize dialog box from either of these access points:

CommandBars("Toolbar List").Enabled = False

Note
Because of the way the Toolbar List command bar is constructed, you cannot disable any of its commands. The only way to disable commands on this command bar is to disable the entire command bar.

Because you also can open the Customize dialog box by clicking Customize on the Tools menu, you will have to disable this command as well to completely restrict access to your custom command bars. The following procedure illustrates how to disable all access to the Customize dialog box:

```
Sub AllowCommandBarCustomization(blnAllowEnabled As Boolean)
  ' This procedure allows or prevents access to the command bars Customize dialog box according to the value of the blnAllowEnabled  argument.
  CommandBars("Tools").Controls("Customize...").Enabled = blnAllowEnabled
  CommandBars("Toolbar List").Enabled = blnAllowEnabled
End Sub
```

Working with Personalized Menus

Personalized menus are a feature in Microsoft® Office XP that makes it possible for you see a collapsed subset of menu items that you use most often. You specify whether personalized menus are enabled by pointing to Toolbars on the View menu, clicking Customize, clicking the Options tab, and then selecting the Always show full menus commands first check box. Personalized menus are turned on by default.

Note
The personalized menus feature does not apply to shortcut menus.

You can turn on personalized menus for all command bars in an application or for individual command bars only. You can use the CommandBars collection's AdaptiveMenus property to specify whether personalized menus are on or off for all command bars. You use a CommandBar object's AdaptiveMenu property to specify whether that object's menus are displayed as personalized menus.

You use a CommandBarControl object's Priority property to specify whether a control on a menu will be visible when personalized menus are on. When you add a custom CommandBarControl object to a command bar, it will be visible by default. If you set a control's Priority property to 1, the control will always be visible. If you set the Priority property to 0, the control will initially be visible but might be hidden by the host application if it is not used regularly. When a control is hidden, it is still available on the menu, but you must expand the menu to see it.

The CommandBarControl object's IsPriorityDropped property specifies whether a control is currently displayed. When this property is set to True, the control is hidden. Selecting a control that has its IsPriorityDropped property set to True changes the property setting to False, which makes the control visible the next time its menu is displayed.

The host application might change the IsPriorityDropped property setting if the control is not used again within a certain time period. For more information about how long a control remains visible, search the Microsoft Office Visual Basic Reference Help index for "IsPriorityDropped property."

The following procedure turns personalized menus on or off for all command bars or a single command bar according to the value of the blnState argument:

```
Function SetPersonalizedMenuState(blnState As Boolean, Optional cbrBar As CommandBar = Nothing)
  ' This procedure sets the AdaptiveMenus property to the value of the  blnState argument. If a CommandBar object is supplied in the cbrBar  argument, the AdaptiveMenu property for that command bar is set to the value of the blnState argument.
  On Error Resume Next
```

```
  If cbrBar Is Nothing Then
    Application.CommandBars.AdaptiveMenus = blnState
  Else
    cbrBar.AdaptiveMenu = blnState
  End If
End Function
```

The following procedure changes the setting of the Priority property for a menu item:

```
Function PromoteMenuItem(cbrBar As CommandBar,  strItemCaption As String)
  ' This procedure changes the Priority property setting for the  cbrBar command bar control whose Caption property setting
  ' matches the value of the strItemCaption argument.
  Dim ctlMenuItem As CommandBarControl

  On Error Resume Next
  If cbrBar.AdaptiveMenu = False Then Exit Function
  Set ctlMenuItem = cbrBar.Controls(strItemCaption)
  With ctlMenuItem
    If .Priority <> 1 Then
      .Priority = 1
    End If
  End With
End Function
```

Working with Images on Command Bar Buttons

Every built-in command bar button has an image associated with it. You can use these images on your own command bar buttons as long as you know the FaceId property value of the built-in button that contains the image. The values for the FaceId property range from zero (no image) to the total number of button images used in the host application (typically a few thousand). One easy way to browse the available button images is to build a toolbar, add some buttons, and assign FaceId property values to those buttons. The buttons display the image associated with the specified FaceId property value. For example, to see button images with values from 200 to 299, you would call CBShowButtonFaceIDs from the Immediate window in this way:

```
? CBShowButtonFaceIDs(200, 299)
```

And the button images would display as shown in the following figure.

A Collection of Built-in Toolbar Icons

You can see the value of the FaceId property for any image on the command bar by resting your mouse pointer on the image until the value appears in the button's ToolTip.

Another way to copy the image from one command bar button to another is to use the FindControl method of the CommandBars collection to determine the value of the FaceId property for the image you want to copy. Then, you can use the CommandBarControl object's CopyFace and PasteFace methods to copy the image to a new control. The following sample code illustrates how to use these methods to paste the icon associated with an existing command bar button to a new command bar button.

```
Private Sub CBCopyIconDemo()
  ' This procedure demonstrates how to copy the image associated with a known toolbar button to a new toolbar button. This
example copies the image associated with the "Contents and Index" control on the Help menu to a new command bar control.
  Dim cbrNew              As CommandBar
  Dim ctlNew              As CommandBarControl
  Const ERR_CMDBAR_EXISTS   As Long = 5

  On Error Resume Next
  Set cbrNew = CommandBars.Add("TestCopyFaceIcon")
  If Err = ERR_CMDBAR_EXISTS Then
    Call CBDeleteCommandBar("TestCopyFaceIcon")
    Set cbrNew = CommandBars.Add("TestCopyFaceIcon")
  ElseIf Err <> 0 Then
    Exit Sub
  End If
```

```
  On Error GoTo 0
  Set ctlNew = cbrNew.Controls.Add(msoControlButton)
  Call CBCopyControlFace("Help", "Contents and Index")
  With ctlNew
    .PasteFace
  End With
  cbrNew.Visible = True
End Sub
```

This procedure calls two other custom procedures. The CBDeleteCommandbar procedure was discussed in the previous section and is used to delete the command bar if it already exists. The CBCopyControlFace procedure copies the image of the specified control to the Clipboard:

```
Function CBCopyControlFace(strCBarName As String, strCtlCaption As String)
  ' This procedure uses the CopyFace method to copy the image associated with the control specified in the strCtlCaption argument to the Clipboard.
  Dim ctlCBarControl As CommandBarControl

  Set ctlCBarControl = CommandBars.FindControl(msoControlButton, CBGetControlID(strCBarName, strCtlCaption))
  ctlCBarControl.CopyFace
End Function
```

The CBCopyControlFace procedure uses the CBGetControlID procedure as the Id argument for the FindControl method. CBGetControlID returns the Id property for the specified control by using the following line:

```
CBGetControlID = Application.CommandBars(strCBarName).Controls(strControlCaption).ID
```

The CommandBar object also supports a FindControl method that searches for the specified control only on the CommandBar object itself.

To see these procedures working together, place the insertion point (cursor) anywhere in the CBCopyIconDemo procedure and use the F8 key to step through the code. Try changing the command bar name and control name to copy different images to the new control.

Working with Command Bar Controls

Each CommandBar object has a CommandBarControls collection, which contains all the controls on the command bar. You use the Controls property of a CommandBar object to refer to a control on a command bar. If the control is of the type msoControlPopup, it also will have a Controls collection representing each control on the pop-up menu. Pop-up menu controls represent menus and submenus and can be nested several layers deep, as shown in the second example below.

In this example, the code returns a reference to the New button on the Standard toolbar:

```
Dim ctlCBarControl As CommandBarControl
Set ctlCBarControl = Application.CommandBars("Standard").Controls("New")
```

Here the code returns a reference to the Macros... control on the Macro pop-up menu on the Tools menu on the "Menu Bar" main menu bar:

```
Dim ctlCBarControl As CommandBarControl
Set ctlCBarControl = Application.CommandBars("Menu Bar").Controls("Tools").Controls("Macro").Controls("Macros...")
```

Note
The "Menu Bar" CommandBar object refers to the main menu bar in Microsoft® Word, Microsoft® PowerPoint®, and Microsoft® Access. The main menu bar in Microsoft® Excel is called "Worksheet Menu Bar." To experiment with sample code that refers to the "Menu Bar" CommandBar object in Excel, simply change the reference from "Menu Bar" to "Worksheet Menu Bar."

Because each pop-up menu control is actually a CommandBar object itself, you also can refer to them directly as members of the CommandBars collection. For example, the following line of code returns a reference to the same control as the previous example:

```
Set ctlCBarControl = Application.CommandBars("Macro").Controls("Macros...")
```

When you have a reference to a control on a command bar, you can access all available properties and methods of that control.

Note
When you refer to a command bar control by using the control's Caption property, you must be sure to specify the caption exactly as it appears on the menu. For example, in the previous code sample, the reference to the control caption "Macros..." requires the ellipsis (...) so it matches how the caption appears on the menu.

Adding Controls to a Command Bar

To add a control to a command bar, use the Add method of the Controls collection, specifying which type of control you want to create. You can add controls of the following type: button (msoControlButton), text box (msoControlEdit), drop-down list box (msoControlDropdown), combo box (msoControlComboBox), or pop-up menu (msoControlPopup).

The following example adds a new menu to the "Menu Bar" command bar and then adds three controls to the menu:

```
Private Sub CBAddMenuDemo()
  ' Illustrates adding a new menu and filling it with controls. Also illustrates deleting a menu control from a menu bar.
  ' In Microsoft Excel, the main menu bar is named "Worksheet Menu Bar"  rather than "Menu Bar".
  Dim strCBarName    As String
  Dim strMenuName    As String
  Dim cbrMenu        As CommandBarControl

  strCBarName = "Menu Bar"
  strMenuName = "Custom Menu Demo"
  Set cbrMenu = CBAddMenu(strCBarName, strMenuName)
  ' Note: The following use of the MsgBox function in  the OnAction property setting will work only with command bars in Microsoft Access. In the other Office  applications, you call built-in VBA functions for the OnAction property setting. To call a built-in VBA  function from a command bar control in the other Office applications, you must create a custom procedure that uses the VBA function and call that custom procedure in  the OnAction property setting.
  Call CBAddMenuControl(cbrMenu, "Item 1", "=MsgBox('You selected Menu1 Control 1.')")
  Call CBAddMenuControl(cbrMenu, "Item 2", "=MsgBox('You selected Menu1 Control 2.')")
  Call CBAddMenuControl(cbrMenu, "Item 3", "=MsgBox('You selected Menu1 Control 3.')")
  ' The menu should now appear to the right of the Help menu on the menu bar. To see how to delete  a menu from a menu bar, press F8 to step through the remaining code.
  Stop
  Call CBDeleteCBControl(strCBarName, strMenuName)
End Sub
```

Note that the CBAddMenuDemo procedure calls three other procedures: CBAddMenu, CBAddMenuControl, and CBDeleteCBControl. CBAddMenu returns the new pop-up menu as a CommandBarControl object. In addition, if the command bar specified by the strCBarName argument does not exist, CBAddMenu creates it. CBAddMenuControl adds a button control to the menu created by CBAddMenu and sets the control's OnAction property to the code to run when the button is clicked. CBDeleteCBControl just removes the menu created in the CBAddMenu procedure. CBAddMenu and CBAddMenuControl are shown below:

```
Function CBAddMenu(strCBarName As String,  strMenuName As String) As CommandBarControl
  ' Add the menu named in strMenuName to the  command bar named in strCBarName.
  Dim cbrBar            As CommandBar
  Dim ctlCBarControl    As CommandBarControl

  On Error Resume Next
  Set cbrBar = CommandBars(strCBarName)
  If Err <> 0 Then
    Set cbrBar = CommandBars.Add(strCBarName)
    Err = 0
  End If
 With cbrBar
    Set ctlCBarControl = .Controls.Add(msoControlPopup)
    ctlCBarControl.Caption = strMenuName
  End With
  Set CBAddMenu = ctlCBarControl
End Function

Function CBAddMenuControl(cbrMenu As CommandBarControl, strCaption As String,  strOnAction As String) As Boolean
  ' Add a button control to the menu specified in cbrMenu and set  its Caption and OnAction properties to the values specified in  the strCaption and strOnAction arguments.
  Dim ctlCBarControl As CommandBarControl

  With cbrMenu
    Set ctlCBarControl = .Controls.Add(msoControlButton)
    With ctlCBarControl
      .Caption = strCaption
      .OnAction = strOnAction
      .Tag = .Caption
    End With
  End With
End Function
```

You normally set the OnAction property to the name of a procedure to run when the button is clicked. In the example above, however, the OnAction property is set by using a string that contains the built-in VBA MsgBox function and the text to display in the message box. When multiple command bar controls use the same OnAction property setting, you can use the ActionControl property and the Parameter property to determine which command bar button is calling the procedure. In addition, you can use Microsoft® Visual Basic® for Applications (VBA) code that executes in response to CommandBar and CommandBarControl events.

You can add any built-in command bar control to a command bar by using the Id property of the built-in control. The following procedure illustrates a technique to add a built-in control to a command bar.

```
Function CBAddBuiltInControl(cbrDestBar As CommandBar, strCBarSource As String, strCtlCaption As String) As Boolean
  ' This procedure adds the built-in control specified in  strCtlCaption from the strCBarSource command bar to the
  ' command bar specified by cbrDestBar.

  On Error GoTo CBAddBuiltInControl_Err
  If CBDoesCBExist(strCBarSource) <> True Then
    CBAddBuiltInControl = False
    Exit Function
  End If
  cbrDestBar.Controls.Add ID:=CBGetControlID(strCBarSource, strCtlCaption)
  CBAddBuiltInControl = True
CBAddBuiltInControl_End:
  Exit Function
CBAddBuiltInControl_Err:
  CBAddBuiltInControl = False
  Resume CBAddBuiltInControl_End
End Function
```

Note
When you specify a control's Id property, you also specify the action the control will take when it is selected and, if applicable, the image that appears on the face of the control. To add a control's image without its built-in action, you specify only the FaceId property.

Showing and Enabling Command Bar Controls

You specify whether a command bar control appears on a command bar by using its Visible property. You specify whether a command bar control appears enabled or disabled (grayed out) by using its Enabled property. For example, the following two lines of code could be used to toggle the Visible and Enabled properties of the named controls:

```
Application.CommandBars("Menu Bar").Controls("Edit").Enabled =
            Not Application.CommandBars("Menu Bar").Controls("Edit").Enabled
Application.CommandBars("Formatting").Controls("Font").Visible = _
            Not Application.CommandBars("Formatting").Controls("Font").Visible
```

Note
The "Menu Bar" CommandBar object refers to the main menu bar in Microsoft® Word, Microsoft® PowerPoint®, and Microsoft® Access. The main menu bar in Microsoft® Excel is called "Worksheet Menu Bar." To experiment with sample code that refers to the "Menu Bar" CommandBar object in Excel, simply change the reference from "Menu Bar" to "Worksheet Menu Bar."

When a command bar control's Enabled property is False, the control appears on the command bar but is disabled and cannot be manipulated.

Visually Indicating the State of a Command Bar Control

Many menu commands or toolbar buttons are used to toggle the state of some part of an application from one condition to another. For example, in Microsoft®Office applications, the Bold button and the Align Left button will appear pressed in or not pressed in, depending on the formatting applied to text at the current selection. You can achieve this same effect with your custom command bar button controls by setting the State property to one of the msoButtonState constants.

Note
The State property is read-only for built-in command bar controls.

The following procedure shows how to explicit set the State property of a custom command bar button control:

```
Function CBCtlSetState(strCBarName As String,  strCtlCaption As String) As Boolean
  ' Set the State property of the strCtlCaption control on the strCBarName command bar. The State property is
  ' read-only for built-in controls, so if strCtlCaption  is a built-in control, return False and exit the procedure.
  Dim ctlCBarControl As CommandBarControl

  On Error Resume Next
  Set ctlCBarControl = Application.CommandBars(strCBarName).Controls(strCtlCaption)
  If ctlCBarControl.BuiltIn = True Then
```

```
    CBCtlSetState = False
    Exit Function
  End If
  If ctlCBarControl.Type <> msoControlButton Then
    CBCtlSetState = False
    Exit Function
  End If
  CtlCBarControl. State =
  If C.State = MsoButtonDown Then
      C.State = MsoButtonUp
    Else If C.State = MsoButtonUp Then
      C.State = MsoButtonDown
    Else
      'State is mixed, leave it
  End If
  If Err = 0 Then
    CBCtlSetState = True
  Else
    CBCtlSetState = False
  End If
End Function
```

Working with Command Bar Events

You can use command bar event procedures to run your own code in response to an event. In addition, you can use these event procedures to substitute your own code for the default behavior of a built-in control. The CommandBars collection and the CommandBarButton and CommandBarComboBox objects expose the following event procedures that you can use to run code in response to an event:

- The CommandBars collection supports the OnUpdate event, which is triggered in response to changes made to a Microsoft®Office document that might affect the state of any visible command bar or command bar control. For example, the OnUpdate event occurs when a user changes the selection in an Office document. You can use this event to change the availability or state of command bars or command bar controls in response to actions taken by the user.
  Note
  The OnUpdate event can be triggered repeatedly in many different contexts. Any code you add to this event that does a lot of processing or performs a number of actions might affect the performance of your application.
- The CommandBarButton control exposes a Click event that is triggered when a user clicks a command bar button. You can use this event to run code when the user clicks a command bar button.
- The CommandBarComboBox control exposes a Change event that is triggered when a user makes a selection from a combo box control. You can use this method to take an action depending on what selection the user makes from a combo box control on a command bar.

To expose these events, you must first declare an object variable in a class module by using the WithEvents keyword. The following code, entered in the Declarations section of a class module, creates object variables representing the CommandBars collection, three command bar buttons, and a combo box control on a custom toolbar:

```
Public WithEvents colCBars        As Office.CommandBars
Public WithEvents cmdBold         As Office.CommandBarButton
Public WithEvents cmdItalic       As Office.CommandBarButton
Public WithEvents cmdUnderline    As Office.CommandBarButton
Public WithEvents cboFontSize     As Office.CommandBarComboBox
```

When you use the WithEvents keyword to declare an object variable in a class module, the object appears in the Object box in the Code window, and when you select it, the object's events are available in the Procedure box. For example, if the clsCBarEvents class module contained the previous code, you could select the colCBars, cmdBold, cmdItalic, cmdUnderline, and cboFontSize objects from the Object drop-down list and each object's event procedure template would be added to your class module as follows:

```
Private Sub colCBars_OnUpdate()
  ' Insert code you want to run in response to selection changes in an  Office document.
End Sub

Private Sub cmdBold_Click (ByVal Ctrl As Office.CommandBarButton,  CancelDefault As Boolean)
  ' Insert code you want to run in response to this event.
End Sub
Private Sub cmdItalic_Click (ByVal Ctrl As Office.CommandBarButton, CancelDefault As Boolean)
  ' Insert code you want to run in response to this event.
End Sub

Private Sub cmdUnderline_Click (ByVal Ctrl As Office.CommandBarButton,  CancelDefault As Boolean)
  ' Insert code you want to run in response to this event.
```

```
End Sub

Private Sub cboFontSize_Change (ByVal Ctrl As Office.CommandBarComboBox)
   ' Insert code you want to run when a selection is made in a combo box.
End Sub
```

You add to the event procedures the code that you want to run when the event occurs.

Note
If you set the variable for a command bar control object to a built-in command button, you can set the Click event's CancelDefault argument to True to prevent the button's default behavior from occurring. This behavior is useful if you are developing an add-in and want code to run instead of, or in addition to, the application code that runs when a built-in button is clicked.

After you have added code to the event procedures, you create an instance of the class in a standard or class module and use the Set statement to link the control events to specific command bar controls. In the following example, the InitEvents procedure is used in a standard module to link clsCBarEvents object variables to specific command bar controls on the Formatting Example toolbar:

```
Option Explicit
Dim clsCBClass As New clsCBEvents

Sub InitEvents()
  Dim cbrBar As Office.CommandBar

  Set cbrBar = CommandBars("Formatting Example")
  With cbrBar
    Set clsCBClass.cmdBold = .Controls("Bold")
    Set clsCBClass.cmdItalic = .Controls("Italic")
    Set clsCBClass.cmdUnderline = .Controls("Underline")
    Set clsCBClass.cboFontSize = .Controls("Set Font Size")
  End With
  Set clsCBClass.colCBars = CommandBars
End Sub
```

When the InitEvents procedure runs, the code you placed in the command bar's and command bar controls' event procedures will run whenever the related event occurs.

Working with Document Properties

Every file created by a Microsoft® Office XP application supports a set of built-in document properties. In addition, you can add your own custom properties to an Office document either manually or through code. You can use document properties to create, maintain, and track information about an Office document such as when it was created, who the author is, where it is stored, and so on. In addition, when you save an Office document as an HTML file, all of the document properties are written to the HTML file within <XML> tag pairs. This makes it possible for you to use document properties to track or index files according to properties you specify, regardless of what format you use to save the file.

Note
Office uses the term "document" to represent any file created by using an Office application.

You can view and set built-in and custom document properties by clicking Properties on the File menu. (In Microsoft® Access, click Database Properties on the File menu.)

In This Section

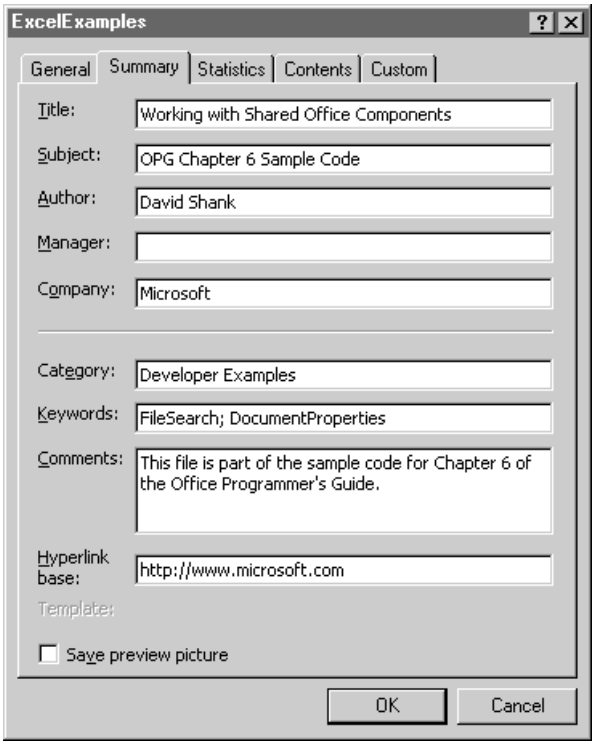
Document Properties in Microsoft Access, Microsoft FrontPage, and Microsoft Outlook
Understand document properties in Microsoft® Office XP applications.

Working with the HTMLProject Object
Determine the current state of an Office document, access individual HTMLProjectItem objects, and save current projects and documents.

Document Properties in Microsoft Access, Microsoft FrontPage, and Microsoft Outlook

Microsoft® Access does not use the DocumentProperties collection to store the built-in and custom properties displayed in its Database Properties dialog box. You can access these properties by using Data Access Objects (DAO) in an .mdb-type database and in a SQL database. For more information about database properties, search the Microsoft Access Visual Basic Reference Help index for "database properties."

The Document Properties Dialog Box



Microsoft® FrontPage® also does not use the DocumentProperties collection to store the built-in and custom properties displayed in its Page Properties dialog box (File menu). In FrontPage, built-in and custom properties are stored in the MetaTags and Properties collections of a WebFile object.

Microsoft® Outlook®does not provide a Document Properties dialog box from the File menu as the other Microsoft® Office applications do.

You access the DocumentProperties collection by using the BuiltInDocumentProperties and CustomDocumentProperties properties of an Office document. For an example that prints all built-in and custom document properties for an Office document to the Immediate window, see the PrintAllDocProperties procedure in the modDocumentPropertiesCode module in the ExcelExamples.xls file.

Note
The BuiltInDocumentProperties property returns a collection that contains properties that might apply only to certain Office applications. If you try to return the value of these properties in the wrong context, an error occurs. The sample code shows how to trap this error and continue to identify all the properties that are valid in a given context.

The following code sample shows how to determine the value of a built-in document property. The GetBuiltInProperty procedure accepts an Office document object (Workbook, Document, or Presentation) and a property name and returns the value of the built-in property, if available:

```
Function GetBuiltInProperty(objDoc As Object,  strPropname As String) As Variant
  ' This procedure returns the value of the built-in document  property specified in the strPropName argument for the Office
  ' document object specified in the objDoc argument.
  Dim prpDocProp      As DocumentProperty
  Dim varValue        As Variant

  Const ERR_BADPROPERTY     As Long = 5
  Const ERR_BADDOCOBJ       As Long = 438
  Const ERR_BADCONTEXT      As Long = -2147467259

  On Error GoTo GetBuiltInProp_Err
  Set prpDocProp = objDoc.BuiltInDocumentProperties(strPropname)
  With prpDocProp
    varValue = .Value
    If Len(varValue) <> 0 Then
      GetBuiltInProperty = varValue
    Else
      GetBuiltInProperty = "Property does not currently have a value set."
    End If
  End With
GetBuiltInProp_End:
  Exit Function
GetBuiltInProp_Err:
  Select Case Err.Number
```

```
    Case ERR_BADDOCOBJ
      GetBuiltInProperty = "Object does not support BuiltInProperties."
    Case ERR_BADPROPERTY
      GetBuiltInProperty = "Property not in collection."
    Case ERR_BADCONTEXT
      GetBuiltInProperty = "Value not available in this context."
    Case Else
  End Select
  Resume GetBuiltInProp_End:
End Function
```

Note
For a complete list of built-in document properties, search the Microsoft Office Visual Basic Reference Help index for "DocumentProperty object."

You can determine the value of an existing custom document property by using the same techniques as those illustrated in the previous code example. The only difference is that you would use the Office document's CustomDocumentProperties collection to return the DocumentProperty object you were interested in.

You use the Add method of the CustomDocumentProperties collection to add a custom DocumentProperty object to the DocumentProperties collection. When you add a custom property, you specify its name, data type, and value. You can also link a custom property to a value in the Office document itself. When you add linked properties, the value of the custom property changes when the value in the document changes. For example, if you add a custom property linked to a named range in a Microsoft® Excel spreadsheet, the property will always contain the current value of the data in the named range.

The following procedure illustrates how to add both static and linked custom properties to the DocumentProperties collection. It is essentially a wrapper around the Add method of the DocumentProperties collection that includes parameter validation and deletes any existing custom property before adding a property that uses the same name.

```
Function AddCustomDocumentProperty(strPropName As String, lngPropType As Long, Optional varPropValue As Variant = "",  Optional blnLinkToContent As Boolean = False,  Optional varLinkSource As Variant = "") As Long
  ' This procedure adds the custom property specified in the strPropName  argument. If the blnLinkToContent argument is True, the custom  property is linked to the location specified by varLinkSource.
  ' The procedure first checks for missing or inconsistent input parameters.  For example, a value must be provided unless the property is linked, and  when you are using linked properties, the source of the link must be provided.
  Dim prpDocProp As DocumentProperty

  ' Validate data supplied in arguments to this procedure.
  If blnLinkToContent = False And Len(varPropValue) = 0 Then
    ' No value supplied for custom property.
    AddCustomDocumentProperty = ERR_CUSTOM_LINKTOCONTENT_VALUE
    Exit Function
  ElseIf blnLinkToContent = True And Len(varLinkSource) = 0 Then
    ' No source provided for LinkToContent scenario.
    AddCustomDocumentProperty = ERR_CUSTOM_LINKTOCONTENT_LINKSOURCE
    Exit Function
  ElseIf lngPropType < msoPropertyTypeNumber Or _
      lngPropType > msoPropertyTypeFloat Then
    ' Invalid value for data type specifier. Must be one of the
    ' msoDocProperties enumerated constants.
    AddCustomDocumentProperty = ERR_CUSTOM_INVALID_DATATYPE
    Exit Function
  ElseIf Len(strPropName) = 0 Then
    ' No name supplied for new custom property.
    AddCustomDocumentProperty = ERR_CUSTOM_INVALID_PROPNAME
    Exit Function
  End If
  Call DeleteIfExisting(strPropName)
  Select Case blnLinkToContent
    Case True
      Set prpDocProp = ActiveWorkbook.CustomDocumentProperties
        .Add(Name:=strPropName, LinkToContent:=blnLinkToContent,  Type:=lngPropType, LinkSource:=varLinkSource)
        ActiveWorkbook.Save
    Case False
      Set prpDocProp = ActiveWorkbook.CustomDocumentProperties. _
        Add(Name:=strPropName, LinkToContent:=blnLinkToContent, Type:=lngPropType, Value:=varPropValue)
  End Select
End Function
```

Note

When you programmatically add a custom property to the DocumentProperties collection and the property is linked to a value in the underlying Office document, you must use the document's Save method, as illustrated previously, before the property value will be reflected correctly for the new DocumentProperty object.

Working with the HTMLProject Object

The HTMLProject object is the top-level object representing the HTML code in a Microsoft® Office document. It is the equivalent of the top-level project branch in the Microsoft Script Editor Project Explorer when it contains an Office document. The HTMLProject object has properties you can use to determine the current state of an Office document and to access individual HTMLProjectItem objects, and methods you can use to save the current project or document.

For example, you can tell if a document is currently opened in the Microsoft Script Editor and if the HTML code that exists in the Script Editor is the same as what is contained in the document. If the HTML code is out of sync, you can programmatically synchronize the HTML code before manipulating the document's contents. You can add HTML to a document programmatically or load it from a file saved on disk. In addition, you can use the objects contained within the HTMLProject object and their properties and methods to manipulate the HTML code or add script to the HTML code.

Note
The HTMLProject object is not available in Microsoft® Access, Microsoft® FrontPage®, or Microsoft® Outlook®. To manipulate the HTML code in an Access DataAccessPage object, you use the object's Document property. To work with the HTML code in a page in FrontPage, you use the HTML tab in the FrontPage design environment.

The HTMLProject object's HTMLProjectItems property returns a collection of all of the HTMLProjectItem objects in the project. The default number of HTMLProjectItem objects in an Office application will depend on the kind of Office document you are working with. The following table shows the default number of HTMLProjectItem objects in a new Office document.

Application	Default number of HTMLProjectItem objects
Microsoft® Excel	5 items (Book, Tab, Sheet1, Sheet2, Sheet3)
Microsoft® PowerPoint®	2 items (SlideMaster, Slide1)
Microsoft® Word	1 item (Document Web Page)

You reference an HTMLProject object by using the HTMLProject property of an Office document. For example, the following code illustrates how to return a reference to the top-level HTMLProject object in each Office application:

```
' Create Word reference:
Dim prjWord As Word.HTMLProjectItem
Set prjWord = ActiveDocument.HTMLProject

' Create PowerPoint reference:
Dim prjPPT As PowerPoint.HTMLProjectItem
Set prjPPT = ActivePresentation.HTMLProject
' Create Excel reference:
Dim prjXL As Excel.HTMLProjectItem
Set prjXL = ActiveWorkbook.HTMLProject
```

When you have created a reference to the HTMLProject object, you then use the HTMLProjectItems property to access individual HTMLProjectItem objects. In the following example, the IsHTMLProjectDirty procedure can be used to determine if the HTMLProject object in an Office document is "dirty" (contains changes). You use the blnRefreshProject argument to specify whether to refresh, or synchronize, the HTML code with the source Office document.

```
Function IsHTMLProjectDirty(objOffDoc As Object, blnRefreshProject As Boolean) As Boolean
  ' This procedure determines if the HTMLProject object  in the document represented by the objOffDoc argument
  ' is dirty and, if so, refreshes the project according  to the value of the blnRefreshProject argument.
  Dim prjProject As HTMLProject

  On Error GoTo IsHTMLDirty_Err
  Set prjProject = objOffDoc.HTMLProject
  With prjProject
    ' The Office document will be locked as soon as any changes are made to the HTML code in the document.
    If .State = msoHTMLProjectStateDocumentLocked Then
      IsHTMLProjectDirty = True
      If blnRefreshProject = True Then
        ' Merge the changes to the HTML code with the underlying Office document.
        .RefreshDocument
      End If
    Else
      IsHTMLProjectDirty = False
```

```
    End If
  End With
IsHTMLDirty_End:
  Exit Function
IsHTMLDirty_Err:
  Select Case Err
    Case Is > 0
      IsHTMLProjectDirty = False
      Resume IsHTMLDirty_End
  End Select
End Function
```

You could use the preceding procedure to determine the state of any Office document by using the ActiveWorkbook, ActivePresentation, or ActiveDocument property in the first argument.

When you have a reference to an HTMLProjectItem object, you can work directly with the HTML code in the document by using the object's Text property. For example, you can run the following code from the Immediate window to print all of the HTML code in a Word document:

```
? ActiveDocument.HTMLProject.HTMLProjectItems(1).Text
```

You can change the HTML code in an Office document by using the LoadFromFile method or by setting the Text property to the HTML code you want to use. The following example illustrates how to replace the HTML code in a Word document with the HTML code contained in a file on disk:

```
ActiveDocument.HTMLProject.HTMLProjectItems(1).LoadFromFile = "c:\MyHTMLFile.htm"
```

Often, you will want to leave the existing HTML code in a document unchanged, but you will want to insert additional HTML code or script to give the document additional functionality when viewed in a Web browser. In the following example, the AddHTMLAndScriptExample procedure inserts within the first section of a Word document HTML code that includes formatted text, a command button, and script that executes when the command button is clicked. The formatted text and command button are contained in text returned by the GetText procedure and the script that executes when the command button is clicked is returned by the GetScript procedure. The InsertHTMLText procedure inserts the HTML code and script in an existing document just after the location specified by the procedure's second argument.

```
Sub AddHTMLAndScriptExample(objOffDoc As Object)
  Dim itmPrjItem     As HTMLProjectItem
  Dim strNewText     As String
  Dim strNewScript   As String
  Dim strNewHTML     As String

  strNewText = GetText()
  strNewScript = GetScript()
  Set itmPrjItem = objOffDoc.HTMLProject.HTMLProjectItems(1)
  With itmPrjItem
    strNewHTML = .Text
    Call InsertHTMLText(strNewHTML, "<div class=Section1>", strNewText & vbCrLf & strNewScript)
    .Text = strNewHTML
  End With
End Sub
```

Working with Scripts

You can use the Scripts collection and the Script object to programmatically access script, or insert script into a cell or range in a Microsoft® Excel worksheet, a Microsoft® PowerPoint® slide, or a Microsoft® Word document or Word Selection object. In addition, if you use a Microsoft® Office application to open an HTML page, any script contained in that page will be available through the Scripts collection.

Every Script object that is inserted in an Office document includes a Shape object of the type msoScriptAnchor. In Excel and PowerPoint, these shapes are added to the Worksheet or Slide object's Shapes collection. In Word, these shapes are added to a document's InLineShapes collection.

If you want to write script in a document you create in an Office application, use the Microsoft Script Editor. On the other hand, if you want to add script to an Office document programmatically, from an add-in for example, use the objects, properties, and methods of the script object model discussed here.

In This Section

Understanding Script Object Properties
Learn how to access Script objects and add them to documents.

Adding and Removing Script from a Document

Add script to and remove script from a document by using the Scripts collection's Add and Delete methods.

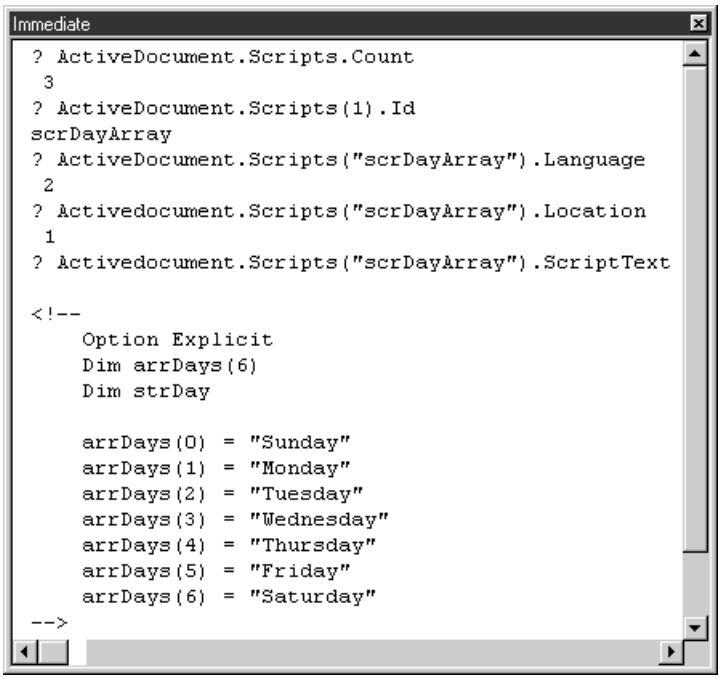
Understanding Script Object Properties

The Scripts collection contains all the Script objects in a Microsoft®Office document. A Script object represents a <SCRIPT> tag pair, its attribute settings, and all the text contained between the tag pair. An Office document or an HTML page can contain several script blocks and each script block can contain any number of procedures. For example, the following HTML code contains three script blocks. The first block initializes an array representing the days of the week, the second block contains a procedure that executes when the page loads, and the third block contains the WriteDay and WriteDate procedures that create a part of the text that is displayed on the page itself.

```
<HTML>
<HEAD>
<TITLE>Developing Office Developer Applications</TITLE>
<SCRIPT LANGUAGE="VBSCRIPT" ID="scrDayArray">
<!--
  Option Explicit
  Dim arrDays(6)
  Dim strDay

  arrDays(0) = "Sunday"
  arrDays(1) = "Monday"
  arrDays(2) = "Tuesday"
  arrDays(3) = "Wednesday"
  arrDays(4) = "Thursday"
  arrDays(5) = "Friday"
  arrDays(6) = "Saturday"
-->
</SCRIPT>

<SCRIPT LANGUAGE="VBSCRIPT" ID="scrShowDay">
<!--
  Option Explicit
  Function ShowDayMessage()
    Dim intDay
    Dim strDayOfWeek

    intDay = WeekDay(Date())
    strDayOfWeek = arrDays(intDay - 1)
    MsgBox "Today is " & strDayOfWeek
    Call WriteDay(strDayOfWeek)
    Call WriteDate()
  End Function
-->
</SCRIPT>
</HEAD>

<BODY ONLOAD="ShowDayMessage()">
<H2>Developer Office Developer VBA and Workflow Solutions:</H2>
<H3>Shared Office Components</H3>
<HR>

<DIV ID=DayText>
<!-- Day and date text is inserted here. -->
</DIV>

<SCRIPT LANGUAGE="VBSCRIPT" ID="scrWriteDay">
<!--
  Option Explicit
  Function WriteDay(strDay)
    DayText.innerText = "Today is " & strDay
  End Function
  Function WriteDate()
    Dim strDay
    Dim strNum

    strDay = Day(Date())
    If Len(strDay) = 2 Then
      If left(strDay, 1) = 1 Then
        strNum = 0
```

```
      Else
        strNum = Right(strDay, 1)
      End If
    Else
      strNum = strDay
    End If
    Select Case CInt(strNum)
      Case 1:       strDay = strDay & "st"
      Case 2:         strDay = strDay & "nd"
      Case 3:         strDay = strDay & "rd"
      Case else:        strDay = strDay & "th"
    End Select
    DayText.innerText = DayText.innerText &  ", the " & strDay & " day of the month."
  End Function
-->
</SCRIPT>
</BODY>
</HTML>
```

Listing of Various Script Properties



You access a Script object within the Scripts collection by using either the index for the object within the collection, or the value of the object's ID attribute. If a <SCRIPT> tag has an ID attribute, the value of that attribute becomes the value of the Script object's Id property. If the <SCRIPT> tag does not have an ID attribute setting, an index value is the only way to locate the Script object within the Scripts collection.

When a document that contains script is opened in an Office application, script blocks are added to the Scripts collection in the order in which they appear in the document. When you add a Script object to a document, the new object is added at the end of the collection of Script objects, regardless of the value of the object's Location property. However, when the document is closed and reopened, that same script will appear in the Scripts collection in the order in which it appears in the document.

Because a Script object's index position within the Scripts collection can change between the time the object is added and the time the document is saved and reopened, it is never a good idea to try to locate a specific Script object within the Scripts collection by using its index position in the collection. Instead, you should make a habit of specifying an Id property value when creating a Script object and using that Id property value to locate the Script object within the Scripts collection. It is useful to use the index when you are looping through all the Script objects in a document, however.

Note

If a <SCRIPT> tag in a document uses the same ID attribute as another <SCRIPT> tag, the Scripts collection will contain only the first Script object that uses the duplicate ID attribute. Any other tags that use the same ID attribute will not be included in the collection.

The Script object's Location property returns a long integer representing an msoScriptLocation constant that specifies whether the script is located in the <HEAD> element or the <BODY> element of the HTML code. If you do not specify a value for this argument, the default location is within the <BODY> element. Similarly, the Language property returns a long integer representing an msoScriptLanguage constant that specifies the LANGUAGE attribute of the <SCRIPT> tag. If you do not

specify a value, the default is Microsoft® Visual Basic® Scripting Edition (VBScript), except in Microsoft® Access, where it is Microsoft® JScript®.

Note
The Script object's ScriptText property returns everything between the <SCRIPT> tags, but not the <SCRIPT> tags themselves. You must account for this when you are using the Script object's Add method to programmatically add your own script to a document.

Adding and Removing Script from a Document

You add script to a document by using the Scripts collection's Add method. The Add method uses optional arguments that make it possible for you specify the script's location, language, ID attribute, additional <SCRIPT> tag attributes, and the script to be contained within the <SCRIPT> tags. The Add method also automatically generates HTML comment tags (<!- and ->) around your script, so browsers that do not recognize script can ignore it. If you use the Add method without specifying any of the method's arguments, you create an empty <SCRIPT> tag pair that looks similar to this:

```
<SCRIPT ID="" LANGUAGE="VBScript">
<!--….->
</SCRIPT>
```

You use the Anchor argument of the Add method to specify where the Script object should be located in a document.

- In Microsoft® Word, if you do not specify a value for the Anchor argument, the Script object is inserted at the current location of the insertion point (cursor). In addition, you can add a Script object to a Selection object or an InLineShape object. When you specify a Selection object in the Anchor argument, the Script object is inserted at the end of the Selection object. When you specify an InLineShape object in the Anchor argument, the Script object is inserted before the paragraph marker for the paragraph that is the anchor point for the shape. You cannot add a Script object to a Range object in Word.
- In Microsoft® Excel, if you do not specify a value for the Anchor argument, the Script object is inserted in the currently active cell. You can also add a Script object to a Range object by specifying the Range object in the Anchor argument. You cannot add a Script object to a Shape object in Excel. If you do specify a Shape object in the Anchor argument, the argument is ignored and the Script object is inserted in the currently active cell.
- In Microsoft® PowerPoint®, you can only add a Script object to a Slide object. If you specify a Shape object in the Anchor argument, the argument is ignored and the Script is inserted in the specified Slide object.

Note
To see the Shape objects that are inserted when script is added to an Office document, point to Macro on the Tools menu, and then click Show All Script. The Show All Script is not on the menu by default; you must add it from the Customize dialog box on the Tools menu.

The following procedure uses the Scripts collection's Add method to add a <SCRIPT> tag pair and some Microsoft® Visual Basic ® Scripting Edition (VBScript) code to the <HEAD> element of the current Word document:

```
Function AddScriptToDocumentDemo()
  ' This procedure illustrates how to use the Scripts collection  to add VBScript code to an Office document.
  Dim strScriptCode As String
  Dim scrArrayScript As Script
    On Error Resume Next
  Set scrArrayScript = ActiveDocument.Scripts("scrDayArray")
If Err = 0 Then  Exit Function                    ' The script is already in the document so no need to add it again.
  strScriptCode = vbTab & "Option Explicit" & vbCrLf & vbTab _
    & "Dim arrDays(6)" & vbCrLf & vbTab _
    & "arrDays(0) = " & """Sunday""" & vbCrLf & vbTab _
    & "arrDays(1) = " & """Monday""" & vbCrLf & vbTab _
    & "arrDays(2) = " & """Tuesday""" & vbCrLf & vbTab _
    & "arrDays(3) = " & """Wednesday""" & vbCrLf & vbTab _
    & "arrDays(4) = " & """Thursday""" & vbCrLf & vbTab _
    & "arrDays(5) = " & """Friday""" & vbCrLf & vbTab _
    & "arrDays(6) = " & """Saturday"""
With Application.ActiveDocument
    .Scripts.Add Location:=msoScriptLocationInHead, Language:=msoScriptLanguageVisualBasic, ID:="scrDayArray", _
      ScriptText:=strScriptCode
End With
End Function
```

You can remove all the script and <SCRIPT> tags from a document by using the Scripts collection's Delete method. You remove a single script from the Scripts collection by using the Script object's Delete method.

4. GETTING THE MOST OUT OF VISUAL BASIC FOR APPLICATIONS

As a developer, your goal is to write code that's fast, efficient, easy to read and maintain, and if possible, reusable. To do so, you must have a solid working knowledge of Microsoft® Visual Basic® for Applications (VBA)-what features the language includes and what you can do with it.

As you develop applications, you'll find that there are a number of operations that you must perform repeatedly-parsing a file path, for example, or returning all the files in a directory. Rather than rewriting these routines every time you require them, you can begin building an arsenal of procedures that solve common problems. This section gives you a head start by providing functions that perform some often-required operations on strings, numbers, dates and times, files, and arrays. It also explains the key aspects of each procedure and covers fundamental VBA programming issues so that you can continue to expand your code arsenal yourself. You can use these procedures not only in VBA code but also in Visual Basic Scripting Edition (VBScript) code in HTML documents.

In This Section

Working with Strings
Understand how to get information from strings.

Working with Numbers
Learn how to use numeric values and data types in Microsoft® Visual Basic® for Applications (VBA).

Working with Dates and Times
Manipulate date values in Microsoft®Visual Basic® for Applications (VBA), and understand how VBA stores date values internally.

Working with Files
Understand the Microsoft Scripting Runtime object library, and work with drives, folders, and files as objects.

Understanding Arrays
Use arrays when you must store a number of values of the same type, but you don't want to create individual variables to store them all.

Tips for Defining Procedures in VBA
Define a Function or Sub procedure, and use the options available to you to make your code more extensible or more flexible.

Optimizing VBA Code
Understand how to streamline your Microsoft® Visual Basi® for Applications (VBA) code to streamline your memory requirements.

Working with Strings

Data structures composed of a sequence of alphanumeric characters, strings are basic in concept, but getting the information you require from them can be a different story.

In This Section

Comparing Strings
Compare strings to determine whether they contain equivalent characters and how they differ if they do not match.

Calculating String Length
Determine the length of a string using the Len function to parse its contents.

Searching a String
Search strings to find out whether they contain a particular character or group of characters by using the InStr or InStrRev functions.

Returning Portions of a String
Learn how to parse the string to be able to work with part of a string's contents.

Working with Strings as Arrays
Quickly turn once-lengthy string-manipulation procedures into just a few lines of code.

Finding and Replacing Text Within a String
Find and replace all occurrences of a substring within a string using the Replace function.

Converting Text in a String from One Case to Another
Use the Microsoft® Visual Basic® for Applications (VBA) StrConv function to convert text in a string from one case to another.

Working with String Variables
Understand how to dimension a variable, assign a value to it, and output that variable as part of a string.

Comparing Strings

You can compare strings to determine whether they contain equivalent characters and how they differ if they do not match. When you compare two strings, you're actually comparing the ANSI value of each character to the value of the corresponding character in the other string. You can specify whether you want to make comparisons case-sensitive or whether you want to ignore the case and simply compare the strings' characters.

Specifying the String-Comparison Setting for a Module

The Option Compare statement determines how strings are compared within a module. There are three settings for the Option Compare statement:

- Option Compare Binary Strng comparisons are case-sensitive. Option Compare Binary is the default string-comparison setting for all the Microsoft® Office applications except Microsoft® Access, for which Option Compare Database is the default.
- Option Compare Text Strng comparisons are case-insensitive. To make case-insensitive string comparison the default method for a module, add this statement to the module's Declarations section.
- Option Compare Database Strng comparisons depend on the sort order for the specified locale; the default sort order is case-insensitive. The Option Compare Database setting is available only for Access databases. Note that when you create a new module in Access, the Option Compare Database statement is automatically inserted in the module's Declarations section. If you delete the Option Compare Database statement, the default string-comparison setting for the module is Option Compare Binary.

Note
If you're writing code in Access that you might want to export to another Microsoft® Visual Basic® for Applications (VBA) host application, you should explicitly specify string comparisons as binary or text-based in the line that performs the comparison. Because the Option Compare Database setting is available only in Access, the code will not compile when you import it into another application unless you remove this setting. If you have explicitly specified the string-comparison method for each line that performs comparisons, you can export the code and be confident that string comparisons will continue to work as expected after you remove the Option Compare Database setting.

Tip
To change the sort order for a database, click Options on the Tools menu, click the General tab, and then change the New Database Sort Order setting. After you change this setting, any new database you create will perform text comparisons based on the new sort order; changing this option has no effect on existing databases.

Comparing Strings by Using Comparison Operators

Because you're actually comparing ANSI values when you compare two strings, you can use the same comparison operators that you would use with numeric expressions-greater than (>), less than (<), equal to (=), and so on. In addition to these numeric comparison operators, you can use the Like operator, which is specifically for use in comparing strings, including strings that contain wildcard characters.

Using Comparison Operators

When you use comparison operators such as the greater than (>) and less than (<) operators to compare two strings, the result you get depends on the string-comparison setting for the module. Consider the following example:

"vba" > "VBA"

If the string-comparison setting is Option Compare Binary, the comparison returns True.

When Microsoft® Visual Basic® for Applications (VBA) performs a binary text comparison, it compares the binary values for each corresponding position in the string until it finds two that differ. In this example, the lowercase letter "v" corresponds to the ANSI value 118, while the uppercase letter "V" corresponds to the ANSI value 86. Because 118 is greater than 86, the comparison returns True.

If the string-comparison setting is Option Compare Text, "vba" > "VBA" returns False, because the strings are equivalent apart from case.

In a Microsoft® Access database, if the string-comparison setting is Option Compare Database and the New Database Sort Order option is set to General (the default setting), the string comparison is case-insensitive and the example returns False.

Using the Like Operator

You can perform wildcard string comparisons by using the Like operator. The following table shows the wildcard characters supported by VBA.

Wildcard	Represents	Example
*	Any number of characters	t* matches any word beginning with "t."
?	Any single character	t??t matches any four-letter word beginning and ending with "t."

#	Any single digit (0-9)	1#3 matches any three-digit number beginning with "1" and ending with "3."
[charlist]	Any single character in charlist	[a-z] matches any letter that falls between "a" and "z" (case-sensitivity depends on Option Compare setting).
[!charlist]	Any single character not in charlist	[!A-Z] excludes the uppercase alphabetic characters (case-sensitivity depends on Option Compare setting).

You can use the Like operator to perform data validation or wildcard searches. For example, suppose you want to ensure that a user has entered a telephone number in the format nnn-nnn-nnnn. You can use the Like operator to check that the entry is valid, as the following procedure does:

```
Function ValidPhone(strPhone As String) As Boolean
   ' This procedure checks that the passed-in value is  a valid, properly formatted telephone number.
   ValidPhone = strPhone Like "###-###-####"
End Function
```

This procedure compares characters in a string to make sure that certain positions contain numeric characters. To return True, all characters must be digits between 0 and 9 or hyphens, and the hyphens must be present at the correct position in the string.

Overriding the Default String-Comparison Setting

To perform a string comparison within a procedure and override the string-comparison setting for the module, you can use the StrComp function. The StrComp function takes two strings as arguments, along with a compare argument, which you can use to specify the type of comparison. The possible settings for the compare argument are vbBinaryCompare, vbTextCompare, and (in Microsoft® Access) vbDatabaseCompare. If you omit this argument, the StrComp function uses the module's default comparison method.

The following table lists the possible return values for the StrComp function.

If	Then StrComp returns
string1 < string2	-1
string1 = string2	0
string1 > string2	1
string1 Or string2 Is Null	Null

For example, running the following code from the Immediate window prints "1", indicating that the ANSI value of the first character in the first string is greater than the ANSI value of the first character in the second string:

```
? StrComp("vba", "VBA", vbBinaryCompare)
```

On the other hand, if you specify text-based string comparison, this code prints "0", indicating that the two strings are identical:

```
? StrComp("vba", "VBA", vbTextCompare)
```

Other Microsoft® Visual Basic® for Applications (VBA) string functions that perform string comparison also provide a compare argument that you can use to override the default string-comparison setting for that function call. For example, the InStr and InStrRev functions both have a compare argument.

Calculating String Length

Often you must know the length of a string to parse its contents. You can use the Len function to calculate the length of a string:

```
Dim lngLen As Long
lngLen = Len(strText)
```

When Microsoft® Visual Basic® for Applications (VBA) stores a string in memory, it always stores the length of the string in a long integer at the beginning of the string. The Len function retrieves this value and is therefore quite fast.

The Len function is useful when you must determine whether a string is a zero-length string (""). Rather than comparing the string in question to a zero-length string to determine whether they're equivalent, you can simply check whether the length of the string is equal to 0. For example:

```
If Len(strText) > 0 Then
   ' Perform some operation here.
End If
```

Searching a String

When you must know whether a string contains a particular character or group of characters, you can search the string by using one of two functions. The traditional candidate for this job is the InStr function, which you can use to find one string within another. The InStr function compares two strings, and if the second string is contained within the first, it returns the position at which the substring begins. If the InStr function doesn't find the substring, it returns 0.

The InStr function takes an optional argument, the start argument, in which you can specify the position to begin searching. If you omit this argument, the InStr function starts searching at the first character in the string.

The newest version of Microsoft® Visual Basic® for Applications (VBA) includes a function called InStrRev, which behaves in the same way as the InStr function, except that it begins searching at the end of the string rather than at the beginning. As with the InStr function, you can specify a starting position for the InStrRev function; it will search backward through the string beginning at that point. If you know that the substring you're looking for probably falls at the end of the string, the InStrRev function might be a better option. For example, the InStrRev function makes it easier to parse a file path and return just the file name.

Note
Both the InStr and InStrRev functions return the same value when they locate the same substring. Although the InStrRev function begins searching at the right side of the string, it counts characters from the left side, as does the InStr function. For example, calling either the InStr or InStrRev function to search the string "C:\Temp" for the substring "C:\" returns 1. However, if the substring appears more than when, and you haven't specified a value for the start argument, the InStr function returns the position of the first instance and the InStrRev function returns the position of the last instance.

The following procedure counts the occurrences of a particular character or group of characters in a string. To call the procedure, you pass in the string, the substring that you're looking for, and a constant indicating whether the search should be case-sensitive. The CountOccurrences procedure uses the InStr function to search for the specified text and return the value of the position at which it first occurs; for example, if it's the third character in the string, the InStr function returns 3. The procedure increments the counter variable, which keeps track of the number of occurrences found, and then sets the starting position for the next call to the InStr function. The new starting position is the position at which the search text was found, plus the length of the search string. By setting the start position in this manner, you ensure that you don't locate the same substring twice when you're searching for text that's more than one character in length.

The possible constant values are specified by the built-in enumerated constants in vbCompareMethod, which groups the three VBA string-comparison constants (vbBinaryCompare, vbDatabaseCompare, and vbTextCompare). If you declare an argument as type vbCompareMethod, VBA lists the constants in that grouping when you call the procedure. This is a convenient way to remember what values an argument takes. In addition, you can define your own enumerated constants and use them as data types.

```
Function CountOccurrences(strText As String, strFind As String,  Optional lngCompare As VbCompareMethod) As Long
  ' Count occurrences of a particular character or characters.  If lngCompare argument is omitted, procedure performs binary comparison.
  Dim lngPos      As Long
  Dim lngTemp     As Long
  Dim lngCount    As Long
   ' Specify a starting position. We don't need it the first time through the loop, but we'll need it on subsequent passes.
  lngPos = 1
  ' Execute the loop at least once.
  Do
    ' Store position at which strFind first occurs.
    lngPos = InStr(lngPos, strText, strFind, lngCompare)
    ' Store position in a temporary variable.
    lngTemp = lngPos
    ' Check that strFind has been found.
    If lngPos > 0 Then
      ' Increment counter variable.
      lngCount = lngCount + 1
      ' Define a new starting position.
      lngPos = lngPos + Len(strFind)
    End If
  ' Loop until last occurrence has been found.
  Loop Until lngPos = 0
  ' Return the number of occurrences found.
  CountOccurrences = lngCount
End Function
```

Calling this function from the Immediate window as follows returns "3":

```
? CountOccurrences("This is a test", "t", vbTextCompare)
```

Returning Portions of a String

To work with part of a string's contents, you must parse the string. You can use the InStr or InStrRev function to find the position at which to begin parsing the string. When you've located that position, you can use the Left, Right, and Mid functions to do the job. The Left and Right functions return a specified number of characters from either the left or right portion of the string. The Mid function is the most flexible of the parsing functions-you can specify a starting point anywhere within the string, followed by the number of characters you want to return.

Note
Some of the Microsoft® Visual Basic® for Applications (VBA) string functions come in two varieties, one that returns a string, and one that returns a string-type Variant value. The names of the functions that return a string include a dollar sign ("$"); for example, Chr$, Format$, LCase$, Left$, LTrim$, Mid$, Right$, RTrim$, Space$, Trim$, and UCase$. The functions that return a string-type Variant value have no dollar sign; for example, Chr, Format, LCase, Left, LTrim, Mid, Right, RTrim, Space, Trim, and UCase. The string-returning functions are faster; however, you'll get an error if you call them with a value that is Null. The functions that return a string-type Variant value handle Null values without an error. Code examples in this section use the string-returning functions where appropriate.

The following procedure parses a file path and returns one of the following portions: the path (everything but the file name), the file name, the drive letter, or the file extension. You specify which part of the string you want to return by passing a constant to the lngPart argument. The lngPart argument is defined as type opgParsePath, which contains custom enumerated constants declared in the modPublicDefs module in VBA.mdb.

Note that this procedure uses the InStrRev function to find the last path separator, or backslash (\), in the string. If you used the InStr function, you'd have to write a loop to make sure that you'd found the last one. With the InStrRev function, you know that the first backslash you find is actually the last one in the string, and the characters to the right of it must be the file name.

```
Function ParsePath(strPath As String, lngPart As opgParsePath) As String
  ' This procedure takes a file path and returns  the path (everything but the file name), the drive letter, or the file extension, depending on which constant was passed in.
  Dim lngPos          As Long
  Dim strPart         As String
  Dim blnIncludesFile   As Boolean
    ' Check that this is a file path. Find the last path separator.
  lngPos = InStrRev(strPath, "\")
  ' Determine whether portion of string after last backslash contains a period.
  blnIncludesFile = InStrRev(strPath, ".") > lngPos
    If lngPos > 0 Then
    Select Case lngPart
       Case opgParsePath.FILE_ONLY         ' Return file name.
         If blnIncludesFile Then
           strPart = Right$(strPath, Len(strPath) - lngPos)
         Else
           strPart = ""
         End If
       Case opgParsePath.PATH_ONLY         ' Return path.
         If blnIncludesFile Then
           strPart = Left$(strPath, lngPos)
         Else
           strPart = strPath
         End If
       Case opgParsePath.DRIVE_ONLY        ' Return drive.
         strPart = Left$(strPath, 3)
       Case opgParsePath.FILEEXT_ONLY    ' Return file extension.
         If blnIncludesFile Then
           ' Take three characters after period.
           strPart = Mid(strPath, InStrRev(strPath, ".") + 1, 3)
         Else
           strPart = ""
         End If
       Case Else
         strPart = ""
     End Select
  End If
  ParsePath = strPart
ParsePath_End:
  Exit Function
End Function
```

Calling this function as follows from the Immediate window returns "Test.txt":

```
? ParsePath("C:\Temp\Test.txt", opgParsePath.FILE_ONLY)
```

Working with Strings as Arrays

It might be hard to believe, but some of the most exciting features in Microsoft® Visual Basic®for Applications (VBA) in Microsoft® Office XP are the functions for working with strings as arrays. These functions can turn once-lengthy string-manipulation procedures into just a few lines of code. And in many cases, they're faster than using loops and string-parsing techniques to work with the contents of a very large string.

The Split Function

The Split function takes a string and converts it into an array of strings. By default, it divides the string into elements by using the space character as a delimiter, so that if you pass in a sentence, each element of the array contains a word. For example, if you pass this string to the Split function

"This is a test"

you'll get an array that contains the following four elements:

"This"
"is"
"a"
"test"

You can specify that the Split function split the string based on a different delimiter by passing in the delimiter argument.

When you've split a string into an array, it's easy to work with the individual elements. The Split function sizes the array for you, so you don't have to worry about maintaining the array's size.

The following example uses the Split function to count the number of words in a string. The procedure takes a string and returns a long integer indicating the number of words found. Because the string is divided into elements at the space between each word, each element of the resulting array represents a word. To determine the number of words, you simply must determine the number of elements in the array. You can do this by subtracting the lower bound from the upper bound and adding 1.

```
Function CountWords(strText As String) As Long
  ' This procedure counts the number of words in a string.
  Dim astrWords() As String
  astrWords = Split(strText)
  ' Count number of elements in array -- this will be the number of words.
  CountWords = UBound(astrWords) - LBound(astrWords) + 1
End Function
```

The Join Function

After you've finished processing an array that's been split, you can use the Join function to concatenate the elements of the array together into a single string again. The Join function takes an array of strings and returns a concatenated string. By default it adds a space between each element of the string, but you can specify a different delimiter.

The following procedure uses the Split and Join functions together to trim extra space characters from a string. It splits the passed-in string into an array. Wherever there is more than one space within the string, the corresponding array element is a zero-length string. By finding and removing these zero-length string elements, you can remove the extra white space from the string.

To remove zero-length string elements from the array, the procedure must copy the non-zero-length string elements into a second array. The procedure then uses the Join function to concatenate the second array into a whole string.

Because the second array isn't created by the Split function, you must size it manually. It's easy to do, however-you can size it initially to be the same size as the first array, then resize it after you've copied in the non-zero-length string elements.

```
Function TrimSpace(strInput As String) As String
  ' This procedure trims extra space from any part of  a string.
  Dim astrInput()    As String
  Dim astrText()     As String
  Dim strElement     As String
  Dim lngCount       As Long
  Dim lngIncr        As Long
  astrInput = Split(strInput)                 ' Split passed-in string.
  ReDim astrText(UBound(astrInput))           ' Resize second array to be same size.
  lngIncr = LBound(astrInput)                 ' Initialize counter variable for second array.
' Loop through split array, looking for non-zero-length strings.
  For lngCount = LBound(astrInput) To UBound(astrInput)
    strElement = astrInput(lngCount)
    If Len(strElement) > 0 Then
      astrText(lngIncr) = strElement          ' Store in second array.
```

```
        lngIncr = lngIncr + 1
      End If
    Next
' Resize new array.
    ReDim Preserve astrText(LBound(astrText) To lngIncr - 1)
    TrimSpace = Join(astrText)                    ' Join new array to return string.
End Function
```

To test the TrimSpace procedure, try calling it from the Immediate window with a string such as the following:

```
? TrimSpace("  This  is  a  test  ")
```

Tip

To see the elements in each array while the code is running, step through the procedure, and use the Locals window to view the values contained in each variable.

The Filter Function

The Filter function searches a string array for all elements that match a given text string. The Filter function takes three arguments: a string array, a string containing the text to find, and a constant specifying the string-comparison method. It returns a string array containing all the matches that it finds.

You can use the Filter function to determine whether a particular element exists in an array. An example, the ConvertToProperCase procedure, appears in Converting Strings.

When working with the Filter function, you might notice that it returns a particular element even if only part of the element matches the search text. In other words, if your search text is the letter "e," and the array you're searching contains the element "test," the array returned by the Filter function will contain the element "test."

Given this behavior, you might be tempted to use the Filter function to rewrite the CountOccurrences procedure shown earlier in this section. Before doing so, bear in mind that the CountOccurrences procedure counts every occurrence of a particular character in a string, even if there is more than one occurrence in a word. When you are using the Filter function, on the other hand, you can count an occurrence only once per element, even if the character occurs twice within a single element in the array.

Replacing Text Within a String

Microsoft® Visual Basic® for Applications (VBA) provides another function, the Replace function, which makes it easy to find and replace all occurrences of a substring within a string. The Replace function takes up to six arguments: the string to be searched, the text to find within the string, the replacement text, what character to start at, how many occurrences to replace, and a constant indicating the string-comparison method. You don't even have to write a loop to use the Replace function-it automatically replaces all the appropriate text for you with one call.

For example, suppose you want to change the criteria for an SQL statement based on some condition in your application. Rather than re-creating the SQL statement, you can use the Replace function to replace just the criteria portion of the string, as in the following code fragment:

```
strSQL = "SELECT * FROM Products WHERE ProductName Like 'M*' ORDER BY ProductName;"
strFind = "'M*'"
strReplace = "'T*'"
Debug.Print Replace(strSQL, strFind, strReplace)
```

Running this code fragment prints this string to the Immediate window:

```
SELECT * FROM Products WHERE ProductName Like 'T*' ORDER BY ProductName;
```

Wildcard Search and Replace

The Replace function greatly simplifies string search-and-replace operations, but it doesn't make it possible for you to perform wildcard searches. Here's another place where the Split and Join functions come in handy.

The ReplaceWord procedure shown below takes three mandatory arguments: a string to be searched, the word to find within the string, and the replacement text. When you call this procedure, you can include wildcard characters in the string that you pass for the strFind argument. For example, you might call the ReplaceWord procedure from the Immediate window with these parameters:

```
? ReplaceWord("There will be a test today", "t*t", "party")
```

The procedure splits the strText argument into an array, then uses the Like operator to compare each element of the array to strFind, replacing the elements that match the wildcard specification.

```
Function ReplaceWord(strText As String, strFind As String, strReplace As String) As String
  ' This function searches a string for a word and replaces it. You can use a wildcard mask to specify the search string.
  Dim astrText()   As String
  Dim lngCount     As Long

   astrText = Split(strText)              ' Split the string at specified delimiter.
  ' Loop through array, performing comparison against wildcard mask.
```

```
  For lngCount = LBound(astrText) To UBound(astrText)
    If astrText(lngCount) Like strFind Then
      ' If array element satisfies wildcard search, replace it.
      astrText(lngCount) = strReplace
    End If
  Next
  ReplaceWord = Join(astrText)      ' Join string, using same delimiter.
End Function
```

Converting Text in a String from One Case to Another

To convert text in a string from one case to another, you can use the Microsoft® Visual Basic®for Applications (VBA) StrConv function. The StrConv function converts a string to lowercase, uppercase, or proper case (initial capital letters). It takes a string and a constant that specifies how to convert the string. For example, the following code fragment converts a string to proper case:

```
Debug.Print StrConv("washington, oregon, and california", vbProperCase)
```

Running this code prints the following text to the Immediate window:

Washington, Oregon, And California

Note

The StrConv function performs other string conversions as well. For example, it converts a string from Unicode to ANSI, or vice versa. For more information about the StrConv function, search the Visual Basic Reference Help index for "StrConv function."

Most likely, you'll be about three-fourths satisfied with this result-you probably want "washington," "oregon," and "california" to be capitalized, but not "and." The word "and" is a minor word that isn't capitalized according to grammatical convention, unless it's the first word in the sentence. Unfortunately, VBA doesn't know which words to convert and which to leave alone, so it converts everything. You must manually write code to handle the cases you don't want capitalized.

If you want VBA to omit the minor words, you can define those words in a file or a table, and perform a comparison against the file or table when you convert each word. The following procedure, ConvertToProperCase, does just that-it takes a string, splits it into individual words, compares each word against a list in a text file, and converts all non-minor words to proper case.

The ConvertToProperCase procedure calls another procedure, the GetMinorWords procedure. This procedure reads a text file containing a list of minor words and returns an array of strings containing each word in the text file. The ConvertToProperCase procedure then uses the Filter function to compare each word in the string to be converted against the list of words contained in the array of minor words. If a word doesn't appear in the list, then it's converted to proper case. If it does appear, it's converted to lowercase.

```
Function ConvertToProperCase(strText As String) As String
  ' This function takes a string and converts it to proper  case, except for any minor words.
  Dim astrText()      As String
  Dim astrWords()     As String
  Dim astrMatches()   As String
  Dim lngCount        As Long
    astrWords = GetMinorWords                ` ' Return array containing minor words.
  astrText = Split(strText)                    ' Split string into array.
  ' Check each word in passed-in string against array  of minor words.
  For lngCount = LBound(astrText) To UBound(astrText)
    ' Filter function returns array containing matches found. If no matches are found, upper bound of array is less than
    ' lower bound. Store result returned by Filter function in a String array, then compare upper bound with lower bound.
    astrMatches = Filter(astrWords, astrText(lngCount))
    If UBound(astrMatches) < Lbound(astrMatches) Then
      ' If word in string does not match any word in array of minor words, convert word to proper case.
      astrText(lngCount) = StrConv(astrText(lngCount), vbProperCase)
    Else
      ' If it does match, convert it to lowercase.
      astrText(lngCount) = StrConv(astrText(lngCount), vbLowerCase)
    End If
  Next
  ConvertToProperCase = Join(astrText)      ' Join the string.
End Function
```

The ConvertToProperCase procedure calls the GetMinorWords procedure, which opens the text file that contains the list of minor words, gets a string containing all the words in the list, splits the string into an array, and returns the array. GetMinorWords calls another procedure, the GetLikelyDelimiter procedure, which finds the first likely delimiter character in the text file.

Note

To call the ConvertToProperCase procedure, you must set a reference to the Microsoft Scripting Runtime object library.

Working with String Variables

Almost any application uses strings that contain variables in some form or another; you dimension a variable, assign a value to it, and output that variable as part of a string. If you must output a string that contains multiple variables, it can often be a painstaking process adding all of the quote characters and concatenation operators in the right places.

For example, the following code contains several string variables:

```
Dim Fname As String, Lname As String
Dim varAge As Variant, varDate as Variant

Fname = "John"
Lname = "Doe"
varAge = 42
varDate = "August 15"

TextBox1.Text = Fname & Lname & " will be " & varAge &  " od on " & varDate & " of this year."
```

In the preceding example, it would be easy to leave out a quotation mark character or space or to misplace a concatenation character. The result would be a compile-time error, and finding your mistake could prove especially difficult in a long string.

The String Editor add-in, included in Microsoft®Office XP Developer, greatly simplifies the process of formatting complex strings such as SQL statements or scripts. Using the String Editor, you can simply enter your string as straight text, then mark any string variables within the string. On completion, the String Editor will automatically format the string for you, inserting all of the necessary quotes and other formatting characters.

To insert a formatted string into your code

1. Select an insertion point in the Code Editor where you want to add a string.
2. From the **Add-Ins** menu, select **String Editor** .
   Note
   The String Editor menu item is only available when the VBA String Editor add-in is loaded.
3. Type the string into the String Editor. For example:
   Fname Lname will be varAge od on varDate of this year.
4. For each variable within the string, select the variable, and click the **Toggle String** button on the **String Editor** toolbar.Each selection will be marked as a variable and color-coded blue in the String Editor.
5. Click the **Update** button to insert the formatted string into your code with all of the necessary formatting characters.

The following table shows some examples of the formatting applied by the String Editor.

In the String Editor	Resulting code
The dog is happy.	"The dog is happy."
The dog is happy.	"The dog is " & vbNewLine & "happy."
The **strAnimal** is happy.	"The " & strAnimal & " is happy."
The dog is happy.	vbTab & "The dog is happy."
"The dog is happy"	Chr$(34) & "The dog is happy" & Chr$(34)

Working with Numbers

Almost every procedure you write in Microsoft® Visual Basic® for Applications (VBA) uses numeric values in some way. For optimal performance and efficiency, and for accuracy in calculations, it is important to understand the different numeric data types and when to use which.

In This Section

The Integer, Long, and Byte Data Types
Understand the three data types in Microsoft® Visual Basic® for Applications (VBA) that can represent integers-the Integer, Long, and Byte data types.

The Boolean Data Type
Use the Boolean data type to specify True or False.

The Floating-Point Data Types
Specify extremely small or large numbers using the Single and Double data types.

The Currency and Decimal Data Types
Use these scaled integer data types when you cannot afford rounding errors and you do not require as many decimal places as the floating-point data types provide.

Conversion, Rounding, and Truncation
Learn about the functions that help you covert, round, and truncate decimals.

Formatting Numeric Values
Format numbers using the following Microsoft® Visual Basic® for Applications (VBA) functions: FormatNumber, FormatCurrency, FormatPercent, and Format.

Using the Mod Operator
Determine whether two numbers divide evenly or how close they come to dividing evenly using the Mod operator, which divides two numbers and returns the remainder.

Performing Calculations on Numeric Arrays
Understand how to perform mathematical functions on a variable set of numbers.

The Integer, Long, and Byte Data Types

Three data types in Microsoft® Visual Basic® for Applications (VBA) can represent integers, or whole numbers: the Integer, Long, and Byte data types. Of these, the Integer and Long types are the ones you are most likely to use regularly.

The Integer and Long data types can both hold positive or negative values. The difference between them is their size: Integer variables can hold values between -32,768 and 32,767, while Long variables can range from -2,147,483,648 to 2,147,483,647. Traditionally, VBA programmers have used integers to hold small numbers, because they required less memory. In recent versions, however, VBA converts all integer values to type Long, even if they are declared as type Integer. Therefore, there is no longer a performance advantage to using Integer variables; in fact, Long variables might be slightly faster because VBA does not have to convert them.

The Byte data type can hold positive values from 0 to 255. A Byte variable requires only a single byte of memory, so it is very efficient. You can use a Byte variable to hold an Integer value if you know that value will never be greater than 255. However, the Byte data type is typically used for working with strings. For some string operations, converting the string to an array of bytes can significantly enhance performance.

The Boolean Data Type

The Boolean data type is a special case of an integer data type. The Boolean data type can contain True or False; internally, Microsoft® Visual Basic® for Applications (VBA) stores the value of True as -1, and the value of False as 0.

You can use the CBool function to convert any numeric value to a Boolean value. When another numeric data type is converted to a Boolean value, any nonzero value is equivalent to True, and zero (0) is equivalent to False. For example, CBool(7) returns True, and CBool(5 + 2 - 7) returns False, because it evaluates to CBool(0).

The following procedure determines whether a number is even. The procedure uses the Mod operator to determine whether a number can be divided by 2 with no remainder. If a number is even, dividing by 2 leaves no remainder; if it is odd, dividing by 2 leaves a remainder of 1:

```
Function IsEven(lngNum As Long) As Boolean
  ' Determines whether a number is even or odd.
  If lngNum Mod 2 = 0 Then
    IsEven = True
  Else
    IsEven = False
  End If
End Function
```

Another way to write this procedure is to convert the result of an expression to a Boolean value and then use the Not keyword to toggle its value, as shown in the following example. If the lngNum argument is odd, then it must be nonzero; converting lngNum to a Boolean value yields True. Because the procedure must return False if the value is odd, using the Not keyword to toggle the Boolean value gives the correct result.

```
Function IsEven(lngNum As Long) As Boolean
  ' Determines whether a number is even or odd.
  IsEven = Not CBool(lngNum Mod 2)
End Function
```

Note that the revised IsEven procedure condenses a five-line If...Then statement into a single line of code. If you are using an If...Then statement to set a value to True under one condition and to False under another, as the IsEven procedure does, you can condense the If...Then statement by modifying its condition to return True or False. However, the revised procedure might be somewhat harder to understand.

The Floating-Point Data Types

Microsoft® Visual Basic®for Applications (VBA) provides two floating-point data types, Single and Double. The Single data type requires 4 bytes of memory and can store negative values between -3.402823 x 1038 and -1.401298 x 10-45 and positive values between 1.401298 x 10-45 and 3.402823 x 1038. The Double data type requires 8 bytes of memory and can store negative values between -1.79769313486232 x 10308 and -4.94065645841247 x 10-324 and positive values between 4.94065645841247 x 10-324 and 1.79769313486232 x 10308.

The Single and Double data types are very precise-that is, they make it possible for you to specify extremely small or large numbers. However, these data types are not very accurate because they use floating-point mathematics. Floating-point mathematics has an inherent limitation in that it uses binary digits to represent decimals. Not all the numbers within the range available to the Single or Double data type can be represented exactly in binary form, so they are rounded. Also, some numbers cannot be represented exactly with any finite number of digits-pi, for example, or the decimal resulting from 1/3.

Because of these limitations to floating-point mathematics, you might encounter rounding errors when you perform operations on floating-point numbers. Compared to the size of the value you are working with, the rounding error will be very small. If you do not require absolute accuracy and can afford relatively small rounding errors, the floating-point data types are ideal for representing very small or very large values. On the other hand, if your values must be accurate-for example, if you are working with money values-you should consider one of the scaled integer data types.

The Currency and Decimal Data Types

The two scaled integer data types, Currency and Decimal, provide a high level of accuracy. These are also referred to as fixed-point data types. They are not as precise as the floating-point data types-that is, they cannot represent numbers as large or as small. However, if you cannot afford rounding errors, and you do not require as many decimal places as the floating-point data types provide, you can use the scaled integer data types. Internally, the scaled integer types represent decimal values as integers by multiplying them by a factor of 10.

The Currency data type uses 8 bytes of memory and can represent numbers with fifteen digits to the left of the decimal point and four to the right, in the range of -922,337,203,685,477.5808 to 922,337,203,685,477.5807.

The Decimal data type uses 12 bytes of memory and can have between 0 and 28 decimal places. The Decimal data type is a Variant subtype; to use the Decimal data type, you must declare a variable of type Variant, and then convert it by using the CDec function.

The following example shows how to convert a Variant variable to a Decimal variable. It also demonstrates how using the Decimal data type can minimize the rounding errors inherent in the floating-point data types.

```
Sub DoubleVsDecimal()
  ' This procedure demonstrates how using the Decimal data type can minimize rounding errors.
  Dim dblNum     As Double
  Dim varNum     As Variant
  Dim lngCount   As Long
  For lngCount = 1 To 100000                ' Increment values in loop.
    dblNum = dblNum + 0.00001
    ' Convert value to Decimal using CDec.
    varNum = varNum + CDec(0.00001)
  Next
  Debug.Print "Result using Double: " & dblNum
  Debug.Print "Result using Decimal: " & varNum
End Sub
```

The procedure prints these results to the Immediate window:

Result using Double: 0.999999999998084

Result using Decimal: 1

A Note About Division

Any time you use the floating-point division operator (/), you are performing floating-point division, and your return value will be of type Double. This is true whether your dividend and divisor are integer, floating-point, or fixed-point values. It is true whether or not your result has a decimal portion.

For example, running the following code from the Immediate window prints "Double":

```
? TypeName(2.34/5.9)
```

So does this code, even though the result is an integer:

```
? TypeName(9/3)
```

Because all floating-point division returns a floating-point value, you cannot be certain that your result is accurate to every decimal place, even if you are performing division on Decimal or Currency values. There will always be an inherent possibility of rounding errors, although they are likely to be small.

If you are dividing integers, or if you do not care about the decimal portion of the result, you can use the integer division operator (\). Integer division is faster than floating-point division, and the result is always an Integer or Long value, either of which requires less memory than a Double value. For example, running this code from the Immediate window prints "Integer":

```
? TypeName(9\3)
```

Conversion, Rounding, and Truncation

When you convert a decimal value to an integer value, Microsoft® Visual Basic® for Applications (VBA) rounds the number to an integer value. How it rounds depends on the value of the digit immediately to the right of the decimal place-digits less than 5 are rounded down, while digits greater than 5 are rounded up. If the digit is 5, then it is rounded down if the digit immediately to the left of the decimal place is even, and up if it is odd. When the digit to be rounded is a 5, the result is always an even integer.

For example, running the following line of code from the Immediate window prints "8," because VBA rounds down when the number immediately to the left of the decimal is even:

? CLng(8.5)

However, this code prints "10," because 9 is odd:

? CLng(9.5)

If you want to discard the decimal portion of a number, and return the integer portion, you can use either the Int or Fix function. These functions simply truncate without rounding. For example, Int(8.5) returns 8, and Int(9.5) returns 9. The Int and Fix functions behave identically unless you are working with negative numbers. The Int function rounds to the lower negative integer, while the Fix function rounds to the higher one.

For example, the following code evaluates to "-8":

? Fix(-8.2)

Using the Int function, on the other hand, yields "-9":

? Int(-8.2)

Note
The Int and Fix functions always return a Double value. You might want to convert the result to a Long value before performing further operations with it.

VBA includes a new rounding function called Round, which you can use to round a floating-point or fixed-point decimal to a specified number of places. For example, the following code rounds the number 1.2345 to 1.234:

? Round(1.2345, 3)

Although the Round function is useful for returning a number with a specified number of decimal places, you cannot always predict how it will round when the rounding digit is a 5. How VBA rounds a number depends on the internal binary representation of that number. If you want to write a rounding function that will round decimal values according to predictable rules, you should write your own.

Formatting Numeric Values

Microsoft®Visual Basic® for Applications (VBA) provides several functions that you can use to format numbers, including the FormatNumber, FormatCurrency, FormatPercent, and Format functions. Each of these functions returns a number formatted as a string.

The FormatNumber function formats a number with the comma as the thousands separator. You can specify the number of decimal places you want to appear. For example, calling the following code from the Immediate window prints "8,012.36":

? FormatNumber(8012.36)

The FormatCurrency function formats a number with a dollar sign, including two decimal places by default. Calling this code from the Immediate window prints "$10,456.45":

? FormatCurrency(10456.45)

The FormatPercent function formats a number as a percentage, including two decimal places by default. For example, calling this code from the Immediate window prints "80.00%":

? FormatPercent(4/5)

If you must have finer control over the formatting of a number, you can use the Format function to specify a custom format. For example, to display leading zeros before a number, you can create a custom format that includes placeholders for each digit. If a digit is absent, a zero appears in that position. The following procedure shows an example that returns a formatted string complete with leading zeros:

```
Function FormatLeadingZeros(lngNum As Long) As String
  ' Formats number with leading zeros.
  FormatLeadingZeros = Format$(lngNum, "00000")
End Function
```

For more information about creating custom formats, search the Visual Basic Reference Help index for "Format function."

Using the Mod Operator

The Mod operator divides two numbers and returns the remainder. It is useful when you must determine whether two numbers divide evenly, or how close they come to dividing evenly. The Mod operator always returns an Integer or Long value, even when you divide floating-point or fixed-point numbers.

For example, the IsFactor procedure takes two arguments, a number and a potential factor, and returns True if the second argument is indeed a factor of the first. The procedure uses the Mod operator to determine whether one value divides evenly into the other.

```
Function IsFactor(lngNum As Long, lngFactor As Long) As Boolean
  ' Determines whether one number is a factor of another number.
  IsFactor = Not CBool(lngNum Mod lngFactor)
End Function
```

Performing Calculations on Numeric Arrays

Many mathematical functions operate on a variable set of numbers. For example, you can take the median, or middle value, of a set of any size. Because you will not know how many numbers the set will contain while you are writing code to find the median, you cannot create a procedure with a set number of arguments. Instead, you can use a dynamic array to store an indeterminate number of values and perform an operation on them.

The following procedure takes a parameter array and returns the median of the values in the array. A parameter array encompasses a variable number of arguments that are passed to a procedure as an array. The ParamArray keyword specifies a parameter array, which must be defined as type Variant.

The Median procedure calls another procedure, IsNumericArray, which determines whether the array contains any non-numeric elements before the Median procedure attempts to find the median. It then calls the QuickSortArray procedure, which sorts the array. Finally, it determines whether the array contains an even or odd number of elements. If the number of elements is odd, the middle element in the sorted array is the median. If the number of elements is even, the median is the average of the two midmost elements.

```
Function Median(ParamArray avarValues() As Variant) As Double
  ' Return the median of a set of numbers.
  Dim lngCount     As Long
  Dim varTemp      As Variant
  varTemp = avarValues()                    ' Store array in temporary variable.
  If IsNumericArray(varTemp) Then           ' Check whether array is numeric.
    ' Determine how many elements are in array.
    lngCount = UBound(varTemp) - LBound(varTemp) + 1
    QuickSortArray varTemp                  ' Sort the array.
    ' Determine whether array contains an odd or even number of elements.
    If IsEven(lngCount) Then
      ' If even, need to find the two middle elements and return the average of their values.
      ' Remember we're working with a zero-based array!
      Median = (varTemp(lngCount / 2 - 1) + varTemp(lngCount / 2)) / 2
    Else
      Median = varTemp(Int(lngCount / 2))   ' If odd, need to find the middle element.
    End If
  Else
    Median = -1                             ' Return -1 if array isn't numeric.
  End If
End Function
```

To test the Median procedure, try calling it with an even set of numbers, then with an odd set of numbers, as follows:

```
? Median(45, 67, 23, 89, 52, 101)
```

To make sure it is working properly, you can check it against the Excel Median worksheet function. Note that the Excel Median function can take no more than 30 arguments, while the procedure shown here can take any number of arguments.

You could also modify this procedure to take an array, rather than a parameter array. The parameter array is somewhat easier to test in isolation, but a procedure that takes an array might be more practical for use within your code. For example, you might have a procedure that fills an array with numeric data from a data source, which you then can pass to the Median procedure to determine the median of the set of numbers, without having to pass each value as an argument to the procedure.

The strategy shown here for finding the median also works for other operations that take an indeterminate number of values, such as finding the average or standard deviation, or performing other statistical calculations.

Working with Dates and Times

Microsoft® Visual Basic® for Applications (VBA) provides a data type for storing date and time values, the Date data type. Convenient as the Date data type is, manipulating date values in VBA can still be tricky. To easily work with dates, you must understand how VBA stores date values internally.

In This Section

The Date Data Type

Microsoft® Visual Basic® for Applications (VBA) provides the Date data type to store date and time values. The Date data type is an 8-byte floating-point value, so internally it is the same as the Double data type. The Date data type can store dates between January 1, 100, and January 1, 9999.

VBA stores the date value in the integer portion of the Date data type, and the time value in the decimal portion. The integer portion represents the number of days since December 30, 1899, which is the starting point for the Date data type. Any dates before this one are stored as negative numbers; all dates after are stored as positive values. If you convert a date value representing December 30, 1899, to a double, you'll find that this date is represented by zero.

The decimal portion of a date represents the amount of time that has passed since midnight. For example, if the decimal portion of a date value is .75, three-quarters of the day has passed, and the time is now 6 P.M.

Because the integer portion of a date represents number of days, you can add and subtract days from one date to get another date.

Getting the Current Date and Time

Three functions in Microsoft® Visual Basic® for Applications (VBA) can tell you exactly when it is: the Now, Date, and Time functions. The Now function returns both the date and time portions of a Date variable. For example, calling the Now function from the Immediate window returns a value such as this one:

2/23/98 6:16:47 PM

The Date function returns the current date. You can use it if you do not have to know the time. The Time function returns the current time, without the date.

Formatting a Date

You can use predefined formats to format a date by calling the FormatDateTime function, or you can create a custom format for a date by using the Format function.

The following procedure formats a date by using both built-in and custom formats:

```
Sub DateFormats(Optional dteDate As Date)
  ' This procedure formats a date using both built-in and custom formats.
    ' If dteDate argument has not been passed, then dteDate is initialized to 0 (or December 30, 1899, the date equivalent of 0).
  If CLng(dteDate) = 0 Then
     dteDate = Now                    ' Use today's date.
  End If
  ' Print date in built-in and custom formats.
  Debug.Print FormatDateTime(dteDate, vbGeneralDate)
```

```
  Debug.Print FormatDateTime(dteDate, vbLongDate)
  Debug.Print FormatDateTime(dteDate, vbShortDate)
  Debug.Print FormatDateTime(dteDate, vbLongTime)
  Debug.Print FormatDateTime(dteDate, vbShortTime)
  Debug.Print Format$(dteDate, "ddd, mmm d, yyyy")
  Debug.Print Format$(dteDate, "mmm d, H:MM am/pm")
End Sub
```

Date Delimiters

When you work with date literals in your code, you must indicate to Microsoft® Visual Basic® for Applications (VBA) that a value is a date. If you do not, VBA might think you are performing subtraction or floating-point division.

For example, if you run the following fragment, the value that VBA assigns to the Date variable is not April 5, 1998, but 4 divided by 5 divided by 98. Because you are assigning it to a Date variable, VBA converts the number to a date, and prints "12:11:45 AM" to the Immediate window:

```
Dim dteDate As Date
dteDate = 4 / 5 / 98
Debug.Print dteDate
```

To avoid this problem, you must include delimiters around the date. The preferred date delimiter for VBA is the number sign (#). In addition, you can use double quotation marks, as you would for a string, but doing so requires VBA to perform an extra step to convert the string to a date. If you rewrite the fragment as follows to include the date delimiter, VBA prints "4/5/98" to the Immediate window:

```
Dim dteDate As Date
dteDate = #4/5/98#
Debug.Print dteDate
```

Assembling a Date

To work with a date in code, you sometimes must break it down into its component parts-that is, its day, month, and year. When you have done this, you can perform a calculation on one element, and then reassemble the date. To break a date into components, you can use the Day, Month, and Year functions. Each of these functions takes a date and returns the day, month, or year portion, respectively, as an Integer value. For example, Year(#2/23/98#) returns "1998."

To reassemble a date, you can use the DateSerial function. This function takes three integer arguments: a year, a month, and a day value. It returns a Date value that contains the reassembled date.

Often you can break apart a date, perform a calculation on it, and reassemble it all in one step. For example, to find the first day of the month, given any date, you can write a function similar to the following one:

```
Function FirstOfMonth(Optional dteDate As Date) As Date
  ' This function calculates the first day of a month, given a date. If no date is passed in, the function uses the current date.
  If CLng(dteDate) = 0 Then
    dteDate = Date
  End If
  FirstOfMonth = DateSerial(Year(dteDate), Month(dteDate), 1)        ' Find the first day of this month.
End Function
```

The FirstOfMonth procedure takes a date or, if the calling procedure does not pass one, uses the current date. It breaks the date into its component year and month, and then reassembles the date using 1 for the day argument. Calling this procedure with the dteDate argument #2/23/98# returns "2/1/98".

The following procedure uses the same strategy to return the last day of a month, given a date:

```
Function LastOfMonth(Optional dteDate As Date) As Date
  ' This function calculates the last day of a month, given a date. If no date is passed in, the function uses the current date.
  If CLng(dteDate) = 0 Then
    dteDate = Date
  End If
  ' Find the first day of the next month, then subtract one day.
  LastOfMonth = DateSerial(Year(dteDate), Month(dteDate) + 1, 1) - 1
End Function
```

Microsoft® Visual Basic® for Applications (VBA) also provides functions that you can use to disassemble and reassemble a time value in the same manner. The Hour, Minute, and Second functions return portions of a time value; the TimeSerial function takes an hour, minute, and second value and returns a complete time value.

Getting Part of a Date

The previous section showed how to return the year, month, and day from a date. You can get other information about a date as well, such as what quarter or week it falls in, or what day of the week it is.

The Weekday function takes a date and returns a constant indicating on what day of the week it falls. The following procedure takes a date and returns True if the date falls on a workday-that is, Monday through Friday-and False if it falls on a weekend.

```
Function IsWorkday(Optional dteDate As Date) As Boolean
  ' This function determines whether a date falls on a weekday. If no date passed in, use today's date.
  If CLng(dteDate) = 0 Then
    dteDate = Date
  End If
  Select Case Weekday(dteDate)      ' Determine where in week the date falls.
    Case vbMonday To vbFriday
      IsWorkday = True
    Case Else
      IsWorkday = False
  End Select
End Function
```

In addition to the individual functions that return part of a date-Year, Month, Day, and Weekday-Microsoft® Visual Basic® for Applications (VBA) includes the DatePart function, which can return any part of a date. Although it might seem redundant, the DatePart function gives you slightly more control over the values you return, because it gives you the option to specify the first day of the week and the first day of the year. For this reason, it can be useful when you are writing code that might run on systems in other countries. In addition, the DatePart function is the only way to return information about what quarter a date falls into.

Adding and Subtracting Dates

To add an interval to a given date, you must use the DateAdd function, unless you are adding days to a date. As mentioned earlier, because the integer portion of a Date variable represents the number of days that have passed since December 30, 1899, adding integers to a Date variable is equivalent to adding days.

By using the DateAdd function, you can add any interval to a given date: years, months, days, weeks, quarters. The following procedure finds the anniversary of a given date; that is, the next date on which it occurs. If the anniversary has already occurred this year, the procedure returns the date of the anniversary in the next year.

```
Function Anniversary(dteDate As Date) As Date
  ' This function finds the next anniversary of a date. If the date has already passed for this year, it returns the date on which the anniversary occurs in the following year.
  Dim dteThisYear As Date
  dteThisYear = DateSerial(Year(Date), Month(dteDate), Day(dteDate))          ' Find corresponding date this year.
  ' Determine whether it's already passed.
  If dteThisYear < Date Then
    Anniversary = DateAdd("yyyy", 1, dteThisYear)
  Else
    Anniversary = dteThisYear
  End If
End Function
```

To find the interval between two dates, you can use the DateDiff function. The interval returned can be any of several units of time: days, weeks, months, years, hours, and so on.

The following example uses the DateDiff function to return the day number for a particular day of the year. The procedure determines the last day of the last year by using the DateSerial function, and then subtracts that date from the date that was passed in to the procedure.

```
Function DayOfYear(Optional dteDate As Date) As Long
  ' This function takes a date as an argument and returns  the day number for that year. If the dteDate argument is omitted, the function uses the current date.
    ' If dteDate argument has not been passed, dteDate is  initialized to 0 (or December 30, 1899, the date equivalent of 0).
  If CLng(dteDate) = 0 Then
    dteDate = Date                    ' Use today's date.
  End If
  ' Calculate the number of days that have passed since December 31 of the previous year.
  DayOfYear = Abs(DateDiff("d", dteDate, DateSerial(Year(dteDate) - 1, 12, 31)))
End Function
```

Calling this procedure with the value of #2/23/98# returns "54."

Calculating Elapsed Time

You can use the DateAdd and DateDiff functions to calculate the time that has elapsed between two dates, and then, with a little additional work, present that time in the desired format. For example, the following procedure calculates a person's age in years, taking into account whether his or her birthday has already occurred in the current year.

Using the DateDiff function to determine the number of years between today and a birthdate does not always give a valid result because the DateDiff function rounds to the next year. If a person's birthday has not yet occurred, using the DateDiff function will make the person one year older than he or she actually is.

To remedy this situation, the procedure checks to see whether the birthday has already occurred this year, and if it has not, it subtracts 1 to return the correct age.

```
Function CalcAge(dteBirthdate As Date) As Long
  Dim lngAge As Long
  If Not IsDate(dteBirthdate) Then  dteBirthdate = Date   ' Make sure passed-in value is a date.
  If dteBirthdate > Date Then  dteBirthdate = Date   ' Make sure birthdate is not in the future. If it is, use today's date.
  ' Calculate the difference in years between today and birthdate.
  lngAge = DateDiff("yyyy", dteBirthdate, Date)
  ' If birthdate has not occurred this year, subtract 1 from age.
  If DateSerial(Year(Date), Month(dteBirthdate), Day(dteBirthdate)) > Date Then  lngAge = lngAge - 1
  CalcAge = lngAge
End Function
```

Working with Files

With the advent of the Scripting Runtime object library, you can work with drives, folders, and files as objects.

In This Section

The Microsoft Scripting Runtime Object Library
Understand the Scripting Runtime Object Library, and learn how to set a reference to it.

Returning Files from the File System
Use the FileSystemObject to work with drives, folders, and files in the file system.

Setting File Attributes
Use the File object and Folder object to read or set file or folder attributes.

Logging Errors to a Text File
Use objects to write to a text file, return an object that refers to a new or existing file, and use methods to open it for input or output.

The Dictionary Object
Understand the features of the Dictionary object-the Exists method, the CompareMode property, the Key property, and the RemoveAll method.

The Microsoft Scripting Runtime Object Library

When you install the Office XP applications, one of the object libraries installed on your system is the Scripting Runtime object library. This object library contains objects that are useful from either Microsoft® Visual Basic® for Applications (VBA) or script, so it is provided as a separate library.

The objects in the Scripting Runtime library provide easy access to the file system, and make reading and writing to a text file much simpler than it is in previous versions.

By default, no reference is set to this library, so you must set a reference before you can use it. If Microsoft Scripting Runtime does not appear in the References dialog box (Tools menu), you should be able to find it in the Windows system directory as Scrrun.dll.

The top-level objects in the Scripting Runtime object library are the Dictionary object and the FileSystemObject object. To use the Dictionary object, you create an object variable of type Dictionary, then set it to a new instance of a Dictionary object:

```
Dim dctDict As Dictionary
Set dctDict = New Dictionary
```

To use the other objects in the Scripting Runtime library in code, you must first create a variable of type FileSystemObject, and then use the New keyword to create a new instance of the FileSystemObject, as shown in the following code fragment:

```
Dim fsoSysObj As FileSystemObject
Set fsoSysObj = New FileSystemObject
```

You can then use the variable that refers to the FileSystemObject to work with the Drive, Folder, File, and TextStream objects.

The following table describes the objects contained in the Scripting Runtime library.

Object	Collection	Description
Dictionary		Top-level object. Similar to the VBA Collection object. Use this to store data key item pairs.
Drive	Drives	Refers to a drive or collection of drives on the system.
File	Files	Refers to a file or collection of files in the file system.
FileSystemObject		Top-level object. Use this object to access drives, folders, and files in the file system.
Folder	Folders	Refers to a folder or collection of folders in the file system.
TextStream		Refers to a stream of text that is read from, written to, or appended to a text file.

Returning Files from the File System

When you have created a new instance of the FileSystemObject, you can use it to work with drives, folders, and files in the file system.

The following procedure returns the files in a particular folder to a Dictionary object. The GetFiles procedure takes three arguments: the path to the directory, a Dictionary object, and an optional Boolean argument that specifies whether the procedure should be called recursively. It returns a Boolean value indicating whether the procedure was successful.

The procedure first uses the GetFolder method to return a reference to a Folder object. It then loops through the Files collection of that folder and adds the path and file name for each file to the Dictionary object. If the blnRecursive argument is set to True, the GetFiles procedure is called recursively to return the files in each subfolder.

```
Function GetFiles(strPath As String, dctDict As ScriptingDictionary, Optional blnRecursive As Boolean) As Boolean
    ' This procedure returns all the files in a directory into a Dictionary object. If called recursively, it also returns all files in subfolders.
  Dim fsoSysObj      As FileSystemObject
  Dim fdrFolder      As Folder
  Dim fdrSubFolder   As Folder
  Dim filFile        As File

  Set fsoSysObj = New FileSystemObject            ' Return new FileSystemObject.
  On Error Resume Next
  Set fdrFolder = fsoSysObj.GetFolder(strPath)            ' Get folder.
  If Err <> 0 Then                        ' Incorrect path.
    GetFiles = False
    GoTo GetFiles_End
  End If
  On Error GoTo 0
  For Each filFile In fdrFolder.Files             ' Loop through Files collection, adding to dictionary.
    dctDict.Add filFile.Path, filFile.Name
  Next filFile
  If blnRecursive Then                            ' If Recursive flag is true, call recursively.
    For Each fdrSubFolder In fdrFolder.SubFolders
      GetFiles fdrSubFolder.Path, dctDict, True
    Next fdrSubFolder
  End If
  GetFiles = True                                 ' Return True if no error occurred.

GetFiles_End:
  Exit Function
End Function
```

You can use the following procedure to test the GetFiles procedure. This procedure creates a new Dictionary object and passes it to the GetFiles procedure.

```
Sub TestGetFiles()
  ' Call to test GetFiles function.
  Dim dctDict As ScriptingDictionary
  Dim varItem As Variant

  Set dctDict = New Dictionary                    ' Create new dictionary.
  ' Call recursively, return files into Dictionary object.
  If GetFiles(GetTempDir, dctDict, True) Then
```

```
    For Each varItem In dctDict                ' Print items in dictionary.
      Debug.Print varItem
    Next
  End If
End Sub
```

You can also use the Office FileSearch object to find a file or group of files. The FileSearch object has certain advantages in that you can search subfolders, search for a particular file type, or search the contents of a file by simply setting a few properties.

On the other hand, the Microsoft Scripting Runtime object library makes it possible for you to work with individual files or folders as objects that have their own methods and properties.

Setting File Attributes

The File object and Folder object provide an Attributes property that you can use to read or set a file or folder's attributes, as shown in the following example.

The ChangeFileAttributes procedure takes four arguments: the path to a folder, an optional constant that specifies the attributes to set, an optional constant that specifies the attributes to remove, and an optional argument that specifies that the procedure should be called recursively. You can specify many attributes by using any logical combination of the file attributes.

If the folder path passed in is valid, the procedure returns a Folder object. It then checks to see if the lngSetAttr argument was provided. If so, it loops through all the files in the folder, appending the new attribute or attributes to each file's existing attributes. It does the same for the lngRemoveAttr argument, except in this case it removes the specified attributes if they exist for files in the collection.

Note
The following code does not handle the case of setting the attributes to Normal or zero (0). If you want to set the attribute to Normal, you must use lngRemoveAttr for all the attributes.

Finally, the procedure checks whether the blnRecursive argument has been set to True. If so, it calls the procedure for each file in each subfolder of the strPath argument.

```
Function ChangeFileAttributes(strPath As String, Optional lngSetAttr As FileAttribute, Optional lngRemoveAttr As FileAttribute, Optional blnRecursive As Boolean) As Boolean
  ' This function takes a directory path, a value specifying file  attributes to be set, a value specifying file attributes to be
  ' removed, and a flag that indicates whether it should be called  recursively. It returns True unless an error occurs.
  Dim fsoSysObj      As FileSystemObject
  Dim fdrFolder      As Folder
  Dim fdrSubFolder   As Folder
  Dim filFile        As File

  Set fsoSysObj = New FileSystemObject              ' Return new FileSystemObject.
  On Error Resume Next
  Set fdrFolder = fsoSysObj.GetFolder(strPath)      ' Get folder.
  If Err <> 0 Then                                  ' Incorrect path.
    ChangeFileAttributes = False
    GoTo ChangeFileAttributes_End
  End If
  On Error GoTo 0
' If caller passed in attribute to set, set for all.
  If lngSetAttr Then
    For Each filFile In fdrFolder.Files
        filFile.Attributes = filFile.Attributes Or lngSetAttr
    Next
  End If
' If caller passed in attribute to remove, remove for all.
  If lngRemoveAttr Then
    For Each filFile In fdrFolder.Files
        filFile.Attributes = filFile.Attributes - lngRemoveAttr
    Next
  End If
' If caller has set blnRecursive argument to True, then call
  ' function recursively.
  If blnRecursive Then
    ' Loop through subfolders.
    For Each fdrSubFolder In fdrFolder.SubFolders
      ' Call function with subfolder path.
      ChangeFileAttributes fdrSubFolder.Path, lngSetAttr, lngRemoveAttr, True
    Next
```

```
    End If
    ChangeFileAttributes = True
ChangeFileAttributes_End:
    Exit Function
End Function
```

Logging Errors to a Text File

The Scripting Runtime object library simplifies the code required to read from and write to a text file. To use the new objects to write to a text file, you return a file object that refers to a new or existing file, and then use the OpenAsTextStream method to open it for input or output. The OpenAsTextStream method has an IOMode argument, which you can set to indicate whether you want to read from the file, write to it, or append to it.

The OpenAsTextStream method returns a TextStream object, which is the object you use to work with the text in the file. To read a line, for example, you can use the TextStream object's ReadLine method; to write a line, you can use the WriteLine method. When you're finished working with the file, you can use the Close method to close it.

The following procedure logs an error to a text file. It takes two arguments: an ErrObject argument, which is a reference to the Err object that contains the current error, and an optional strProcName argument, which specifies the procedure in which the error occurred.

The LogError procedure writes to a text file in the Microsoft®Windows® Temp folder. To determine where the Windows Temp folder is, it calls another procedure, the GetTempDir procedure. This procedure makes a call to the Windows application programming interface (API) to determine the Temp folder. Windows cannot boot without a designated Temp folder, so you can be certain that the Temp folder will always be available.

The LogError procedure is meant to be used to log multiple errors. The first time the procedure is called, no log file exists, so it must create one. On each subsequent call, the procedure must open the existing log file. The simplest way to do this is to look for the name of the file that you're expecting, and if it is not there, handle the error and create the file.

Unfortunately, when the procedure is first called and the error occurs, the existing information in the Err object is cleared and the information for the new error takes its place. Because there is only one Err object available in Microsoft® Visual Basic® for Applications (VBA), the error information that you passed to the procedure is lost when a new error occurs. Therefore, the first thing that the procedure does is to store the error number and description of the error in variables.

When the procedure has a reference to the text file (APP_ERROR_LOG), it opens it for appending, and then writes the error information to the file line by line.

```
Sub LogError(errX As ErrObject, Optional strProcName As String)
  ' This procedure logs errors to a text file. It is used in this section to log synchronization errors.
  ' Arguments:  errX: A variable that refers to the VBA Err object.
  Dim fsoSysObj    As FileSystemObject
  Dim filFile      As File
  Dim txsStream    As TextStream
  Dim lngErrNum    As Long
  Dim strPath      As String
  Dim strErrText   As String
   Set fsoSysObj = New FileSystemObject
  lngErrNum = errX.Number                    ' Store error information.
  strErrText = errX.Description
  errX.Clear                                 ' Clear error.
  strPath = GetTempDir                       ' Return Windows Temp folder.
  If Len(strPath) = 0 Then  GoTo LogError_End
  On Error Resume Next
  ' See if file already exists.
  Set filFile = fsoSysObj.GetFile(strPath & APP_ERROR_LOG)
  ' If not, then create it.
  If Err <> 0 Then   Set filFile = fsoSysObj.CreateTextFile(strPath & APP_ERROR_LOG)
  On Error GoTo 0
  ' Open file as text stream for reading.
  Set txsStream = filFile.OpenAsTextStream(ForAppending)
  ' Write error information and close.
  With txsStream
    .WriteLine lngErrNum
    .WriteLine strErrText
    If Len(strProcName) > 0 Then .WriteLine strProcName
    .WriteLine Now
    .WriteBlankLines 1
    .Close
```

```
    End With
LogError_End:
    Exit Sub
End Sub
```

To try the LogError procedure, you can call the following procedure. This procedure suspends error handling, then uses the Raise method of the Err object to force an error. It then passes the Err object to the LogError procedure, along with the name of the procedure that caused the error.

```
Sub TestLogError()
    ' This procedure tests the LogError function.
    On Error Resume Next
    ' Raise an error.
    Err.Raise 11
    ' Log it.
    LogError Err, "TestLogError"
End Sub
```

The Dictionary Object

The Dictionary object is a data structure that can contain sets of pairs, where each pair consists of an item, which can be any data type, and a key, which is a unique String value that identifies the item. The Dictionary object is similar in some ways to the VBA Collection object; however, the Dictionary object offers certain features that the Collection object lacks, including:

- The Exists method. You can use this method to determine whether a particular key, and its corresponding item, exist in a Dictionary object. The Exists method makes it simpler and more efficient to search a Dictionary object than to search a Collection object.
- The CompareMode property. Setting this property specifies the text-comparison mode for the Dictionary object, so that you can search for a key in either a case-sensitive or case-insensitive manner. By default, it is set to BinaryCompare, which means that the Exists method will return True only if it finds a binary match. There is no way to specify a text-comparison mode for a key that retrieves an item from a Collection object.
- The Key property. This property enables you to return the key for a particular item in the dictionary. An item in a Collection object also has a key, which you can use to retrieve that item; however, there is no way to retrieve the key itself.
- The RemoveAll method. This method removes all items in the Dictionary object. A Collection object, on the other hand, has no method for removing all items at once, although setting the Collection object to Nothing has the same effect.

The primary advantage of the Dictionary object over the Collection object is the fact that it is easier to search a Dictionary object for a given item. Despite this advantage, the Dictionary object does not replace the Collection object entirely. The Collection object is useful in some situations where the Dictionary object is not. For example, if you're creating a custom object model, you can use a Collection object to store a reference to a custom collection, but you cannot use a Dictionary object to do this.

For more information about the Dictionary object, see the VBScript documentation on the Microsoft Scripting Technologies Web site at http://msdn.microsoft.com/scripting/default.htm

Understanding Arrays

Arrays make it possible for you to refer to a series of variables by the same name and to use a number (an index) to tell them apart. This helps you create smaller and simpler code in many situations, because you can set up loops that deal efficiently with any number of cases by using the index number. Arrays are useful when you must store a number of values of the same type, but you do not know how many, or you do not want to create individual variables to store them all.

For example, suppose you must store a numeric value for every day of the year. You could declare 365 separate numeric variables, but that would be a lot of work. Instead, you can create an array to store all the data in one variable. The array itself is a single variable with multiple elements; each element can contain one piece of data.

You can use loops, together with a couple of special functions for working with arrays, to assign values to or retrieve values from the various elements of an array.

In This Section

Creating Arrays
Understand how to create two types of arrays in Microsoft® Visual Basic® for Applications (VBA)-fixed-size arrays and dynamic arrays.

Arrays and Variants
Learn how a Variant variable can store an array.

Assigning One Array to Another
Assign one array to another if two dynamic arrays have the same data type.

Returning an Array from a Function
Call a procedure that returns an array and assign it to another array.

Passing an Array to a Procedure
Declare an array in one procedure, and then pass that array to another procedure to be modified.

Sorting Arrays
Understand how to sort an array, which is an iterative process that requires a complex algorithm.

Using the Filter Function to Search String Arrays
Search a string array if you simply must know whether an item exists in the array by using the Filter function.

Using a Binary Search Function to Search Numeric Arrays
Learn how the binary-search algorithm performs efficient searching on a sorted array-whether numeric or string.

Searching a Dictionary
Use object programming constructs, such as For Each...Next and With...End With statements, to work with the Dictionary object.

Creating Arrays

You can create two types of arrays in Microsoft®Visual Basic® for Applications (VBA)-fixed-size arrays and dynamic arrays. A fixed-size array has a fixed number of elements, and is useful only when you know exactly how many elements your array will have while you're writing the code. Most of the time you'll create dynamic arrays.

Arrays can be of any data type. The data type for an array specifies the data type for each element of the array; for example, each element of an array of type Long can contain a Long value. The following code fragment declares an array variable of type Long:

```
Dim alngNum() As Long
```

Note
You do not have to include the parentheses when you refer to an array variable, except when you declare it, resize it, or refer to an individual element. However, you might want to include the parentheses everywhere to make it clear that the variable is an array.

When you have declared a dynamic array variable, you can resize the array by using the ReDim statement. To resize the array, you provide a value for the upper bound, and optionally, for the lower bound. The upper and lower bound of an array refer to the beginning and ending indexes for the array.

You must specify the upper bound for the array when you resize it. The lower bound is optional, but it is a good idea to include it, so that it is obvious to you what the lower bound of the array is:

```
' This array contains 100 elements.
ReDim alngNum(0 To 99)
```

If you do not include the lower bound, it is determined by the Option Base setting for the module. By default, the Option Base setting for a module is 0. You can set it to 1 by entering Option Base 1 in the Declarations section of the module.

If you are using the ReDim statement on an array that contains values, those values might be lost when the array is resized. To ensure that any values in the array are maintained, you can use the Preserve keyword with the ReDim statement, as follows:

```
ReDim Preserve alngNum(0 To 364)
```

Resizing an array with the Preserve keyword can be slow, so you want to do it as infrequently as possible. A good way to minimize use of the Preserve keyword in your code is to estimate the amount of data you require to store and size the array accordingly. If an error occurs because you have not made the array large enough, you can resize it within the error handler as many times as necessary. When you're through working with the array, if it is larger than you require, you can resize it to make it just large enough to contain the data it currently has.

Arrays and Variants

A Variant variable can store an array. For example, the following code fragment assigns an array of type String to a Variant variable:

```
Dim astrItems(0 To 9)    As String
Dim varItems             As Variant
varItems = astrItems
```

When a static array is initialized, or when a dynamic array is redimensioned, each element is initialized according to its type. In other words, String type elements are initialized to zero-length strings, Integer and Long type elements are initialized to zero (0), and Variant type elements are initialized to Empty. The point is that in the preceding example, it is not necessary to fill the array to work with it. By simply declaring an array of ten elements as type String, we've created an array containing ten zero-length strings.

An array of type Variant can store any data type in any of its elements. For example, a Variant type array can have one element of type String, one element of type Long, and another of type Date. It can even store a Variant variable that contains another array.

A Variant type array can also store an array of objects. If you know that an array will store only objects, you can declare it as type Object rather than as type Variant. And if you know that an array of objects will contain only one type of object, you can declare the array as that object type.

Tip
You might want to consider using a Collection or Dictionary object to store groups of objects in a single variable, rather than creating an array of objects. The advantage to using an array over a Collection or Dictionary object is that it is easy to sort. But if you're storing objects, you probably do not care about the sort order. Because a Collection or Dictionary object resizes itself automatically, you do not have to worry about keeping track of its size, as you do with an array.

Assigning One Array to Another

If two dynamic arrays have the same data type, you can assign one array to another. Assigning one array to another of the same type is quick because the first array is simply pointed to the memory location that stores the second array.

For example, the following code fragment assigns one string array to another:

```
Dim astr1() As String
Dim astr2(0 To 9) As String
astr1 = astr2
```

Note
This type of assignment works only for arrays of the same type. The two arrays must both be dynamic arrays, and they must be declared as the exact same type: if one is type String, the other must be type String. It cannot be type Variant or any other data type. If you want to assign one array's elements to an array of a different type, you must create a loop and assign each element one at a time.

Returning an Array from a Function

The previous example assigned one array variable to another. Based on this example, you might guess that you can also call a procedure that returns an array and assign that to another array, as in the following code fragment:

```
Dim astr1() As String
astr1 = ReturnArray
```

To return an array, a procedure must have a return value type of the array's data type, or of type Variant. The advantage to declaring a procedure to return a typed array versus a Variant value is that you are not required to use the IsArray function to ensure that the procedure indeed returned an array. If a procedure returns a value of type Variant, you might want to check its contents before performing array operations.

The ReturnArray procedure prompts the user for input and creates an array of the resulting values, resizing the array as required. Note that to return an array from a procedure, you simply assign the array to the name of the procedure.

```
Function ReturnArray() As String()
  ' This function fills an array with user input, then returns the array.
  Dim astrItems()     As String
  Dim strInput        As String
  Dim strMsg          As String
  Dim lngIndex        As Long
  On Error GoTo ReturnArray_Err
  strMsg = "Enter a value or press Cancel to end:"
  lngIndex = 0
  strInput = InputBox(strMsg)                    ' Prompt user for first item to add to array.
  If Len(strInput) > 0 Then
    ReDim astrItems(0 To 2)                      ' Estimate size of array.
    astrItems(lngIndex) = strInput
    lngIndex = lngIndex + 1
  Else
    ' If user cancels without adding item, don't resize array.
    ReturnArray = astrItems
    GoTo ReturnArray_End
  End If
  ' Prompt user for additional items and add to array.
  Do
    strInput = InputBox(strMsg)
    If Len(strInput) > 0 Then
      astrItems(lngIndex) = strInput
      lngIndex = lngIndex + 1
    End If
  Loop Until Len(strInput) = 0                   ' Loop until user cancels.
  ' Resize to current value of lngIndex - 1.
```

```
  ReDim Preserve astrItems(0 To lngIndex - 1)
  ReturnArray = astrItems
ReturnArray_End:
  Exit Function
ReturnArray_Err:
  ' If upper bound is exceeded, enlarge array.
  If Err = ERR_SUBSCRIPT Then ' Subscript out of range
    ReDim Preserve astrItems(lngIndex * 2)        ' Double the size of the array.
    Resume
  Else
    MsgBox "An unexpected error has occurred!", vbExclamation
    Resume ReturnArray_End
  End If
End Function
```

When you call a procedure that returns an array, you must take into account the case in which the returned array does not contain any elements. For example, in the preceding ReturnArray procedure, if you cancel the input box the first time that it appears, the array returned by the procedure contains no elements. The calling procedure must check for this condition. The best way to do this is to define a procedure such as the following one, which takes an array and checks the upper bound. If the array contains no elements, checking the upper bound causes a trappable error.

```
Function IsArrayEmpty(varArray As Variant) As Boolean
  ' Determines whether an array contains any elements. Returns False if it does contain elements, True  if it does not.
  Dim lngUBound As Long
  On Error Resume Next
  ' If the array is empty, an error occurs when you  check the array's bounds.
  lngUBound = UBound(varArray)
  If Err.Number <> 0 Then
    IsArrayEmpty = True
  Else
    IsArrayEmpty = False
  End If
End Function
```

Note

The VBA Split and Filter functions can also return an array that contains no elements. Checking the upper or lower bounds on an array returned by either of these procedures does not cause an error, however. When the Split or Filter function returns an array containing no elements, the lower bound of that array is 0, and the upper bound is -1. Therefore, to determine whether the returned array contains any elements, you can check for the condition where the upper bound of the array is less than the lower bound.

Passing an Array to a Procedure

You can declare an array in one procedure, and then pass the array to another procedure to be modified. The procedure that modifies the array does not have to return an array. Arrays are passed by reference, meaning that one procedure passes to the other a pointer to the array's location in memory. When the second procedure modifies the array, it modifies it at that same memory location. Therefore, when execution returns to the first procedure, the array variable refers to the modified array.

Sorting Arrays

Sorting an array is an iterative process that requires a fairly sophisticated algorithm. An example of a common sorting algorithm, the QuickSort algorithm. The QuickSort algorithm is explained in thorough detail in the Visual Basic Language Developer's Handbook by Ken Getz and Mike Gilbert (Sybex, 2000)-a good place to start if you're looking for more information about sorting arrays.

In brief, the QuickSort algorithm works by using a divide-and-sort strategy. It first finds the middle element in the array, then works its way from the rightmost element to the middle, and from the leftmost element to the middle, comparing elements on both sides of the middle value and swapping their values if necessary. When this part of the sort is complete, the values on the right side are all greater than those on the left, but they're not necessarily in order. The procedure then looks at the values on the left side by using the same strategy-finding a middle value and swapping elements on both sides. It does this until all the elements on the left side have been sorted, and then it tackles the right side. The procedure calls itself recursively and continues executing until the entire array has been sorted.

Using the Filter Function to Search String Arrays

The Filter function makes it easy to search a string array if you simply must know whether an item exists in the array. The Filter function takes a string array and a string containing the search text. It returns a one-dimensional array containing all the elements that match the search text.

One potential disadvantage of using the Filter function to search an array is that it does not return the index of the elements of the array that match the search text. In other words, the Filter function tells you whether an element exists in an array, but it does not tell you where.

Another potential problem with using the Filter function to search an array is that there is no way to specify whether the search text should match the entire element or whether it only must match a part of it. For example, if you use the Filter function to search for an element matching the letter "e," the Filter function returns not only those elements containing only "e," but also any elements containing larger words that include "e."

The following procedure augments the capabilities of the Filter function to search an array and returns only elements that match exactly. The FilterExactMatch procedure takes two arguments: a string array to search and a string to find. It uses the Filter function to return an array containing all elements that match the search string, either partially or entirely. It then checks each element in the filtered array to verify that it matches the search string exactly. If the element does match exactly, it is copied to a third string array. The function returns this third array, which contains only exact matches.

```
Function FilterExactMatch(astrItems() As String, strSearch As String) As String()
  ' This function searches a string array for elements that exactly match the search string.
  Dim astrFilter()    As String
  Dim astrTemp()      As String
  Dim lngUpper        As Long
  Dim lngLower        As Long
  Dim lngIndex        As Long
  Dim lngCount        As Long
  astrFilter = Filter(astrItems, strSearch)         ' Filter array for search string.
  lngUpper = UBound(astrFilter)                     ' Store upper and lower bounds of resulting array.
  lngLower = LBound(astrFilter)
  ReDim astrTemp(lngLower To lngUpper)      ' Resize temporary array to be same size.
  ' Loop through each element in filtered array.
  For lngIndex = lngLower To lngUpper
    ' Check that element matches search string exactly.
    If astrFilter(lngIndex) = strSearch Then
      astrTemp(lngCount) = strSearch                ' Store elements that match exactly in another array.
      lngCount = lngCount + 1
    End If
  Next lngIndex
  ReDim Preserve astrTemp(lngLower To lngCount - 1)                 ' Resize array containing exact matches.
  ' Return array containing exact matches.
  FilterExactMatch = astrTemp
End Function
```

Using a Binary Search Function to Search Numeric Arrays

The Filter function works well for searching string arrays, but it is inefficient for numeric arrays. To use the Filter function for a numeric array, you have to convert all of the numeric elements to strings, an extra step that impairs performance. Then you must perform string-comparison operations, when numeric comparisons are much faster.

Although it is more involved, the binary-search algorithm performs efficient searching on a sorted array-whether numeric or string. The binary-search algorithm divides a set of values in half, and determines whether the value being sought lies in the first half or the second half. Whichever half contains the value is kept, and the other half is discarded. The remaining half is then again divided in half, and the process repeats until the algorithm either arrives at the sought value or determines that it is not in the set. Note that the array must be sorted for this algorithm to work.

For an in-depth discussion of the binary-search algorithm, see the Visual Basic Language Developer's Handbook by Ken Getz and Mike Gilbert (Sybex, 2000).

Searching a Dictionary

Strictly speaking, a Dictionary object is not an array, but it is similar. Both are data structures that can store multiple values. The Dictionary object has certain advantages over an array: you can use object programming constructs such as For Each...Next and With...End With statements to work with it, and you do not have to worry about sizing it, as you do an array.

If you use a Dictionary object instead of an array to store a set of data, you can check whether a particular item exists in the dictionary by calling the Exists method of the Dictionary object and passing it the key for the item you want. However, the Exists method does not provide any information regarding where the item is within the dictionary or how many times it occurs.

An advantage of using the Exists method with a Dictionary object, rather than using the Filter function with an array, is that the Exists method returns a Boolean value, while the Filter function returns another array. If you are not required to know how many times the search item occurs, using the Dictionary object might simplify your code.

Tips for Defining Procedures in VBA

When you are defining a Function or Sub procedure, you have options available to you that can make your code more extensible or more flexible. The following sections discuss how to extend your procedures by using optional arguments, using parameter arrays to pass a variable number of arguments, and passing arguments by value and by reference.

In This Section

Using Optional Arguments
Add functionality without updating all the code that calls those procedures by adding optional arguments to user-defined procedures.

Using Parameter Arrays
Pass an array of arguments to a procedure by using a parameter array.

Passing Arguments by Value or by Reference
Understand the difference between passing arguments by value and passing arguments by reference when you define a procedure.

Using Optional Arguments

Optional arguments are arguments that are not required for a procedure to be compiled and run. Many built-in functions and methods take optional arguments. Adding optional arguments to user-defined procedures is a way to add functionality without updating all the code that calls those procedures. In addition, if you declare arguments that are not always required as optional, you can minimize resource use by passing only those arguments that are necessary for a given procedure call.

To define an optional argument in a user-defined procedure, use the Optional keyword. You can have as many optional arguments as you want, but when you denote one argument as optional, any arguments that follow it in the argument list must be optional also, as shown in the following procedure definition:

```
Function SomeProc(strRequired1 As String, strRequired2 As String, Optional lngOpt1 As Long, Optional blnOpt2 As Boolean)
```

Within the body of the procedure, you must have a way to check whether the optional argument was passed in. In many cases, if an optional argument has not been passed in, you might want it to have a default value. If the calling procedure does not provide a value for an optional argument, the optional argument is automatically initialized in the same way it would be if it were a variable-string arguments are initialized to a zero-length string, numeric arguments to zero (0), Boolean arguments to False, and so on.

You can override this default initialization by providing a different default value for the optional argument in the procedure definition. The value you provide becomes the default value when the calling procedure fails to pass a value for the optional argument. The following procedure definition sets the default value for an argument of type Long to 1 and for an argument of type Boolean to True:

```
Function SomeProc(strRequired1 As String, strRequired2 As String, Optional lngOpt1 As Long=1, _
        Optional blnOpt2 As Boolean=True)
```

As you can see, an argument of any data type except Variant always will have a value, and it might not be possible to determine within the procedure whether the value was passed in or whether it is the default value. If you must know whether the argument was passed in, define the optional argument as type Variant. Then, use the IsMissing function within the procedure to determine whether the argument has been passed in, as shown in the following procedure:

```
Sub TestIsMissing(varTest As Variant)
  If IsMissing(varTest) Then
    Debug.Print "Missing"
  Else
    Debug.Print varTest
  End If
End Sub
```

The IsMissing function works only with the Variant data type; because any other data type always will have a default initialization value, the IsMissing function will return False regardless of whether a value has been passed for the argument.

Using Parameter Arrays

You can pass an array of arguments to a procedure by using a parameter array. The advantage to using a parameter array is you are not required to know at design time how many arguments will be passed to a procedure-you can pass a variable number of arguments when you call it.

To define a parameter array, use the ParamArray keyword followed by an array of type Variant, as shown in the following procedure definition:

```
Function SomeProc(ParamArray avarItems() As Variant)
```

A parameter array always must be an array of type Variant, and it always must be the last argument in the argument list.

To call a procedure that includes a parameter array, pass in a set of any number of arguments, as shown here:

? SomeProc("red", "yellow", "blue", "green", "orange")

Within the body of the procedure, you can work with the parameter array as you would with any other array.

Passing Arguments by Value or by Reference

When you define a procedure, you have two choices regarding how arguments are passed to it: by reference or by value. When a variable is passed to a procedure by reference, Microsoft® Visual Basic®for Applications (VBA) actually passes the variable's address in memory to the procedure, which can modify it directly. When execution returns to the calling procedure, the variable contains the modified value.

When an argument is passed by value, VBA passes a copy of the variable to the procedure. Then, the procedure modifies the copy, and the original value of the variable remains intact; when execution returns to the calling procedure, the variable contains the same value that it had before being passed.

By default, VBA passes arguments by reference. To pass an argument by value, precede the argument with the ByVal keyword in the procedure definition, as shown here:

Function SomeProc(strText As String, ByVal lngX As Long) As Boolean

If you want to denote explicitly that an argument is passed by reference, you can preface the argument with the ByRef keyword in the argument list.

Passing by reference can be useful as long as you understand how it works. For example, you must pass arrays by reference; you will get a syntax error if you try to pass an array by value. Because arrays are passed by reference, you can pass an array to another procedure to be modified, and then you can continue working with the modified array in the calling procedure.

Optimizing VBA Code

There are many tips for optimizing your Microsoft®Visual Basic® for Applications (VBA) code, such as streamlining your code to conserve memory resources, creating object variables when you must refer to an object more than once within a procedure, minimizing concatenation operations, and so on.

In This Section

Declaring Variables
Streamline your memory requirements and speed up performance when you are using variables.

Mathematical Operations
Learn how to speed up operations on numbers.

String Operations
Understand how to enhance the performance of string operations.

Loops
Determine how to save resources when you are executing loops.

Declaring Variables

The following points provide suggestions for ways to streamline your memory requirements and speed up performance when you are using variables:

- To conserve memory resources, always declare all your variables with specific data types. When you declare a variable without a specific data type, Microsoft® Visual Basic® for Applications (VBA) creates a variable of type Variant, which requires more memory than any of the other data types.
- Be aware of how much memory each data type requires and what range of values it can store. Always use a smaller data type if possible, except in the case where using a smaller data type will force an implicit conversion. For example, because variables of type Integer are converted to variables of type Long, it makes sense to declare variables that will store integer values as type Long instead of as type Integer.
- Avoid using floating-point data types unless you must have them. Although it is larger, the Currency data type is faster than the Single data type, because the Currency data type does not use the floating-point processor.
- If you refer to an object more than once within a procedure, create an object variable and assign to it a reference to the object. Because the object variable stores the object's location in memory, VBA will not have to look up the location again.
- Declare object variables as specific types rather than as type Object, so you can take advantage of early binding.

Mathematical Operations

The following points provide suggestions for ways to speed up operations on numbers:

- When performing division on integers, use the integer division operator (\) rather than the floating-point division operator (/), which always returns a value of type Double regardless of the types of the numbers being divided.
- Keep in mind that any time you use a Single or Double value in an arithmetic expression with integer values, the integers are converted to Single or Double values, and the final result is a Single or Double value. If you are performing several operations on a number that is the result of an arithmetic operation, you might want to explicitly convert the number to a smaller data type.

String Operations

The following points provide suggestions for ways to enhance the performance of string operations:

- Minimize concatenation operations when you can. You can use the Mid function on the left side of the equal sign to replace characters within the string, rather than concatenating them together. The drawback to using the Mid function is that the replacement string must be the same length as the substring you are replacing.

```
Dim strText As String
strText = "this is a test"
Mid(strText, 11, 4) = "tent"
Debug.Print strText
```

- Microsoft® Visual Basic® for Applications (VBA) provides a number of intrinsic string constants that you can use to replace function calls. For example, you can use the vbCrLf constant to represent a carriage return/linefeed combination within a string, rather than using Chr(13) & Chr(10).

- String-comparison operations are slow. Sometimes, you can avoid them by converting a character in the string to an ANSI value. For example, the following code checks whether the first character in a string is a space:

```
If Asc(strText) = 32 Then
```

The previous code is faster than the following:

```
If Left(strText, 1) = " " Then
```

Loops

The following points provide suggestions for ways to save resources when you are executing loops:

- Analyze your loops to see whether you are repeating memory-intensive operations needlessly. For example, are there any variables that you can set outside the loop, rather than within it? Are you performing a conversion procedure each time through the loop that could be done outside the loop?
- Consider whether you must loop only until a certain condition is met. If so, you might be able to exit the loop early. For example, suppose you are performing data validation on a string that should not contain numeric characters. If you have a loop that checks each character in a string to determine whether the string contains any numeric characters, you can exit the loop as soon as you find the first numeric character.
- If you must refer to an element of an array within a loop, create a temporary variable that stores the element's value rather than referring to it within the array. Retrieving values from an array is slower than reading a variable of the same type.

5. ADD-INS, TEMPLATES, WIZARDS, AND LIBRARIES

Creating a Microsoft® Office XP application is about enhancing and extending powerful applications that you and other users already have on your desktops. You can take advantage of the features in Microsoft® Word, Microsoft® Excel, Microsoft® PowerPoint®, Microsoft® Access, Microsoft® FrontPage®, and Microsoft® Outlook®, as well as all the time and resources Microsoft has invested in developing and testing these applications, to build an application quickly and easily that meets users' requirements without requiring a lot of training and support.

One way to provide users with a custom application is to build an add-in. An add-in extends an application by adding functionality that is not in the core product itself. If you are a frequent user of Excel or Access, you might already be familiar with some of the add-ins that these applications include. For example, the Linked Table Manager in Access is an add-in that was built in Microsoft® Visual Basic® for Applications (VBA).

You can create two different types of add-ins: Component Object Model (COM) add-ins and application-specific add-ins. COM add-ins can work in more than one of the Office XP applications.

The other type of add-in you can create is an application-specific add-in. You can create application-specific add-ins in Office XP, as well as in previous versions of Office. An application-specific add-in works in only one application.

Another way to distribute a custom Office application is to create a template. A template provides the user with a basis for creating a new document. For example, a Word template might include the basic layout for a report that an employee can use to create a new document with the same layout and simply fill in the new information.

In addition to the add-ins and templates mentioned earlier, you also can create two specialized kinds of add-ins: wizards and code libraries. Wizards are add-ins, and they help users through a complex process step-by-step. Code libraries are add-ins in which you can store frequently used procedures and generic code. By setting a reference to a code library, you can call procedures stored within that library from your current VBA project.

In This Section

What Is a COM Add-in?
Extend the functionality of your Microsoft® Office-based applications without adding complexity for the user.

Building COM Add-ins for the Visual Basic Editor
Customize your development environment and work with components in a Microsoft® Visual Basic®for Applications (VBA) project from code.

Building COM Add-ins for Office Applications
By building COM add-ins, you can extend the functionality of your Microsoft® Office-based applications without adding complexity for the user.

Building Application-Specific Add-ins
Add functionality to Microsoft®Office XP applications by creating application-specific add-ins.

Creating Templates
Learn how to give users a framework within which to complete common tasks by using templates.

Creating Wizards
Understand how to create a wizard to walk users through a series of steps to create a new document, spreadsheet, presentation, database, or Web application and to deliver an application is that is easy to use.

What Is a COM Add-in?

A COM add-in is a dynamic-link library (DLL) that is specially registered for loading by the Microsoft® Office XP applications. You can build COM add-ins with any of the Office applications in Office XP Developer. In addition, you can create COM add-ins with Microsoft® Visual Basic® or Microsoft® Visual C++®. For more information about these tools, see the Microsoft Developer Network (MSDN®) Web site at http://msdn.microsoft.com/

Note
A COM add-in also can be a Microsoft® ActiveX® .exe file for Visual Basic. However, DLLs generally provide better performance than .exe files.

COM add-ins use the Component Object Model that makes it possible for you to create a single add-in that is available to one or many of the Office applications-Microsoft® Word, Microsoft® Excel, Microsoft® Access, Microsoft® PowerPoint®, Microsoft® Outlook®, Microsoft® FrontPage®, or even the Visual Basic Editor. By developing COM add-ins, you can extend the functionality of your Office-based applications without adding complexity for users.

COM Add-ins vs. Application-Specific Add-ins

In the previous and current versions of Microsoft® Word, Microsoft® Excel, Microsoft® Access, and Microsoft® PowerPoint®, you can use Microsoft® Visual Basic® for Applications (VBA) to create add-ins specific to each of those applications. For example, you can create an add-in for Word that builds a custom report from a selected database and another

add-in for Excel that performs a similar task. You save the Word add-in as a Word template file (*.dot), and the Excel add-in as an Excel add-in file (*.xla). Despite the fact that the add-ins share common code, you have to create separate add-ins to add functionality to both applications.

Note
Microsoft® Outlook® and Microsoft® FrontPage® do not provide any way to create application-specific add-ins by using VBA.

A COM add-in, on the other hand, can share some add-in functionality and code across applications. The COM Add-in project contains a component for each application in which it will run and is registered for each application. Usually, a COM add-in contains some code that is common across all applications and some that is specific to each application. For example, if you build a COM add-in to create a custom report in Word or Excel from a database, the code that accesses the database and retrieves a set of data can be shared. When you have retrieved the data, you must work with the Word object model to write the data to Word and with the Excel object model to write the data to Excel.

The following table lists both types of add-ins and their file extensions.

Add-ins	File extensions	Available to
Word add-ins (application-specific)	.dot, .wll, .wiz	Word only
Excel add-ins (application-specific)	.xla, .xll	Excel only
PowerPoint add-ins (application-specific)	.ppa, .pwz	PowerPoint only
Access add-ins (application-specific)	.mda, .mde	Access only
Exchange Client extensions (application-specific)	.dll	Outlook and Microsoft® Exchange clients only
COM add-ins	.dll	Word, Excel, Access, PowerPoint, Outlook, and FrontPage

COM add-ins and application-specific add-ins also differ in terms of how the user views and installs available add-ins. In all Microsoft® Office XP applications, the COM Add-Ins dialog box displays the available COM add-ins.

Viewing the List of Available COM Add-ins

By default, there is no menu item or toolbar button to display the COM Add-ins dialog box, but you can easily display it.

To add a menu item or toolbar button for the COM Add-ins dialog box

1. In the Microsoft® Office XP application, click **Customize** on the **Tools** menu.
2. Click the **Commands t** ab.
3. In the **Categorie** s list, click **Tools** .
4. In the **Commands** list, click **COM Add-ins** You might have to scroll through the list to find it.
5. Drag the **COM Add-ins** command to a toolbar or a menu.
6. Close the **Customize** dialog box.

Note
In Microsoft®Outlook®, you can access the **COM Add-Ins** dialog box if you click **Options** on the **Tools** menu, click the **Other** tab, and then click **Advanced Options** . In the **Advanced Options** dialog box, click **COM Add-Ins** .

When you click the COM Add-Ins toolbar button or menu item, the COM Add-Ins dialog box appears, showing the list of available COM add-ins. You can load (connect) or unload (disconnect) an add-in by selecting the check box next to it. Loading a COM add-in loads it into memory, so you can work with it. Unloading an add-in removes it from memory; you cannot use the add-in until you load it again.

You can add a new COM add-in to the list by clicking Add and locating the add-in. Clicking Add and selecting an add-in that does not appear in the list registers the add-in DLL if it is not registered already and adds the add-in to the list of available COM add-ins for an Office XP application.

To remove a COM add-in from the list, select it, and click Remove. Removing an add-in deletes the registry key that contains the name and load behavior of the add-in. The registry contains information about a COM add-in in two places. As with any other DLL, the add-in's DLL is registered as a unique object on the system. Additionally, information about the add-in is placed in another section of the registry to notify Office applications that the add-in exists. This section is deleted when you remove an add-in from the list. The DLL itself remains registered, and if you add the add-in to the list again, the add-in's informational section is re-created in the registry.

Note
You can add only DLLs that are COM add-ins to the list of available add-ins in the COM Add-Ins dialog box. Moreover, only add-ins registered for the application you are working in can be registered. For example, if you are working in Microsoft® Access, you cannot add a COM add-in that is registered only for Microsoft® Word and Microsoft® Excel. In addition, you can

create COM add-ins for the Microsoft® Visual Basic® Editor. Loading and unloading a COM add-in for the Visual Basic Editor is slightly different from doing so for COM add-ins in the host application's user interface.

Viewing Available Application-Specific Add-ins

Application-specific add-ins appears in various dialog boxes depending on which application you are using. In Microsoft® Word, this is the Templates and Add-Ins dialog box; in Microsoft® Excel and Microsoft® PowerPoint®, it is the Add-Ins dialog box; in Microsoft® Access, it is the Add-In Manager.

Each dialog box has buttons to add or remove add-ins from the list of application-specific add-ins, and a check box to indicate whether the add-in is loaded. As with COM add-ins, the application-specific add-in must be loaded into memory before it can be used.

Building COM Add-ins for the Visual Basic Editor

By creating COM add-ins for the Microsoft® Visual Basic® Editor, you can customize your development environment and work with components in a Visual Basic for Applications (VBA) project from code. For example, you can build a code wizard that walks a programmer through a series of steps and then builds a procedure, or you can build a code analyzer that determines how many times and from where a procedure is called.

When you create a COM add-in for the Visual Basic Editor, it appears in all instances of the Visual Basic Editor. You cannot, for example, create a COM add-in that appears only in the Visual Basic Editor in Microsoft® Word; it will also appear in the Visual Basic Editor in Microsoft® Access, Microsoft® Excel, Microsoft® PowerPoint®, Microsoft® FrontPage®, and any other VBA host applications on the computer where the COM add-in DLL is registered.

Note also that you can create multiple add-ins in a single DLL. Each add-in designer in the Add-in project represents a separate add-in. For example, you can create a single DLL that contains a suite of add-ins for developers, and the developers can load just the add-ins they want to use.

To control the Visual Basic Editor from the code inside an add-in, you use the Microsoft Visual Basic for Applications Extensibility library. This object library contains objects that represent the parts of a VBA project, such as the VBProject object and the VBComponent object. The top-level object in the VBA Extensibility library object model is the VBE object, which represents the Visual Basic Editor itself. For more information about the object model, use the Object Browser.

Note
Do not confuse the VBA Extensibility library with the IDTExtensibility2 library. Although their names are similar, the VBA Extensibility library provides objects that you can use to work with the Visual Basic Editor from an add-in while it is running, and the IDTExtensibility2 library provides events that are triggered when the add-in is connected or disconnected. In addition, do not confuse the VBA Extensibility library with the Microsoft Visual Basic 6.0 Extensibility library, which is used for creating add-ins in Microsoft Visual Basic.

Creating COM Add-ins in Office Developer

You can create your own COM add-ins in Microsoft®Visual Basic® for Applications (VBA) with Microsoft® Office XP Developer. You do not require external development tools, such as Microsoft® Visual C++® or Microsoft® Visual Basic®, to create COM add-ins.

You can use Add-In Designers to create COM add-ins for use in VBA or any Office application. For example, you might create an add-in tool to format and print code that could be shared with other developers, or you might create an add-in for Microsoft® Excel to calculate tax rates that could be shared with Office users.

COM add-ins created with Office Developer are packaged as dynamic link libraries (DLL files) and are registered so that they can be loaded by Office XP applications.

To add an Add-In Designer to your project

1. From the **File** menu, select **New Project** .
2. In the **New Project** dialog box, select **Add-In Project.**
3. An Add-In Designer will be added to your project.

The Add-In Designer provides several properties that can be set to define the attributes of your add-in, including Name, Description, and Load Behavior. It also provides several events that can be used to add code, such as OnConnection, OnStartupComplete, and OnDisconnection.

The actual code for your COM add-in depends on what you want the add-in to do, as well as which application the add-in is for. Each of the applications that can use COM add-ins exposes its extensibility structure using its object model; you can view the object model for your particular application in the Object Browser.

To package the COM add-in as a DLL in VBA

After you have written and debugged your code, you can make your add-in into a DLL.

- From the **File** menu, select **Make projectname.dll.**

Note

This will create the COM add-in, add the appropriate registry entries, and make the COM add-in available for use in your Office host.

Creating a COM Add-in for the Visual Basic Editor

For the most part, creating a COM add-in for the Microsoft® Visual Basic® Editor is similar to creating one for a Microsoft®Office XP application. COM add-ins for the Visual Basic Editor also includes the add-in designer or a class module that implements the IDTExtensibility2 library.

One key difference to note is that the initial load behavior setting for a COM add-in for the Visual Basic Editor differs from that of a COM add-in for an Office application. A COM add-in for the Visual Basic Editor can have one of two initial load behaviors: None, meaning that the add-in is not loaded until the user loads it, or startup, meaning that the add-in is loaded when the user opens the Visual Basic Editor.

To create a COM add-in using Visual Basic for Applications in the Visual Basic Editor

1. Create a new Add-in project, select **Visual Basic for Applications IDE** in the **Application** box, and then select **VBE 6.0** in the **Application Version** box. Set the initial load behavior for the add-in to either **None** or **Startup.**
2. Set a reference to the **Microsoft Visual Basic for Applications Extensibility 5.3** library in file Vbe6ext.olb.
   Note
   If the object library does not appear in the list of available references, you can browse for it in C:\Program Files\Common Files\Microsoft Shared\VBA\VBA6, the default installation directory. The name of the library as it appears in the Object Browser is VBIDE.
3. The **OnConnection** event procedure passes in the Application argument, which contains a reference to the instance of the Visual Basic Editor in which the add-in is running. You can use this object to work with all other objects in the VBA Extensibility library. Create a public module-level object variable of type **VBIDE.VBE** , and assign the object referenced by the Application argument to this variable.
4. Within the OnConnection event procedure, you can optionally include code to hook the add-in's form up to a command bar control in the Visual Basic Editor. You can work with the Visual Basic Editor's command bars by using the **CommandBars** property of the **VBE** object.
5. Build any forms or other components to be included in the project.
6. Place a breakpoint in the **OnConnection** event procedure, and then select **Run Project** from the **Run** menu.
7. In a Visual Basic for Applications (VBA) host application, such as Microsoft® Excel, open the Visual Basic Editor, select **Add-In Manager** on the **Add-Ins** menu, and select your add-in from the list. Select the **Loaded/Unloaded** check box to load the add-in, if it is not set to load on startup.
8. Debug the add-in. When you have debugged it to your satisfaction, chose Stop Project from the Run menu, end the running project, and make the add-in's DLL by clicking **Make projectname.dll** on the **File** menu.

Note
To test and debug the add-in, you must open another instance of the Visual Basic Editor to see it. The add-in does not appear in the instance of the editor that you are using to create it.

You can use the same strategies to distribute COM add-ins for the Visual Basic Editor as you use to distribute COM add-ins for the Office XP applications.

Working with the Microsoft Visual Basic for Applications Extensibility Library

The Microsoft® Visual Basic® for Applications (VBA) extensibility library provides objects that you can use to work with the Visual Basic Editor and any VBA projects that it contains. From an add-in created in Visual Basic 6.0, you can return a reference to the VBE object, the top-level object in the VBA Extensibility library, through the Application argument of the OnConnection event procedure. This argument provides a reference to the instance of the Visual Basic Editor in which the add-in is running.

The VBProject object refers to a VBA project that is open in the Visual Basic Editor. A VBProject object has a VBComponents collection, which in turn contains VBComponent objects. A VBComponent object represents a component in the project, such as a standard module, class module, or form. Because a VBComponent object can represent any of these objects, you can use its Type property to determine which type of module you are currently working with.

For example, suppose you have a variable named vbeCurrent, of type VBIDE.VBE, which represents the instance of the Visual Basic Editor in which the add-in will run. The following code fragment prints the names and types of all components in the active project to the Immediate window:

```
Dim vbcComp As VBIDE.VBComponent
For Each vbcComp In vbeCurrent.ActiveVBProject.VBComponents
  Debug.Print vbcComp.Name, vbcComp.Type
Next vbcComp
```

A VBComponent object has a CodeModule property that returns a CodeModule object, which refers to the code module associated with that component. You can use the methods and properties of the CodeModule object to manipulate the code in that module on a line-by-line basis. For example, you can insert lines by using the InsertLines method, or perform find and replace operations by using the Find and Replace methods.

To work with command bars in the Visual Basic Editor, use the CommandBars property of the VBE object to return a reference to the CommandBars collection.

For more information about working with the VBA Extensibility library, search the Visual Basic Reference Help index for "VBProject object."

Building COM Add-ins for Office Applications

Because Microsoft® Office XP applications support the Component Object Model (COM) add-in architecture, you can use the same tools and installation file formats (a Microsoft® ActiveX® .dll or .exe) to develop add-ins for all Office applications. By building COM add-ins, you can extend the functionality of your Office-based applications without adding complexity for the user.

You can also create add-ins for Office Developer and for the Microsoft®Visual Basic® Editor. You can make such add-ins available to or from any application that supports Visual Basic for Applications (VBA), including applications other than Office.

In This Section

Working with Add-in Designers
Create and register your COM add-in with an add-in designer.

Specifying Load Behavior
Load (connect) the add-in, and make it available to the user; or unload (disconnect) the add-in, so it cannot be run.

Writing Code in the Add-in Designer
Begin writing code in the designer's class module when you have specified general information for a COM add-in in the add-in designer.

Hooking a COM Add-in Up to a Command Bar Control
Integrate your COM add-in (if it has a user interface) with the host application in some way, so the user can interact with it.

Debugging a COM Add-in
Load and use the COM add-in from within a Microsoft® Office XP application to test and debug it.

Making the DLL
Turn your COM add-in into a DLL when you have finished debugging it.

Distributing COM Add-ins
Install all the files necessary to distributing your COM add-in to other users on each user's system and register the add-in.

COM Add-ins and Security
Specify security settings for Microsoft® Office XP applications in the Office XP Security dialog box.

Working with Add-in Designers

An add-in designer is a file included with the template project that helps you create and register your COM add-in. You can create a COM add-in without including an add-in designer, but the add-in designer simplifies the process of creating and registering the add-in. You can use an add-in designer to specify important information for your COM add-in: its name and description, the application in which it is to run, and how it loads in that application.

Similar to forms in a Visual Basic project, an add-in designer (shown in the following figure) has a user interface component and an associated class module. The user interface component is never visible to the user when the add-in is running, however; it is visible only to the developer at design time. You can think of the add-in designer as a sort of dialog box where you specify settings for an add-in.

The Add-in Designer (Example)

The class module contains the events that occur when the add-in is loaded or unloaded. You can use these events to integrate the add-in into the application.

When you create the add-in DLL, Visual Basic 6.0 uses the information you have given to the add-in designer to properly register the DLL as a COM add-in. Visual Basic 6.0 writes the add-in's name, description, and initial load behavior setting to the registry. The add-in's host application reads these registry entries and loads the add-in accordingly.

Creating COM Add-ins for Multiple Applications

Each add-in designer in your project creates an add-in that can run in only one application. To create a COM add-in that is available to more than one application, you create a new add-in designer for each application that you want to use the add-in and then customize the add-in designer for each application.

For example, suppose that you want to create an add-in for Microsoft® Word and Microsoft® PowerPoint® that creates an organizational chart from a table in a database and inserts the chart into the document or slide. You would begin by making sure there is an add-in designer for Word and one for PowerPoint.

To create a new Add-in project

1. From the **File** menu in Microsoft®Visual Basic® Editor, click **New Project** , and then click **Add-in Project** . The first add-in designer in the Add-in project appears.
2. Change the add-in designer's **Name** property setting in the **Properties** window. It might be helpful to indicate in the name of the add-in designer which designer goes with which application. For example, the add-in designer for Microsoft®Excel in the Image Gallery project is named dsrImageExcel.
3. Enter the appropriate information in the **General** and **Advanced** tabs of the add-in designer. Select the application that you want the add-in designer to work with on the **General** tab. For details, see "Configuring an Add-in Designer."
4. To add code to the add-in designer, open the **View** menu, and click **Code** .

To add another add-in designer to the Add-in project

1. Open the **Insert** menu, and click **Components** .
2. On the **Designers** tab, select **Addin class** , and click **OK** .
3. Open the **Insert** menu, and click **Addin Class** .

To make a DLL file for the Add-in project

1. Save the Add-in project.
2. Open the **Fil** menu, and click **Make projectname.DLL** (the default is AddInProject1.DLL).
3. In the **Make Project** dialog box, select the desired DLL name and location.
4. Click the **Options** button to open the **Project Properties** dialog box if you want to assign a specific version number, add version information, or specify a DLL base address. After you have entered the information, click **OK** to close the Project Properties dialog box.
5. Click **OK** on the **Make Project** dialog box to make the DLL file.

Configuring an Add-in Designer

To create your add-in, you first must fill out the options on the General tab of the add-in designer. The following table explains each option.

Option	Description
Addin Display Name	The name that will appear in the COM Add-ins dialog box in a Microsoft® Office XP application. The name you supply should be descriptive to the user. If the name is to come from a resource file specified in the Satellite DLL Name box on the Advanced tab, it must begin with a number sign (#), followed by an integer specifying a resource ID within the file.
Addin Description	Descriptive text for a COM add-in, available from Microsoft® Visual Basic® for Applications (VBA) in the Description property of the COMAddIn object. If the description is to come from a resource file specified in the Satellite DLL Name box on the Advanced tab, it must begin with a number sign (#), followed by an integer specifying a resource ID within the file.
Application	The application in which the add-in will run. This list displays applications that support COM add-ins.
Application Version	The version of the application in which the add-in will run.
Initial Load Behavior	The way that the add-in loads in the application. The list of possible settings comes from the registry. Common used behaviors include Startup and On Demand.
Addin is command-line safe (does not put up any UI)	Does not apply to COM add-ins running in Office XP applications.

The Advanced tab of the add-in designer makes it possible for you to specify a file containing localized resource information for the add-in and to specify additional registry data. The following table describes the options available on the Advanced tab.

Option	Description
Satellite DLL Name	The name of a file containing localized (translated) resources for an add-in; the file must be located in the same directory as the add-in's registered DLL.
Registry Key for Additional Add-in Data	The registry subkey to which additional data is to be written.
Add-in Specific Data	The names and values to be stored in the registry subkey. Only String and DWORD type values are permitted.

Working with Host Application Object Models

There are a few things to keep in mind as you add forms and other components to your COM add-in. First, your COM add-in is similar to a separate application running inside a Microsoft® Office XP application. Therefore, you must set references to any object libraries you want to work with from within the COM Add-in project. If your add-in will be run in more than one application, you can use the OnConnection event procedure to determine which application your add-in is currently running in and then selectively run code that works with that application's objects.

To figure out which application the add-in is currently running, use the object supplied by the Application argument of the OnConnection event procedure. Assign this object variable to a global object variable. In the code that interacts with the host application, check to see which application you are working with, and use that application's object model to perform the task.

A DLL is loaded into memory only once, but each application that accesses the DLL gets its own copy of the DLL's data, stored in a separate space in memory. Therefore, you can use global variables in a COM add-in without worrying about data being shared between two applications that are using the COM add-in at the same time. For example, the Image Gallery sample add-in can run simultaneously in Microsoft® Word, Microsoft® Excel, and Microsoft® PowerPoint®. When Word loads the add-in, the OnConnection event occurs and a reference to the Word Application object is stored in a global variable of type Object. If Excel then loads the add-in, the OnConnection event occurs and a reference to the Excel Application object is stored in a global variable of type Object but in a different space in memory. Within the code for the add-in, you can use the If TypeOf...End If construct to check to which application's Application object the variable points.

```
' Global object variable, declared in modSharedCode module.
Public gobjAppInstance As Object
Private Sub cmdInsert_Click()
  ' Insert selected image.  Check which object variable has been initialized.
  If TypeOf gobjAppInstance Is Word.Application Then
    ' Insert into Word.
    Word.Selection.InlineShapes.AddPicture FileName:=  img(mlngSel).Tag, LinkToFile:=False, SaveWithDocument:=True
  ElseIf TypeOf gobjAppInstance Is Excel.Application Then
    gobjAppInstance.ActiveSheet.Pictures.Insert img(mlngSel).Tag
  ElseIf TypeOf gobjAppInstance Is Powerpoint.Application Then
    gobjAppInstance.ActiveWindow.Selection.SlideRange.Shapes.AddPicture _
      FileName:=img(mlngSel).Tag, LinkToFile:=msoFalse,  SaveWithDocument:=msoCTrue, Left:=100, Top:=100
  End If
End Sub
```

Specifying Load Behavior

When a COM add-in has been properly registered, it is available to whatever applications are specified in the add-in designers that the project contains. The registered COM add-in display name appears in the COM Add-in dialog box; if it does not, click Add to browse for the add-in and add it to the list.

Selecting the check box next to an add-in in the COM Add-ins dialog box loads (connects) the add-in and makes it available to the user; clearing the check box unloads (disconnects) the add-in, and it cannot be run.

As the developer, you specify the default setting for when a COM add-in should be loaded. You do this in the Initial Load Behavior list in the add-in designer.

Note
Users can change this setting later by using the Add-in Manager.

You can specify that an add-in be loaded in one of the following ways:

- Only when the user loads it in the COM Add-ins dialog box, or when Microsoft®Visual Basic®for Applications (VBA) code loads it by setting the Connect property of the corresponding COMAddIn object.
- Every time the application starts.
- The first time the application starts, so that it can create a toolbar button or menu item for itself. After that, the add-in is loaded only when the user requests it by clicking the menu item or button.

The following table describes the different settings for the Initial Load Behavior setting.

Initial Load Behavior setting	**Behavior**
None	The COM add-in is not loaded when the application boots. It can be loaded in the COM Add-ins dialog box or by setting the Connect property of the corresponding COMAddIn object.
Startup	The add-in is loaded when the application boots. When the add-in is loaded, it remains loaded until it is explicitly unloaded.

Load on Demand [1]	The add-in is not loaded until the user clicks the button or menu item that loads the add-in, or until a procedure sets its Connect property to True. In most cases, you will not set the initial load behavior to Load on Demand directly; you will set it to Load at Next Startup Only, and it will be set automatically to Load on Demand on subsequent boots of the host application.
Load at Next Startup Only [1]	After the COM add-in has been registered, it loads as soon as the user runs the host application for the first time. The next time the user boots the application, the add-in is loaded on demand-that is, it does not load until the user clicks the button or menu item associated with the add-in, or through the COM Add-in dialog box.
Command line/Startup	Add-in loads either when specifically invoked from a command-line parameter, when Visual Basic starts.
Command line	Add-in loads only when specifically invoked from a command-line parameter.

1 Not available to add-ins developed using VBA.

Writing Code in the Add-in Designer

After you have specified general information for a COM add-in in the add-in designer, you can begin writing code in the designer's class module. To view the add-in designer's class module, right-click the add-in designer in the Project Explorer, and then click View Code on the shortcut menu.

Code that is in the add-in designer handles the add-in's integration with the host application. For example, code that runs when the add-in is loaded or unloaded resides in the add-in designer's module. If the add-in contains forms, the add-in designer might contain code to display the forms.

Implementing the IDTExtensibility2 Library

A COM add-in has events that you can use to run code when the add-in is loaded or unloaded, or when the host application has finished starting up or is beginning to shut down. To use these events, you must implement the IDTExtensibility2 library, which provides a programming interface for integrating COM add-ins with their host applications. When you implement the IDTExtensibility2 library within a class module, the library makes a set of new events available to the module. You must have these events to control your COM add-in.

Using the Add-in project in Microsoft® Visual Basic® for Applications (VBA) implements the IDTExtensibility2 library for you in the add-in designer's class module. If you are creating the COM add-in from scratch in Visual Basic 6.0, use the following procedure:

To manually implement the IDTExtensibility2 library in Visual Basic 6.0

1. Set a reference to the library by clicking **References** on the **Project** menu and then selecting the check box next to **Microsoft Add-in Designer** . If this library does not appear in the list, you can add it by clicking **Browse** and finding the file Msaddndr.dll. By default, this file is located in the C:\Program Files\Common Files \Designer subfolder.
2. In the Declarations section of the add-in designer's class module, add the following code:
   Implements IDTExtensibility2
3. In the Code window, click **IDTExtensibility2** in the **Object** box. This adds the template for the procedure to the **OnConnection** event.
4. Create event procedure templates for the four remaining event procedures by clicking them in the **Procedure** dialog box in the **Code Window.**
5. Add code or a comment to each of the five event procedures.

Note
You must include the event-procedure template for each event provided by the IDTExtensibility2 interface. If you omit any of the event procedures, your project will not compile. If you are not adding code to an event-procedure template, it is a good idea to add a comment; a single apostrophe (') is sufficient.

Working with the IDTExtensibility2 Event Procedures

The IDTExtensibility2 library provides five events that you can use to manipulate your add-in and the host application: OnConnection, OnDisconnection, OnAddInsUpdate, OnStartupComplete, and OnBeginShutdown. The following sections describe each of these event procedures.

The OnConnection Event

The OnConnection event occurs when the COM add-in is loaded (connected). An add-in can be loaded in one of the following ways:

- The user starts the host application and the add-in's load behavior is specified to load when the application starts.
- The user loads the add-in in the COM Add-ins dialog box.
- The Connect property of the corresponding COMAddIn object is set to True. For more information about the COMAddIn object, search the Microsoft® Office Visual Basic Reference Help index for "COMAddIn object."

The OnConnection event procedure takes four arguments, described in the following table.

Argument	Type	Description
Application	Object	Provides a reference to the application in which the COM add-in is currently running.
ConnectMode	Custom Long	A constant that specifies how the add-in was loaded.
AddInInst	Object	A COMAddIn object that refers to the instance of the class module in which code is currently running. You can use this argument to return the programmatic identifier for the add-in.
Custom()	Variant	An array of Variant type values that provides additional data. The numeric value of the first element in this array indicates how the host application was started: from the user interface (1), by embedding a document created in the host application in another application (2), or through Automation (3).

The constants for the ConnectMode argument are grouped in the ext_ConnectMode enumeration. These constants are described in the following table.

Constant	Description
ext_cm_AfterStartup	Add-in was loaded after the application started, or by setting the Connect property of the corresponding COMAddIn object to True.
ext_cm_External	Does not apply to building COM add-ins for Microsoft® Office XP applications.
ext_cm_Startup	Add-in was loaded on startup.

If you are building a COM add-in that will run in more than one host application, you might find that you call the same code from each add-in designer's OnConnection event. For example, you might create a new command bar button in the OnConnection event procedure in the same way within each add-in designer. If so, it is more efficient to create a public procedure in a standard module and call it from within the OnConnection event procedure for each add-in designer than to include the same code in each add-in designer.

The following example shows the OnConnection event procedure. The OnConnection event procedure calls the CreateAddInCommandBarButton procedure in the modSharedCode module. This procedure creates a new command bar button and returns a reference to it. The OnConnection event procedure then assigns this reference to a private event-ready variable of type CommandBarButton.

```
' Event-ready variable declared in add-in designer's module.
Private WithEvents p_ctlBtnEvents   As Office.CommandBarButton

Private Sub IDTExtensibility2_OnConnection(ByVal Application As Object,
   ByVal ConnectMode As AddInDesignerObjects.ext_ConnectMode,  ByVal AddInInst As Object,  custom() As Variant)
   ' Call shared code to create new command bar button    and return a reference to it. Assign reference to event-ready
CommandBarButton object declared with WithEvents within this module.
   Set p_ctlBtnEvents = CreateAddInCommandBarButton(Application, ConnectMode, AddInInst)
End Sub

' Public function in modSharedCode module.
Public Function CreateAddInCommandBarButton(ByVal Application As Object, _
                ByVal ConnectMode As AddInDesignerObjects.ext_ConnectMode,
                ByVal AddInInst As Object) As Office.CommandBarButton
   ' This procedure assigns a reference to the Application  object passed to the OnConnection event to a global object variable.
It then creates a new command bar  button and returns a reference to the button to the OnConnection event procedure. The
advantage to putting this code in a public module is that if you have more than one add-in designer in the project, you can call
' this procedure from each of them rather than duplicating the code.
   Dim cbrMenu             As Office.CommandBar
   Dim ctlBtnAddIn          As Office.CommandBarButton

   On Error GoTo CreateAddInCommandBarButton_Err
   ' Return reference to Application object and store it  in public variable so that other procedures in add-in can use it.
   Set gobjAppInstance = Application
   ' Return reference to command bar.
   Set cbrMenu = gobjAppInstance.CommandBars(CBR_NAME)
   ' Add button to call add-in from command bar, if it doesn't  already exist.
   ' Constants are declared at module level. Look for button on command bar.
   Set ctlBtnAddIn = cbrMenu.FindControl(Tag:=CTL_KEY)
```

```
  If ctlBtnAddIn Is Nothing Then
    ' Add new button.
    Set ctlBtnAddIn = cbrMenu.Controls.Add(Type:=msoControlButton, Parameter:=CTL_KEY)
    ' Set button's Caption, Tag, Style, and OnAction properties.
    With ctlBtnAddIn
      .Caption = CTL_CAPTION
      .Tag = CTL_KEY
      .Style = msoButtonCaption
      ' Use AddInInst argument to return reference to this add-in.
      .OnAction = PROG_ID_START & AddInInst.ProgId  & PROG_ID_END
    End With
  End If
  ' Return reference to new commandbar button.
  Set CreateAddInCommandBarButton = ctlBtnAddIn

 CreateAddInCommandBarButton_End:
  Exit Function
CreateAddInCommandBarButton_Err:
  ' Call generic error handler for add-in.
  AddInErr Err
  Resume CreateAddInCommandBarButton_End
End Function
```

The CreateAddInCommandBarButton procedure first performs a critical step: it assigns the object passed to the procedure in the Application argument to a public module-level object variable. This object variable persists as long as the COM add-in is loaded, so any other procedures in the module can determine in what application the add-in is currently running.

A public module-level variable declared in a standard module in a COM add-in remains in existence from the time the add-in is loaded to the time it is unloaded.

This procedure also contains code that creates a new menu item on the Tools menu of the host application the first time the add-in is loaded. Before creating the new menu item, the procedure checks to see whether the item already exists. If the item does exist, the procedure returns a reference to the existing menu item rather than creating a new one. The OnConnection event procedure then assigns the reference returned by the CreateAddInCommandBarButton procedure to a variable (p_ctlBtnEvents) that has been declared by using the WithEvents keyword, so that the menu item's Click event procedure will be triggered when the user clicks the new menu item.

The OnDisconnection Event

The OnDisconnection event occurs when the COM add-in is unloaded. You can use the OnDisconnection event procedure to run code that restores any changes made to the application by the add-in and to perform general clean-up operations.

An add-in can be unloaded in one of the following ways:

- The user clears the check box next to the add-in in the COM Add-ins dialog box.
- The host application closes. If the add-in is loaded when the application closes, it is unloaded. If the add-in's load behavior is set to Startup, it is reloaded when the application starts again.
- The Connect property of the corresponding COMAddIn object is set to False.

The OnDisconnection event procedure takes two arguments, described in the following table.

Argument	Type	Description
RemoveMode	Custom Long	A constant that specifies how the add-in was unloaded.
custom()	Variant	An array of Variant type values that provides additional data. The numeric value of the first element in this array indicates how the host application was started: from the user interface (1); by embedding a document created in the host application in another application (2); or through Automation (3).

The following table lists the available constants for the RemoveMode method, which are grouped in the ext_DisconnectionMode enumeration.

Constant	Description
ext_dm_HostShutdown	Add-in was unloaded when the application was closed.
ext_dm_UserClosed	Add-in was unloaded when the user cleared the corresponding check box in the COM Add-ins dialog box or when the Connect property of the corresponding COMAddIn object was set to False.

The following code shows the OnDisconnection event procedure that calls the RemoveAddInCommandBarButton procedure located in the modSharedCode module. If the user unloads the add-in, the add-in's menu command is deleted; otherwise, it is maintained for the next time the user starts the application:

```
Private Sub IDTExtensibility2_OnDisconnection(ByVal _
  RemoveMode As AddInDesignerObjects.ext_DisconnectMode,  custom() As Variant)
  ' Call common procedure to disconnect add-in.
  RemoveAddInCommandBarButton RemoveMode
End Sub

Function RemoveAddInCommandBarButton(ByVal RemoveMode As AddInDesignerObjects.ext_DisconnectMode)
  ' This procedure removes the command bar button for  the add-in if the user disconnected it.
  On Error GoTo RemoveAddInCommandBarButton_Err
  ' If user unloaded add-in, remove button. Otherwise,  add-in is being unloaded because application is closing; in that case,_
  ‘ leave button as is.
  If RemoveMode = ext_dm_UserClosed Then
    On Error Resume Next
    ' Delete custom command bar button.
    gobjAppInstance.CommandBars(CBR_NAME).Controls(CTL_NAME).Delete
    On Error GoTo RemoveAddInCommandBarButton_Err
  End If

RemoveAddInCommandBarButton_End:
  Exit Function
RemoveAddInCommandBarButton_Err:
  AddInErr Err
  Resume RemoveAddInCommandBarButton_End

End Function
```

The OnStartupComplete Event

The OnStartupComplete event occurs when the host application completes its startup routines, in the case where the COM add-in loads at startup. If the add-in is not loaded when the application loads, the OnStartupComplete event does not occur-even when the user loads the add-in in the COM Add-ins dialog box. When this event does occur, it occurs after the OnConnection event.

You can use the OnStartupComplete event procedure to run code that interacts with the application and that should not be run until the application has finished loading. For example, if you want to display a form that gives users a choice of documents to create when they start the application, you can put that code in the OnStartupComplete event procedure.

The OnBeginShutdown Event

The OnBeginShutdown event occurs when the host application begins its shutdown routines, in the case where the application closes while the COM add-in is still loaded. If the add-in is not loaded when the application closes, the OnBeginShutdown event does not occur. When this event does occur, it occurs before the OnDisconnection event.

You can use the OnBeginShutdown event procedure to run code when the user closes the application. For example, you can run code that saves form data to a file.

The OnAddInsUpdate Event

The OnAddInsUpdate event occurs when the set of loaded COM add-ins changes. When an add-in is loaded or unloaded, the OnAddInsUpdate event occurs in any other loaded add-ins. For example, if add-ins A and B both are loaded currently, and then add-in C is loaded, the OnAddInsUpdate event occurs in add-ins A and B. If C is unloaded, the OnAddInsUpdate event occurs again in add-ins A and B.

If you have an add-in that depends on another add-in, you can use the OnAddInsUpdate event procedure in the dependent add-in to determine whether the other add-in has been loaded or unloaded.

Note
The OnStartupComplete, OnBeginShutdown, and OnAddInsUpdate event procedures each provide only a single argument, the Custom() argument, which is an empty array of Variant type values. This argument is ignored in COM add-ins for Office XP applications.

Hooking a COM Add-in Up to a Command Bar Control

If your COM add-in has a user interface, it must be integrated with the host application in some way, so the user can interact with it. For example, the user interface for your COM add-in most likely includes a form. At some point, code in the add-in must be run to display the form.

One way to integrate your add-in with an application's user interface is to include code in the OnStartupComplete event procedure that creates a new command bar control (toolbar button or menu item) in the host application. When your add-in is loaded, the user can click the button or menu item to work with the add-in. You can use the OnConnection event procedure, but it does not guarantee that the command bar object has been loaded.

Similarly, you can add code to unload your add-in in the OnBeginShutdown event procedure or the OnDisconnection event procedure.

The critical aspect of integrating an add-in through a command bar control is the process of setting up the event sink. You must create a command bar control that is event-ready, so its Click event is triggered when the user clicks the control. You can use the WithEvents keyword to create an event-ready command bar control.

If you set the load behavior for your add-in to Load at Next Startup Only, you also must set the OnAction property for the command bar control. If you do not set the OnAction property, the add-in will load the first time the application starts. The next time you start the application, however, the load behavior for the add-in will be set to Load on Demand, and the command bar control that you have created for the add-in will not load the add-in unless the OnAction property has been set.

Even if your add-in is not demand-loaded, it is a good idea to set this property in your code, in case you later change the load behavior for the add-in. The syntax for setting the OnAction property for a COM add-in is:

ctlButton.OnAction = "!<ProgID>"

where ctlButton is the CommandBarButton object and ProgID is the programmatic identifier for the add-in. The programmatic identifier is the sub key that is created for the add-in in the Microsoft® Windows® registry. Each add-in designer or class module that implements the IDTExtensibility2 library in the COM Add-in project adds its own programmatic identifier to the registry, beneath the AddIns sub key for the host application in which it will run. The programmatic identifier for a COM add-in consists of the name of the project followed by the name of the add-in designer or class module. For example, the programmatic identifier for the ImageGallery add-in for Microsoft® Word is ImageGallery.dsrImageWord.

To return the programmatic identifier for an add-in, you can use the AddInInst argument that is passed to the OnConnection event procedure. This argument provides a reference to the add-in designer or class module in which code is running currently. The AddInInst argument is an object of type COMAddIn, which has a ProgId property that returns the programmatic identifier. Note that you must concatenate the !< and > delimiters before and after the programmatic identifier string to properly set the OnAction property.

Note
If your add-in will run in Word, you also must set the Tag property for the CommandBarButton object to a unique String value. This makes sure the command bar button will respond to the Click event and load the add-in for each new document window that the user opens. Because the Tag property provides you with additional information about the control, it is a good idea to set the Tag property for a command bar button that loads a COM add-in in any host application.

Creating a Command Bar Control

In some cases, you might want to provide access to your add-in through a menu command.

To create a command bar control that displays the add-in's form

1. In the add-in designer's module, use the **WithEvents** keyword to declare a module-level variable of type **CommandBarButton.** This creates an event-ready **CommandBarButton** object.
2. In the same module, create the **Click** event procedure template for the **CommandBarButton** object by clicking the name of the object variable in the **Object** box and then clicking **Click** in the **Procedure** dialog box.
3. Write code within the event-procedure template to open the form when the **Click** event occurs.
4. In the **OnConnection** event procedure, check to see whether the command bar control already exists, and return a reference to it if it does. If it does not exist, create the new command bar control, and return a reference to it. You must check whether the command bar control exists, so you do not create a new control each time your code runs.
5. When you create the new command bar control, set the **Tag** property for the **CommandBarButton** object to a unique string. This is necessary only for COM add-ins running in Microsoft® Word, but it is recommended for COM add-ins running in any host application.
6. When you create the new command bar control, set the **OnAction** property for the command bar control if the COM add-in is to be demand-loaded. If you fail to set the **OnAction** property, the command bar button will load the add-in the first time the application starts, but it will not load the add-in when the application is closed and reopened.
7. Within the **OnConnection** event procedure, assign the reference to the command bar control to the event-ready **CommandBarButton** object variable.
8. Add code to the **OnDisconnection** event to remove the command bar control when the add-in is unloaded.

Note
The add-in designer in the COM add-in template project includes code that performs all these steps to create a menu item on the **Tools** menu. By default, the template project has a reference set to the Microsoft® Office XP object library, so you can work with Office command bars.

Debugging a COM Add-in

When you are developing a COM add-in in Microsoft® Visual Basic® for Applications (VBA), you can debug the add-in by putting the project into run mode. With the project in run mode, you can load and use the COM add-in from within a Microsoft® Office XP application to test and debug it by using any of the Visual Basic debugging tools.

To debug a COM add-in in the Visual Basic Editor

1. Open the Add-in project in **Visual Basic Editor**
2. Place any desired breakpoints, **Stop** statements, or watches in the code.
3. On the **Run** menu, click **Run Project.** This compiles your project, alerting you to any compilation errors, and then puts the project into run mode.
4. Open the intended host application for the COM add-in. If you have set the add-in's load behavior to **Startup** or **Load at Next Startup Only** , the add-in loads as soon as you start the application. If the add-in's load behavior is set to **None** or **Load on Demand** , open the COM Add-ins dialog box, and select the check box next to your add-in to load it.

When the add-in loads, the OnConnection event occurs. You can now enter break mode in the Add-in project in the Visual Basic Editor and debug the code.

Making the DLL

After debugging your COM add-in to your satisfaction, you can package it as a DLL. If you created your COM add-in using Microsoft® Office XP Developer, it is already a .dll. However, if you created it in Microsoft® Visual Basic® Editor, you must create the .dll. To create the .dll in Visual Basic, click Make projectname.dll on the File menu. The Make Project dialog box appears; note that you can enter a name for the DLL that is different from the suggested name. The process of making the DLL registers it on the local machine.

When you make the DLL in the Visual Basic Editor, the information in the add-in designer is used to add a sub key to the Windows registry, indicating which applications can host the add-in. The COM add-in then appears in the COM Add-ins dialog box in those applications for which it is registered.

Add-in Registration

Before you can use a COM add-in in a Microsoft® Office XP application, the add-in DLL must be registered, just as any other DLL on the computer. The DLL's class ID is registered beneath the \HKEY_CLASSES_ROOT subtree in the registry. The DLL can be registered on a user's computer by using a setup program, such as those created by the Packaging Wizard or by running the Regsvr32.exe command-line utility that is included with Microsoft® Windows®. Adding a COM add-in by using the COM Add-ins dialog box also registers the DLL-if it was created with Microsoft® Visual Basic® 6.0.

Registering the DLL beneath the \HKEY_CLASSES_ROOT subtree informs the operating system of its presence, but additional information must be added to the registry for the add-in to be available to an Office XP application. This is the information that you can specify in the add-in designer-the add-in's name, description, target application, target application version, and initial load behavior. The add-in designer makes sure this application-specific information is written to the correct place in the registry at the same time that the add-in DLL is registered. The COM Add-ins dialog box displays the information contained in the subkey for the corresponding Office XP application.

This subkey must be added to the following registry subkey, where appname is the name of the application in which the add-in will run:

\HKEY_CURRENT_USER\SOFTWARE\Microsoft\Office\appname\AddIns

The new subkey itself must be the programmatic identifier of the COM add-in, which consists of the name of the project followed by the name of the class module or add-in designer. For example, the registry subkey for the Image Gallery add-in for Microsoft® Word would be ImageGallery.dsrImageWord.

The following table describes the entries that you can add beneath this subkey. Only the LoadBehavior entry is required; the others are optional.

Name	Type	Value
Description	String	Name to appear in COM Add-ins dialog box
FriendlyName	String	String returned by Description property
LoadBehavior	DWORD	Integer indicating load behavior: 0 (None), 3 (Startup), 9 (Load on Demand), or 16 (Load At Next Startup Only)

Distributing COM Add-ins

If you are planning to distribute your COM add-in to other users, you must install all the necessary files on each user's system and register the add-in. How you do this depends on the environment in which you are developing the add-in.

Distributing COM Add-ins Created with Office Developer

If you are developing in Microsoft® Office XP Developer, the easiest way to distribute a COM add-in is to create a setup program for the add-in. The user can install and register the add-in by running the setup program.

Before you can create the setup program, you must compile the COM Add-in Project to a DLL.

To create the setup program, run the Packaging Wizard on the Add-in project, which was compiled to DLL. The Packaging Wizard will create a setup program that installs and registers the add-in DLL and any other necessary files but not the code.

Distributing COM Add-ins Created with Visual Basic 6.0

If you are developing in Microsoft® Visual Basic® 6.0, the easiest way to distribute a COM add-in is to include the add-in designer in the Add-in project and then create a setup program for the add-in. The user can install and register the add-in by running the setup program.

To create the setup program, run the Visual Basic 6.0 Package and Deployment Wizard on the Add-in project. When the user runs the setup program, all the files required for the add-in to run will be copied to the user's computer and registered.

For more information about using the Visual Basic 6.0 Package and Deployment Wizard, see the documentation included with Visual Basic 6.0.

COM Add-ins and Security

You can specify security settings for Microsoft® Office XP applications in the Office XP Security dialog box, available by pointing to Macro on the Tools menu and then clicking Security. The Security Level tab includes a check box, Trust all installed add-ins and templates. If this box is selected, Office XP applications will load all COM add-ins, application-specific add-ins, and templates in trusted folders without checking to see whether they have valid digital signatures from trusted sources.

If this check box is not selected, the Office XP application checks to see whether the add-in or template has been signed digitally by a trusted source before loading it. If it has, the add-in will be loaded under any security level. If it has not been signed, if it has not been signed by a trusted source, or if the signature has been invalidated, the add-in will not load under high security. Under medium security, users will be warned that the add-in might not be safe. Under low security, the add-in will load and run without prompting the user.

To digitally sign a COM add-in DLL, you must obtain a digital certificate from a certificate authority, and you must run the Signcode.exe utility included with the Microsoft Internet Client Software Development Kit (SDK) on the COM add-in DLL. A digital certificate identifies the developer of a component as a trusted source. For more information about digitally signing a DLL, search the Microsoft Developer Network (MSDN®) Web site, at http://msdn.microsoft.com/ , for "digital signing."

You can use the COMAddIn object and the COMAddIns collection to control COM add-ins from Microsoft® Visual Basic® for Applications (VBA) code that is running within the host application. For example, you can load an add-in programmatically when a user clicks a button to access a particular feature; or, you can load an add-in from VBA when you open an application through Automation.

The Office XP object library supplies the COMAddIn object and the COMAddIns collection. The Application object for each Office XP application-Microsoft(Word, Microsoft® Excel, Microsoft® PowerPoint®, Microsoft® Access, Microsoft® FrontPage®, and Microsoft® Outlook®-has a COMAddIns property, which returns a reference to the COMAddIns collection. For any application, the COMAddIns collection contains only those COM add-ins that are registered for that application. The COMAddIns collection in Excel, for example, contains no information about COMAddIn objects in Word.

The Connect property of a COMAddIn object sets or returns the load status of the add-in. Setting this property to True loads the add-in, while setting it to False unloads it.

The ProgId property returns the name of the registry subkey that stores information about the COM add-in. The registry subkey takes its name from the COM add-in's programmatic identifier, which consists of the name of the Add-in project followed by the name of the add-in designer or class module that is actually supplying the add-in for a particular application. For example, when it is properly registered, the Image Gallery sample add-in for Excel has the following value for its ProgId property:

ImageGallery.dsrImageExcel

The name of the Add-in project is ImageGallery, and the name of the add-in designer for the Excel version of the add-in is dsrImageExcel.

You can use an add-in's ProgId property value to return a reference to the add-in from the COMAddIns collection, as shown in the following code fragment, which prints the current value of the Excel Image Gallery COM add-in's Connect property:

```
Debug.Print Excel.Application.COMAddIns("ImageGallery.dsrImageExcel").Connect
```

You can use the COMAddIn object and COMAddIns collection to get information about available COM add-ins from code running in an Office XP application. You can also use it to load and unload add-ins from code running in the add-in host application, or from code that is performing an Automation operation on the host application from another application.

If you are concerned about the performance of your application, you might want to load an add-in only at certain times. You can control this by loading and unloading it through VBA code.

The following code uses Automation to launch Word from another application, such as Excel, and load the Image Gallery add-in. To run this code from another application, remember to first set a reference to the Word object library.

```
Function LoadWordWithImageGallery() As Boolean
  ' Loads Word and connects Image Gallery add-in. If Image Gallery add-in is not available, procedure fails silently and_
  ‘ returns False.
  Dim wdApp       As Word.Application
  Dim cmAddIn     As Office.COMAddIn

  ' Create instance of Word and make visible.
  Set wdApp = New Word.Application
  wdApp.Visible = True
  ' Return reference to COM add-in, checking for error in case it doesn't exist.
  On Error Resume Next
  ' Set reference to COM add-in by using its ProgId property value.
  Set cmAddIn = wdApp.COMAddIns("ImageGallery.dsrImageWord")
  If Err.Number = 0 Then
    ' Connect add-in.
    cmAddIn.Connect = True
    ' Perform other operations here.
    LoadWordWithAddIn = True
  Else
    ' Return False if error occurred.
    LoadWordWithAddIn = False
  End If
   ' Enter break mode here to verify that add-in is loaded.
  Stop
  ' Quit Word.
  wdApp.Quit
  Set wdApp = Nothing
End Function
```

Building Application-Specific Add-ins

For some solutions, creating an application-specific add-in is easier and more convenient than building a COM add-in.

In This Section

Word Add-ins
Add functionality to a Microsoft® Word solution by creating a Word-specific add-in.

Excel Add-ins
Build a Microsoft® Excel add-in to add tools or commands to a user's Excel environment.

PowerPoint Add-ins
Build a Microsoft® PowerPoint® add-in to provide additional functionality to users while they are developing or running a PowerPoint slide presentation.

Access Add-ins
Build add-ins for Microsoft® Access to help users manage and analyze their databases.

Adding and Removing Command Bars for Word, Excel, and PowerPoint Add-ins
Include code to display or to create the command bar and control when the add-in loads and to hide or to remove the command bar and control when it unloads.

Controlling Word, Excel, and PowerPoint Add-ins from Code
Use Microsoft® Word, Microsoft® Excel, and Microsoft® PowerPoint® AddIn objects and the AddIns collections to control the behavior of application-specific add-ins from Microsoft® Visual Basic® for Applications (VBA).

Securing an Access, Excel, PowerPoint, or Word Add-in's VBA Project,
Protect your code and prevent users from changing it by setting a password for the add-in Microsoft® Visual Basic® for Applications (VBA) project.

Word Add-ins

You can add functionality to a Microsoft® Word solution by creating a Word-specific add-in (also sometimes referred to as a global template). Add-ins are good for adding generic functionality to the Word environment. For example, you might create a

Word add-in that contains common tools for working with Word documents. The user can use any of these tools with his or her documents by clicking the toolbars and menu commands that the add-in provides.

To see a list of currently available Word add-ins, click Templates and Add-Ins on the Tools menu. The currently loaded add-ins appear checked in the Global templates and add-ins list in the Templates and Add-Ins dialog box.

Although Word add-ins and Word templates both have the .dot file extension, they contribute functionality to a Word document in different ways. An add-in is a supplemental program that adds custom commands or custom features to an application. A template is a special kind of document that provides boilerplate text, custom styles, and macros for shaping a final document.

Creating a Word Add-in

You should create an add-in when:

- Your solution does not require boilerplate text or custom styles.
- You want to make some functionality available to any document the user creates, through toolbar buttons, menu commands, or macros.

To create a Word add-in

1. Create a Microsoft® Word document. Then, from the **File** menu, select **Save As** , and select **Word Document Template** in the **Save as Type** box.
2. From the **Tools** menu, select **Macro** , and then select **Visual Basic Editor** .
3. From the **File** menu, select **New Project,** and then select **Add-In Project.**
4. Specify the new add-in, and add code that creates a new toolbar with buttons that call your code when they are clicked.
5. From the **Debug** menu, select **Compile ProjectName.**
6. If you want, you can protect the project from viewing, as described in Securing an Access, Excel, PowerPoint, or Word Add-in's VBA Project.
7. Save the template as type Document Template with the .dot extension.

To change the default path for templates

1. In Word, from the **Tools** menu, select **Options.**
2. In the **Options** dialog box, select the **File Locations** tab.
3. From the **File types** list, select **User templates** , and then click **Modify.**

Note
If you want the add-in to load automatically when you start Word, save the add-in to the **Word Startup** folder. In addition, you can modify the default location for workgroup templates in the **Options** dialog box. Workgroup templates are templates that you share on a network with other users.

Loading a Word Add-in

You can load an add-in manually, automatically, or programmatically.

To load a Microsoft® Word add-in manually, select Templates and Add-Ins from the Tools menu, and then select the check box next to the template's name in the Global templates and add-ins list.

Note
If the add-in does not appear in the list, click Add to locate it. When an add-in is loaded, it is available to each new document that is created until you clear the check box in the Global templates and add-ins list in the Templates and Add-Ins dialog box.

To load a Word add-in automatically, save the template file in the Word Startup folder on your computer.

To load a Word add-in programmatically, you can try one of the two following methods.

- Call the Add method of the AddIns collection and pass in the add-in file name. By default, the Add method adds the add-in to the AddIns collection, if it is not there already, and loads the add-in. If the add-in is in the AddIns collection, it will appear in the Global templates and add-ins list in the Templates and Add-Ins dialog box. To add the add-in to the collection without loading it, pass in False for the optional Install argument.1.
- Set the Installed property of the corresponding AddIn object to True. When you try to set this property, an error will occur if the add-in has not been added to the collection already.

If an error occurs in a loaded add-in, you cannot debug the add-in code while it is loaded or view or modify its project. To view or change the code that is in the add-in project, open it directly in Word.

Running Code when a Word Add-in Is Loaded or Unloaded

To run code automatically when an add-in is loaded, create a Sub procedure named AutoExec in a standard module in the Add-In project. Any code within this procedure runs when the add-in is loaded. To run code when an add-in is unloaded, add a Sub procedure named AutoExit. If you close and reopen Microsoft® Word while an add-in is loaded, the AutoExec procedure runs when you reopen Word.

Note
The Document_Open event procedure does not run when a document is loaded as an add-in. It runs only when the document is opened directly in Word.

Excel Add-ins

You can build a Microsoft® Excel add-in to add tools or commands to a user's Excel environment. To load an Excel add-in, click Add-Ins on the Tools menu, and select the add-in from the list, or browse to find it if it does not appear in the list.

When the add-in has been loaded, any toolbars or menu items that it includes appear in Excel. An add-in remains loaded until the user unloads it or until Excel is closed, so tools in the add-in are available to all open workbooks. When the user closes Excel, the add-in is unloaded. It will be reloaded when Excel is opened only if the add-in is saved to the XLStart folder.

Several characteristics distinguish an Excel add-in from a typical workbook file:

- An add-in has the file extension .xla to indicate that it is an add-in.
- When you save a workbook as an Excel add-in, the workbook window is made invisible and cannot be viewed. You can use the invisible workbook and worksheets for storing calculations or data that your add-in requires while it is running.
- Users cannot use the SHIFT key to bypass events that are built into the add-in. This feature makes sure any event procedures you have written in the add-in will run at the proper time.
- Excel messages (alerts) are not displayed by code running in an add-in. In a standard workbook file, messages appear to verify that the user wants to perform an operation that might result in data loss, such as deleting a worksheet or closing an unsaved workbook file. In an add-in, you can perform such operations without the messages being displayed.

Creating an Excel Add-in

You create a Microsoft® Excel add-in by creating a workbook, adding code and custom toolbars and menu items to it, and saving it as an Excel add-in file.

To create an Excel add-in

1. Create a new workbook, add code to it, and create any custom toolbars or menu bars.
2. On the **File** menu, click **Properties** . In the **DocumentName Properties** dialog box, click the **Summary** tab, and then use the **Title** box to specify the name for your add-in, as you want it to appear in the **Add-Ins** dialog box.
3. Compile the Add-In project by clicking **Compile Project** on the **Debug** menu in the Visual Basic Editor.
4. If you want, you can protect the project from viewing as described in Securing an Access, Excel, PowerPoint, or Word Add-in's VBA Project.
5. Save the add-in workbook as type Excel add-in, which has the extension .xla.

Note
When you are creating an Excel add-in, pay close attention to the context in which your code is running. When you want to return a reference to the add-in workbook, use the **ThisWorkbook** property, or refer to the workbook by name. To refer to the workbook that is open in Excel currently, use the **ActiveWorkbook** property, or refer to the workbook by name.

When you have saved the add-in, you can reopen it in Excel to make changes to the project. The saved add-in no longer has a visible workbook associated with it, but when you open it, its project is available in the Microsoft® Visual Basic® Editor.

Saving the add-in workbook as an Excel add-in sets the IsAddIn property of the corresponding Workbook object to True.

You can debug an Excel add-in while it is loaded. When you load an add-in, its project appears in the Solution Explorer in the Visual Basic Editor. If the project is protected, you must enter the correct password to view its code.

Loading an Excel Add-in

You can load a Microsoft® Excel add-in in one of three ways:

- **Manually** Select the check box next to the name of the add-in in the Add-Ins dialog box on the Tools menu.
- **Automatically when Excel starts** Save the add-in to the ..\ \Excel\XLStart subfolder. You can change the location of the XLStart subfolder on the General tab of the Options dialog box (Tools menu).
- **Programmatically** Use the Add method of the AddIns collection to add the add-in to the list of available add-ins, and then set the Installed property of the corresponding AddIn object to True.

For example, the following procedure loads an add-in by first checking whether it is in the AddIns collection and adding it if it is not. Then, the procedure sets the add-in's Installed property to True. To call this procedure, pass in the path and file name of the add-in that you want to add:

```
Function Load_XL_AddIn(strFilePath As String) As Boolean
  ' Checks whether add-in is in collection, and then loads it. To call this procedure, pass in add-in's path and file name.
  Dim addXL          As Excel.AddIn
  Dim strAddInName   As String

  On Error Resume Next
  ' Call ParsePath function to return file name only.
  strAddInName = ParsePath(strFilePath, FILE_ONLY)
  ' Remove extension from file name to get add-in name.
  strAddInName = Left(strAddInName, Len(strAddInName) - 4)
  ' Attempt to return reference to add-in.
```

```
  Set addXL = Excel.AddIns(strAddInName)
  If Err <> 0 Then
    Err.Clear
    ' If add-in is not in collection, add it.
    Set addXL = Excel.AddIns.Add(strFilePath)
    If Err <> 0 Then
      ' If error occurs, exit procedure.
      Load_XL_AddIn = False
      GoTo Load_XL_AddIn_End
    End If
  End If
  ' Load add-in.
  If Not addXL.Installed Then addXL.Installed = True
  Load_XL_AddIn = True
Load_XL_AddIn_End:
  Exit Function
End Function
```

Running Code Automatically when an Excel Add-in Is Loaded or Unloaded

To run code automatically when a Microsoft(Excel add-in is loaded, you have two choices:

- **Create a Sub procedure named Auto_Open in a standard module in the Add-In project.** Any code within this procedure runs when the add-in is loaded. To run code when an add-in is unloaded, add a procedure named Auto_Close.

-or-

- **Add code to the add-in workbook's Open event procedure.** The code in this procedure also runs when an add-in is loaded, and it runs before the Auto_Open procedure runs.

Keep in mind that if you want an add-in to load automatically when Excel starts up, you must save it in the ...\Microsoft\Excel\XLStart subfolder. If the add-in is not saved in this folder, it is not loaded when Excel starts.

PowerPoint Add-ins

Microsoft® PowerPoint® add-ins are similar to Microsoft® Excel add-ins. You build a PowerPoint add-in to provide additional functionality to users while they are developing or running a PowerPoint slide presentation. In most cases, the user works with your add-in by clicking a toolbar button or menu item that you have included with the add-in.

Creating a PowerPoint Add-in

To create a Microsoft® PowerPoint®add-in, you create a new presentation and add code and custom toolbars. Then, you save your presentation as both a presentation file (.ppt) and a PowerPoint add-in (.ppa).

To create a PowerPoint add-in

1. Create a new presentation and add code to its Microsoft® Visual Basic® for Applications (VBA) project, and create any custom toolbars or menu bars.
2. When you have tested and debugged the code, compile the project by clicking **Compile VBAProject** on the **Debug** menu.
3. If you want, you can protect the project from viewing as described in Securing an Access, Excel, PowerPoint, or Word Add-in's VBA Project.
4. Save the project as a PowerPoint presentation, with the extension .ppt, and then save the project as a PowerPoint add-in, which has the extension .ppa. By default, PowerPoint add-ins are saved to the same folder as Excel add-ins-..\Microsoft\Addins subfolder. This folder is where PowerPoint looks for add-ins when you browse for a new add-in in the **Add-Ins** dialog box (**Tools** menu).

Note
When you save the project as a PowerPoint add-in, you can no longer view the VBA project, not even in break mode, nor can you view the slides associated with it.

Therefore, you also should save your PowerPoint add-in as a standard presentation, in case you must make changes to it and resave it as an add-in.

Loading a PowerPoint Add-in

You can load a Microsoft®PowerPoint® add-in in any of the following ways:

- **Manually** Click Add-Ins on the Tools menu. The Available Add-Ins list displays the available add-ins; you can add add-ins to the list by clicking Add New and locating the add-in file. Any add-in that is loaded currently has an "x" next to its name. To unload an add-in, select it, and click Unload. You can use an add-in only when it is loaded.
- **Automatically when PowerPoint starts** Set the AutoLoad property of the AddIn object to True in the Microsoft® Visual Basic® Editor The next time you start PowerPoint, the add-in is loaded, and the Loaded property is set to True.
- **Programmatically** Set the Loaded property of the corresponding AddIn object to True.

Running Code Automatically when a PowerPoint Add-in Is Loaded or Unloaded

To run code automatically when an add-in is loaded, create a Sub procedure named Auto_Open in a standard module in the Add-In project. Any code within this procedure runs when the add-in is loaded. To run code when an add-in is unloaded, add a procedure named Auto_Close.

If you close Microsoft® PowerPoint® while an add-in is loaded, the Auto_Open procedure will run when you reopen PowerPoint, because the add-in is reloaded on startup.

Access Add-ins

You can build add-ins for Microsoft® Access to help users manage and analyze their databases. Access includes several add-ins, which are written in Microsoft® Visual Basic® for Applications (VBA). For example, the Linked Table Manager is an add-in that handles the updating of linked tables when the database containing the source tables is moved or renamed. The wizards included with Access are also add-ins.

Access add-ins have the file extension .mda or .mde. A user can open an .mda file and look at the code, unless you have secured the modules by using either user-level security or project-level security. When you create an .mde file, however, all VBA source code is removed. The .mde contains only compiled VBA code, which cannot be viewed by the user. Creating an .mde file is therefore the best way to secure your code, if you are concerned about protecting your source code. For more information about .mda and .mde files, search the Microsoft Access Help index for "MDE files."

When you write code that will run in an Access add-in, use caution when referring to the current database. If you want to refer to the add-in database in which code is currently running, use the CodeProject or CodeData object to return a reference to this database. If you want to refer to the database that is currently open in Access, use the CurrentProject or CurrentData object.

Creating Menu Add-ins for Access

The simplest Microsoft® Access add-in is a menu add-in. A menu add-in calls a procedure in another database, perhaps a database that is serving as a code library. For example, a simple menu add-in might call a procedure that generates a report containing information about the various objects in the current database, such as the date they were created and their descriptions. Menu add-ins appear when you point to Add-Ins on the Tools menu.

To create a menu add-in

1. Add a subkey to the registry that specifies the name of the file containing the procedure and the name of the procedure itself. Menu add-ins are listed beneath the following subkey in the registry:
   \HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Office\Access\Addins\Menu add-ins
2. To specify the command that should appear on the **Add-Ins** submenu of the **Tools** menu, create a new subkey beneath the Add-In's subkey. For example, naming this subkey **&Analyze Database Objects** would result in a command named **Analyze Database Objects** on the **Add-Ins** submenu.
3. To hook up the menu command to the add-in, add two entries (in this case, **String** values) beneath the command's subkey, one named **Expression** and one named **Library** . Set the value of the **Library** entry to the path and file name of the database that contains the procedure that provides an entry point to the add-in. Set the value of the **Expression** entry to the name of the procedure itself. For example, if the procedure is named **AnalyzeDatabaseObjects** and it resides in a database named **CodeLib.mda** , you would set these entries as follows:

Expression: "=AnalyzeDatabaseObjects()"
Library: "C:\Windows\Application Data\Microsoft\AddIns\CodeLib.mda"

After you have added these keys, the new add-in command will appear on the Add-Ins submenu of the Tools menu the next time you open Access.

Note
If you must distribute your Access menu add-in to users, create an installable add-in, so the add-in is properly registered on users' machines.

Creating Installable Add-ins for Access

You can create add-ins that the user can load (install) or unload (uninstall) by using the Add-In Manager. The Add-In Manager can load the following types of add-ins:

- Menu add-ins, such as those described in Creating Menu Add-ins for Access.
- Object wizards, which help the user create a new table, query, form, data access page, or report. Microsoft® Access includes a number of built-in object wizards, which are available in the New Table, New Query, New Form, New Data Access Page, and New Report dialog boxes. An object wizard that you create also will appear in one of these dialog boxes.
- Control wizards, which help the user to add either an Access control or a Microsoft® ActiveX® control to a form, report, or data access page. A control wizard runs only if the Control Wizards tool in the toolbox is depressed. When this button is depressed, clicking a control in the toolbox and dropping it onto a form, report, or data access page launches the wizard that is associated with that control.

- Builders, which help the user to set a property for an object in the database-usually through a dialog box. When a builder is available for a particular property, the Build button (the small button with the ellipsis [...]) appears next to that property's name in the property sheet.

To load or unload one of these add-ins, the Add-In Manager relies on the presence of a table within the add-in, called the USysRegInfo table. The USysRegInfo table provides information that the Add-In Manager writes to the registry. Access uses this registry information to launch the add-in in response to an action taken by the user.

Note
The USysRegInfo table is a system table and usually is hidden. To view system tables, click Options on the Tools menu, click the View tab, and then select the System objects check box.

You must create the USysRegInfo table; it is not created for you automatically when you create a new .mda file. The USysRegInfo table must contain the four fields described in the following table.

Field	Field type	Description
Subkey	Text	The name of the subkey that contains the registry information for the add-in
Type	Number	The type of value to create beneath the subkey: subkey (0), String (1), or DWORD (4)
ValName	Text	The name of the registry entry to be created
Value	Text	The value to be stored in the registry entry defined by the ValName field

Each record in the USysRegInfo table describes a subkey or value that is to be added to the registry for a particular add-in. The table can contain information for multiple add-ins.

For each add-in, the USysRegInfo table must contain a minimum of three records: one to create the subkey for the add-in, one to add the Library entry, and one to add the Expression entry. Note that these are the same values required to create a menu add-in, as described in the previous section. You can add other records to store additional values in the registry. For example, you might add a record that creates a registry entry that indicates where a bitmap file required by the add-in is stored.

In the Subkey field, you can use the HKEY_CURRENT_ACCESS_PROFILE\AddInType\AddInName string to create the new registry entry. The Add-In Manager uses this string to determine the location on the user's machine of Access-specific information in the registry, so Access can create the entry for the add-in in the appropriate place. If the user started Access with the /profile command-line option, this string makes sure the registry entry is created beneath the specified Access user profile; otherwise, the entry is created under the \HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Office\2002\Access\AddInType subkey in the registry. For more information about starting Access from the command line with the /profile option, search the Microsoft Access Help index for "user profiles."

Note
A user profile that you use to start Access from the command line is not the same thing as a user profile that is defined for logging on to the operating system. An Access user profile applies only to Access, and only when you start Access from the command line. A user profile defined for the operating system applies to every application on the system and is used to maintain system data for individual users.

You can also use the HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Office\2002\Access\AddInType string to specify that the registry entries for the add-in always should be created under this registry subtree and that Access user profiles are to be ignored. Note that in this case you must include the full registry path to the add-in's subkey.

Sample USysRegInfo Table

Subkey	Type	ValName	Value
HKEY_CURRENT_ACCESS_PROFILE \Menu Add-ins\&Create Procedures Table	0		
HKEY_CURRENT_ACCESS_PROFILE \Menu Add-ins\&Create Procedures Table	1	Library	|ACCDIR\ProcTable.mda
HKEY_CURRENT_ACCESS_PROFILE \Menu Add-ins\&Create Procedures Table	1	Expression	=AddProcsToTable()

Adding and Removing Command Bars for Word, Excel, and PowerPoint Add-ins

If the user runs tools in your add-in by clicking a command bar control (toolbar button or menu item), you can include code to display or to create the command bar and control when the add-in loads and to hide or to remove the command bar and control when it unloads. Although it might seem to be more effort, creating and destroying the command bar from within your code gives you greater control over when the command bar is displayed than only storing the command bar in the add-in file.

To create the command bar when the add-in is loaded, add code to the procedure that runs when the add-in is loaded: AutoExec for Microsoft® Word, or Auto_Open for Microsoft® Excel and Microsoft® PowerPoint®.

Note
These code examples do not show error handling. For example, the procedures do not handle the case when another add-in might have a command bar with the same name.

First, check whether the command bar already exists. If it does not, create it and add a button that runs a Sub procedure, as shown in the following example:

```
Private Const CBR_INSERT As String = "Insert Info Wizard"
Private Const CTL_INSERT As String = "Insert Info"

Sub AutoExec()
  Dim cbrWiz      As CommandBar
  Dim ctlInsert   As CommandBarButton

  On Error Resume Next
  ' Determine whether command bar already exists.
  Set cbrWiz = CommandBars(CBR_INSERT)
  ' If command bar does not exist, create it.
  If cbrWiz Is Nothing Then
    Err.Clear
    Set cbrWiz = CommandBars.Add(CBR_INSERT)
    ' Make command bar visible.
    cbrWiz.Visible = True
    ' Add button control.
    Set ctlInsert = cbrWiz.Controls.Add
    With ctlInsert
      .Style = msoButtonCaption
      .Caption = CTL_INSERT
      .Tag = CTL_INSERT
      ' Specify procedure that will run when button is clicked.
      .OnAction = "ShowForm"
    End With
...Else
    ' Make sure the existing commandbar is visible
    cbrWiz.Visible = True
  End If
End Sub
```

To delete the command bar when the add-in is unloaded, add code to the procedure that runs when the add-in is unloaded: AutoExit for Word, or Auto_Close for Excel and PowerPoint. The following procedure deletes the command bar created in the previous example:

```
Sub AutoExit()
  On Error Resume Next
  ' Delete command bar, if it exists.
  CommandBars(CBR_INSERT).Delete
End Sub
```

Controlling Word, Excel, and PowerPoint Add-ins from Code

Microsoft® Word, Microsoft® Excel, and Microsoft® PowerPoint® all have an AddIns collection that contains AddIn objects that correspond to application-specific add-ins. You can use these AddIn objects and the AddIns collections to control the behavior of application-specific add-ins from Microsoft® Visual Basic® for Applications (VBA).

Note that the AddIns collection and the COMAddIns collection are two separate collections. Both are returned by a property of the Application object: the AddIns property for application-specific add-ins, and the COMAddIns property for COM add-ins. However, the Microsoft® Office XP object library provides the COMAddIns collection, while the AddIns collection is part of the host application's object model.

Although the AddIn objects and the AddIns collections for Word, Excel, and PowerPoint are similar, they each have different properties and methods. For example, each AddIn object has a read/write property that you can set to load or unload the add-in. In Word and Excel, this is the Installed property; in PowerPoint, it is the Loaded property.

The following code displays information about PowerPoint add-ins in a message box:

```
Sub DisplayPptAddins()
  ' This procedure displays information about add-ins currently  registered and/or loaded in PowerPoint. To determine which
  ' add-ins are registered, VBA looks for add-ins in the registry.
```

```
    Dim lngNumAddIns      As Long
    Dim addPpt            As AddIn
    ' Used to build the dialog box.
    Dim strPrompt         As String
    Dim strRegistered     As String
    Dim strLoaded         As String
    Dim strTitle          As String

    ' Get the total number of add-ins.
    lngNumAddIns = PowerPoint.AddIns.Count
    Select Case lngNumAddIns
      Case 0
        ' No add-ins registered.
        strTitle = "No add-ins"
        strPrompt = "You currently have no PowerPoint"  & " add-ins registered."
      Case 1
        ' One add-in registered.
        strTitle = "One add-in Registered"
        strPrompt = addPpt.FullName
     Case Is > 1
        ' Set up the title for the dialog box.
        strTitle = lngNumAddIns & " add-ins Registered"
        ' Determine which add-ins are loaded and/or registered.
        strLoaded = "Loaded: " & vbCrLf
        strRegistered = vbCrLf & "Registered: " & vbCrLf
        ' Loop through the AddIns collection.
        For Each addPpt In PowerPoint.AddIns
          ' Check Loaded property.
         If addPpt.Loaded = msoTrue Then
            strLoaded = strLoaded & addPpt.FullName  & vbCrLf
         Else
            strRegistered = strRegistered & addPpt.FullName & vbCrLf
         End If
        Next addPpt
        ' Combine the loaded add-ins list with registered
        ' add-ins list.
        strPrompt = strLoaded & strRegistered
    End Select
    ' Display the dialog box.
    MsgBox strPrompt, vbInformation, strTitle
End Sub
```

For more information about using the AddIn object and AddIns collection, search the VBA host application's (Word, Excel, or PowerPoint) Visual Basic Reference Help index for "AddIn Object" and "AddIns collection."

Securing an Access, Excel, PowerPoint, or Word Add-in's VBA Project

If you want to protect your code and prevent users from changing it, you can set a password for the add-in's Microsoft® Visual Basic® for Applications (VBA) project.

To set the project password

1. Click **VBAProject Properties** on the **Tools** menu in the **Visual Basic Editor.**
2. On the **Protection** tab, select the **Lock** project for viewing check box.
3. Enter a password, and confirm it.

Creating Templates

In some cases, your application might require you to give users a framework within which to complete common tasks. A template can provide such a framework. Within a template, you can include boilerplate text and graphics, custom styles, toolbars and menu items, macros, and Microsoft® Visual Basic® for Application (VBA) code.

In This Section

Word Templates
Create custom word-processing applications, and take advantage of the power of Microsoft® Word to create nicely formatted invoicing, reporting, and form letter applications easily.

Excel Templates

Use a Microsoft® Excel template when you want to distribute a custom spreadsheet application that has an Excel user interface component.

PowerPoint Templates
Use a Microsoft®PowerPoint® template when you must have a custom application for building presentations.

Access Templates
Create default templates for the forms and reports stored in a database, so when you create a new form or report, it is based automatically on the default template.

Word Templates

Microsoft®Word is ideal for creating custom word-processing applications. You can take advantage of the power of Word to create nicely formatted invoicing, reporting, form letter applications, and so on.

Every Word document has an associated Microsoft® Visual Basic® for Applications (VBA) project. However, code you write in one document is not available easily to other documents. If you are creating an application in Word, it makes sense to create a custom document template and distribute that template to your users. That way, a number of different documents can call the code in the template. The same holds true for custom styles, toolbars, and recorded macros.

To further illustrate the advantages of packaging code in a template, consider the New event for a Word Document object. This event occurs when you create a new document from a template. The Document_New event procedure itself must reside in the template project; there is no reason to use it in a regular Word document (.doc file), because you cannot create a new document from another document.

The Normal Template

The Normal template (Normal.dot) is loaded automatically when you start Microsoft® Word. By default, new documents are based on the Normal template. Even if you attach another template to a document, any styles, text, AutoText entries, command bars, recorded macros, or code included in the Normal template are available to any document open in Word. If you look at the Project Explorer in the Microsoft® Visual Basic® Editor, you will see that the Normal template always appears.

Although you can customize the Normal template, it is not always the best way to distribute an application to users, because replacing their own Normal template might inconvenience them. They will lose any custom settings or macros they might have created. Moreover, many users and system administrators restrict access to the Normal template, so you might not be able to replace or modify it anyway.

A better way to distribute applications is to create either a custom document template or an add-in (global template) that can be loaded in addition to the Normal template. Which one should you use? If you want to build an application that makes it possible for users to create new documents based on an existing document and that can include text and custom styles, use a custom document template. If you want to add toolbars, menu commands, or macros that are available to every document the user opens, create an add-in. After an add-in is loaded, it is available to every document the user opens until the add-in is unloaded.

Custom Document Templates

One way to build an application in Microsoft® Word is to create a custom template on which a user bases new documents. The template that is attached to a document is specified in the Document template box in the Templates and Add-ins dialog box (Tools menu). A document can have only one document template. Even when a document template is attached to a document, however, the Normal template remains loaded.

You should create a custom document template when:

- Your application requires that some boilerplate text or fields be included in the document when it is created.
- You want to make custom styles available to each document the user creates.
- Your application includes custom toolbars or menus the user can use while working with documents based on the template.
- You want to call Microsoft® Visual Basic® for Applications (VBA) procedures in the template from code running in a document that is based on the template.

Custom document templates are good for ensuring that all users have a consistent set of styles and tools for working on a particular project. For example, if your team is writing a book, you can create a document template the writers use as the basis for each section.

Creating a Custom Document Template

To create a custom document template, click New on the File menu, select General Templates from the templates menu, then click the General tab, click Blank Document, and then select Template under the Create New section.

Note
By default, custom command bars are saved in Normal.dot. To save a command bar with a custom document template, create the command bar by clicking Customize on the Tools menu, clicking the Toolbars tab, and then clicking New. In the New Toolbar dialog box, click the document template's name in the Make toolbar available to list.

Creating a New Document Based on a Word Template

To create a new document based on your custom template, click New on the File menu to open the New dialog box. Your template should appear on the General tab or on one of the other tabs if you saved it in a subfolder of the Templates folder. Click the template, and then click OK.

In addition, you can attach a custom template to an existing document. Doing so will not add any text that is in the template to your document, but any code, styles, and toolbars in the template will be available to your document. On the Tools menu, click Templates and Add-ins, and then click Attach to find and attach your document template.

If you look at the VBA project for a document that has a custom document template attached, you will see that three projects appear in the Project Explorer in the Visual Basic Editor: the document's project, the custom template's project, and the Normal template's project. You can write code in any of these projects. In addition, you can call a procedure in the Normal template or in the custom template from a procedure in the document's project.

Note
When you create a document based on a template, that template appears in the document's References folder in the Project Explorer. If you open the References dialog box by clicking References on the Tools menu, you will see that the template appears selected in the list of available references. Attaching a template to a Word document sets a reference to the template's VBA project, making the code that is in that template available to any procedure in the document.

Word Document Templates vs. Word Add-ins (Global Templates)

Microsoft® Word add-ins and document templates both have the same file extension, the .dot extension. In fact, you can use a template as an add-in or an add-in as a template.

The best way to use a Word template is as the basis for new documents. For example, you might create an invoicing template that employees could use to create customer invoices. When users create a new document based on the template, some of the information is available to them already-the name of your company, the date, and so on. All they must do is enter the customer name and the items purchased.

An add-in, on the other hand, provides custom tools that employees can use to work with all of their Word documents, similar to the custom features provided in the UsefulTools.dot add-in. When you load an add-in, it remains loaded for each document opened in Word until you explicitly unload it.

The following table summarizes the similarities and differences between Word templates and add-ins:

Custom document template	**Add-in**
A document template has the .dot file extension.	An add-in has the .dot file extension.
You can attach only one template to a document. (The Normal template always is loaded whether or not there is an attached template.)	You can load multiple add-ins at the same time.
A template is attached to a document at the time the document is created or after the document is created by clicking the Attach button in the Templates and Add-ins dialog box (Tools menu) and selecting the template.	An add-in is loaded by selecting the corresponding check box in the Global templates and add-ins list in the Templates and Add-ins dialog box.
A template can be used by any document, but it must be attached to each individual document.	When loaded, an add-in is available to all documents.
A template adds toolbar buttons, menu items, macros, styles, or boilerplate text to a specific document.	An add-in adds toolbar buttons, menu items, or macros to the Word environment. It does not display any boilerplate text or contain any custom styles.
The attached template can be accessed from VBA by using the AttachedTemplate property of a Document object. Templates are available in the Templates collection. The Templates collection contains the Normal template, the attached template (if any), and any loaded add-ins.	Add-ins in the Global templates and add-ins list, whether loaded or not, can be accessed from VBA through the Word AddIns collection. In addition, add-ins can be accessed through the Templates collection.
A reference to the template's VBA project is set automatically when you attach a template to a document. Therefore, you can call procedures in the template's project from the document's project.	No reference is set to an add-ins' VBA project when it is loaded. Therefore, although you can call procedures in the add-in project through toolbars, menu items, or macros, you cannot call directly a procedure in an add-ins' project from code running in a document unless you explicitly set a reference to the add-ins' project.

Excel Templates

Microsoft® Excel templates differ from Microsoft® Word templates in that when you create a new workbook based on a template, your workbook is really a copy of that template. In Word, creating a document based on a template loads two Microsoft® Visual Basic® for Applications (VBA) projects-one for the template and one for the document.

Use an Excel template when you want to distribute a custom spreadsheet application that has an Excel user interface component. For example, you might create a reporting template that is formatted in a standardized fashion, with embedded graphics, so any reports users create with the template have the same look.

To create a new Excel template, create a new workbook and add the elements you want to include in the template, such as code, custom dialog boxes, custom worksheet and chart layouts, toolbars, and recorded macros. Save the template file in the C:\Windows\Application Data\Microsoft\Templates folder with the .xlt extension; if user profiles are being used, save the template in the C:\Windows\Profiles\UserName\Application Data\Microsoft\Templates folder.

Excel includes sample templates that you can install to familiarize yourself with how templates work and to get ideas for creating your own templates.

PowerPoint Templates

As with a Microsoft® Excel template, when you create a new Microsoft® PowerPoint® presentation based on a template, the new presentation is a copy of the template. Only one Microsoft® Visual Basic® for Applications (VBA) project is loaded for the new presentation, but it includes all the components you have defined in the presentation template.

Use a PowerPoint template when you must have a custom application for building presentations. A presentation template makes it easy for your users to build attractive slide presentations and saves them time laying out the presentation or looking for the right graphics. You can include content in the template, such as information about departmental contacts, for example, or placeholders for quarterly sales information in a financial presentation. In addition, you can include instructions that guide the user in completing the presentation.

PowerPoint includes a number of custom templates you can use and modify. The templates that appear on the Design Templates tab of the General Templates menu contain only formatted backgrounds. The templates that appear on the Presentations tab also contain text and placeholders for information, navigation buttons, and instructions for completing the presentation.

To create a PowerPoint template, create a new presentation, add any text, graphics, buttons, toolbars, custom dialog boxes, and code, and save the presentation in the C:\Windows\Application Data\Microsoft\Templates folder; if user profiles are being used, save the template in the C:\Windows\Profiles\UserName\Application Data\Microsoft\Templates folder.

To create a new presentation based on your custom template, run PowerPoint, and click New on the File menu. Select your template in the New Presentation dialog box, and then click OK.

Access Templates

Templates in Microsoft® Access are different from templates for any other Microsoft® Office XP application. Instead of creating a template for a database (.mdb) file, you can create default templates for the forms and reports stored in a database. This means, when you create a new form or report, it is based on the default template automatically. You can create a template for a form or a report in one of two ways:

- Create the form or report that you want to be the template, and save it with the name Normal to replace the default template.

-or-

- Create the form or report that you want to be the template, and save it with whatever name you want. On the Tools menu, click Options, click the Forms/Reports tab, and then type the name of your template in the Form Template or Report Template box.

Note

Access saves the settings for the Form Template and Report Template options in your Access workgroup information file, not in your user database (the .mdb file). When you change an option setting, the change applies to any database you open or create. To see the name of the template that is used currently for new forms or reports, click Options on the Tools menu, and then click the Forms/Reports tab.

To use your templates in other databases, copy or export the templates to them. If your templates are not in a database, Access uses the Normal template for any new forms and reports you create. However, the names of your templates appear in the Form Template and Report Template options in every database in your database system, even if the templates are not in every database.

Creating Wizards

A wizard is a template or add-in that walks a user through a series of steps to create a new document, spreadsheet, presentation, database, Web application, or some object within any of those applications. Typically, when users launch a wizard, they are

presented with a series of information-gathering forms, and when they have entered all the necessary information in a form, the wizard creates the new component or completes a task.

The advantages of using a wizard to deliver an application are that it is easy to use and that you can include detailed instructions on each frame of the wizard. For example, Microsoft® Word includes a Letter wizard that gathers information from the user and then creates a new letter based on that information. The wizard saves the user from having to lay the letter out correctly, as well as from having to think about where the information is placed in the final document. The Word letter templates provide the same result as the Letter wizard, but the user has to figure out where each bit of information in the letter goes and navigate around the document to insert it.

In This Section

Common Characteristics of Wizards
Understand how the way you choose to create a wizard depends on the level of complexity of your wizard, which application or applications you want it to run, and how you want to distribute it to your users.

Word Wizards
Create an application-specific wizard for Microsoft® Word, or use wizards that Word includes optionally.

Excel Wizards
Understand that a Microsoft® Excel wizard is a template or add-in.

PowerPoint Wizards
Use the Auto Content wizard, which automatically generates a presentation with generic content based on information the user entered in the wizard.

Access Wizards
Create a table, query, form, or report wizard that can be integrated into the Microsoft® Access user interface.

Common Characteristics of Wizards

You can create a wizard by using any of the following:

- A Microsoft® Word, Microsoft® Excel, or Microsoft® PowerPointr® template
- A Word, Excel, PowerPoint, or Microsoft® Access application-specific add-in
- A COM add-in for Microsoft® Office XP applications or for the Microsoft® Visual Basic® Editor

What you choose depends on the level of complexity of your wizard, which application or applications in which you want it to run, and how you want to distribute it to your users. A template or application-specific add-in is the simplest application. A COM add-in might be more complex, because the add-in DLL and any dependent files must be properly registered on the user's computer.

Some other common characteristics of wizards include:

- A form or set of forms that gathers information from the user and that appears when the user launches the wizard
- Navigation buttons (such as the standard Next, Previous, Cancel, and Finish buttons) that make it possible for the user to move back and forth between pages
- The ability to launch the wizard either from a command bar control or by creating a new document based on the wizard
- An optional special file extension

As you can see, wizards do not significantly differ from add-ins or templates.

Tip
Rather than creating a new form for each page of your wizard, you can create a multi-page control on a form, with a unique control layout on each page. Then, when the user clicks the Next or Previous button, move the focus to the appropriate page. This way, you are not required to re-create the form background and buttons for each page of the wizard. Also, you do not have to manage the opening and closing of multiple forms.

Word Wizards

Microsoft® Word includes several wizards that are installed optionally; the Letter wizard, the Memo wizard, and the Resume wizard are a few examples. These files have the extension .wiz, but they are Word templates. You can open them in Word and view their VBA projects.

To create an application-specific wizard for Word, first create a Word template that contains any boilerplate text, plus the wizard forms and code. The wizard should include code that displays a form as soon as the user launches the wizard.

Next, determine how users will launch the wizard. If they will launch the wizard from a command bar control, you can add the control programmatically from code running in a Word add-in.

To design a wizard that is launched from a command bar control

1. Add the **AutoExec** procedure to a standard module in the wizard's project, and include the code to create the control in that procedure.

2. In the code that creates the control, set the control's **OnAction** property to the name of a procedure in the wizard project that displays the starting form for your wizard.
3. Add the **AutoExit** procedure, and include code to remove the control when the wizard is unloaded, so the user does not see the control unless the wizard is loaded.
4. Load your wizard as an add-in.

If the user will launch the wizard by creating a new document, you are not required to have a command bar control, nor the AutoExec nor AutoExit procedures.

To design a wizard that is launched by creating a new document

1. In the wizard's VBA project, open the **ThisDocument** module.
2. Create the Document_New event procedure by clicking **Document** in the **Object** box and **New** in the **Procedure** box.
3. Within this event procedure, call the procedure that displays the wizard's starting form.
4. Copy the wizard template to the C:\Windows\Application Data\Microsoft\Templates folder, or if user profiles are being used, to the C:\Windows\Profiles\UserName\Application Data\Microsoft\Templates folder, and change the file's extension to .wiz. Confirm this change when Windows prompts you to do so.

When users create a new document by clicking New on the File menu, they will see your wizard displayed in the New dialog box. Clicking the wizard and then clicking OK creates a new document and runs the Document_NewEvent procedure, which displays the wizard's starting form.

Excel Wizards

A Microsoft® Excel wizard is a template or add-in. No special file format indicates that an Excel file is a wizard. To create an Excel wizard, follow the guidelines discussed in Excel Templates and Excel Add-ins.

PowerPoint Wizards

Microsoft® PowerPoint® includes the Auto Content wizard, which automatically generates a presentation with generic content based on information that the user entered in the wizard. Unfortunately, you cannot view the Microsoft® Visual Basic®for Applications (VBA) project associated with the Auto Content wizard, because it is saved as a PowerPoint add-in.

The presentations created by the Auto Content wizard are based on the presentation templates included with PowerPoint. You could create a new presentation based on one of these templates and achieve the same result. Again, the advantage to using the wizard is that it enters some of the information into the presentation for you.

To create a custom PowerPoint wizard, follow the instructions for building a PowerPoint add-in described in PowerPoint Add-ins. Remember to save your presentation as a .ppt file in case you must re-create the add-in.

If you want the user to be able to create a new presentation based on your wizard, copy the wizard to the C:\Windows\Application Data\Microsoft\Templates folder, or if user profiles are being used, to the C:\Windows\Profiles\UserName\Application Data\Microsoft\Templates folder, and change its extension to .pwz. When users click New on the File menu in PowerPoint, they can click your wizard in the New Presentation dialog box, and then click OK to launch the wizard and create a new presentation.

Access Wizards

A Microsoft® Access wizard is an add-in that can be integrated into the Access user interface. You can create a table, query, form, or report wizard, which appears in the list of options in the New Table, New Query, New Form, or New Report dialog box. For example, you can design a wizard to help users build complex queries, such as update queries.

In addition, you can create control wizards, which are launched when users create new controls on a form or report. Users can disable control wizards by toggling the state of the Control Wizards tool in the toolbox.

You can add a USysRegInfo table to a wizard database and use the Add-in Manager to install wizards. The registry subkeys you must create to register a wizard, however, are different from those you create to register an add-in.