

Design Outlook Workgroup Solutions

With Microsoft Outlook, you can create a variety of workgroup solutions and [forms](#). You can create workgroup solutions by using custom views in a [public folder](#). You can also create simple forms with no programming involved or create advanced forms by using custom controls, properties, and VBScript. You can also use another Office program to create a form.

There are four basic approaches to creating Outlook solutions. You can use existing [items](#), such as tasks or appointment items, put the items in a public folder, and assign a view to the folder. This method gives you an instant workgroup solution without writing any code. Another method is to modify an existing item, such as a contact or mail message, by adding additional [pages](#) and [fields](#). With this method, you can extend the use of the item by adding the fields and pages you need without writing any code. To create advanced forms, you can use the **Control Toolbox** and VBScript to access properties, events, methods, and objects within Outlook. You can also use Office document forms, such as using Microsoft Excel to create an expense report.

What method do you want to use?

[Create an instant workgroup solution using public folders](#)

[Create custom forms by using VBScript](#)

Overview of a Typical Folder-based Solution

When used with Microsoft Exchange Server, Microsoft Outlook provides groupware capabilities that allow more than one person to work on the same data. Even without using these capabilities, however, you can still customize Outlook folders for your personal use or to create a solution that other people in your organization can use.

If you're planning to build a groupware solution, you should consider first working with Outlook to develop a single-user solution. Since a groupware solution is often as simple as placing a custom form in a Microsoft Exchange public folder, learning the basics of building a folder-based personal forms solutions will be fully applicable to developing solutions in public folders.

Whether it's in one of your personal folders or in a Microsoft Exchange public folder, putting together an Outlook solution typically involves the following steps:

1. Decide which type of folder and/or form to customize.
2. Open a new form and customize it to suit your needs.
3. Publish the form to the folder so that it's available only for use in that folder.
4. Set the custom form as the default form for the folder. This will ensure that when you create new items for use in this folder they'll be based on the custom form you created.
5. If you have any existing items in the folder, update these items so they will use the new form when they're opened.

Note If you plan to use a custom form in a folder, you should create the

custom form before creating any items based on the form. This will ensure that all the items in the folder use the same set of fields.

If you need to update the form after having created some items in the folder, follow these steps:

1. Open a new, blank item based on your custom form.
2. Update the form to suit your needs.
3. Republish the form to the folder using the same name.

Using views to create a solution

Although custom forms provide a powerful way to work with data in a folder, you can also customize Microsoft Outlook to a great extent simply by creating a view that suits your needs. In some cases, depending on how you want to work with Outlook, you can use a custom view instead of a custom form to enter and modify data.

Entering data in a view is usually done by selecting one of the Outlook table views, wherein each row contains one item in the folder and each column represents a field associated with an item. You can select the fields that Outlook displays, and you can also enter or change the contents of fields directly in the table view. This allows you to quickly modify items, even those that contain custom fields.

To allow field editing in a view

1. Click **View**, point to **Current View**, then click **Customize Current View**.
2. Click **Other Settings**.
3. Select **Allow in-cell editing**.

Editing items in a table view lets you keep track of custom data, view multiple items at the same time, enter data into an item without having to open the item, and avoid having to create a custom form.

Change the form used by existing items in a folder

In some cases you may need to change the form associated with items that are already in a folder. This is often necessary after importing items, or if you create a custom form after you have already created items based on a standard Microsoft Outlook form.

The Message Class field cannot be directly changed using the Outlook user interface, but you can use VBScript, Visual Basic, or Visual Basic for Applications to change the Message Class field.

The following Automation code can be used as a basis for developing your own solution. This code assumes that the name of the new form is MyForm. It will change all contacts in your default contacts folder so that they will use MyForm.

```
Sub ChangeMessageClass()  
Set olApp = New Outlook.Application  
Set olNS = olApp.GetNameSpace("MAPI")  
Set ContactsFolder = _  
    olNS.GetDefaultFolder(olFolderContacts)  
Set ContactItems = ContactsFolder.Items  
For Each Itm in ContactItems  
    If Itm.MessageClass <> "IPM.Contact.MyForm" Then  
        Itm.MessageClass = "IPM.Contact.MyForm"  
        Itm.Save  
    End If  
Next  
End Sub
```

Note If you want to use a folder other than a default folder, use the **Folders** collection object to refer to any folder that is available in your Folder List.

Using contact item selector fields

You can customize a contact form using specially bound controls that display dialog boxes that let users change fields in the item.

Most of these controls are available directly from the **Field Chooser**. The following lists these controls.

Address Selector	Address Selected
------------------	------------------

E-mail Selector	E-mail Selected
-----------------	-----------------

Phone x Selector	Phone x Selected
------------------	------------------

To use these fields on a contact form page, open the form in [design mode](#) and then use the **Field Chooser** to drag the field onto the page. This will automatically create the appropriate controls and labels on the page.

You can also bind a standard control button to one of three special contact dialog boxes:

Categories

Check Address

Check Name

To bind a contact-item dialog box to a button

1. In design mode, drag a command button from the **Control Toolbox** to a page on the form.
2. Right-click the command button, click **Properties**, and then click **Value**.

3. Click **Choose Field**, point to **All Contact fields**, and then click the name of the dialog box you want to bind the button to.

Note You can also use the same method to bind the **Categories** dialog box to a button on the form of any item except a mail message. However, the **Categories** dialog box is listed only under **All contact fields**, regardless of the type of form being designed.

Outlook fields and equivalent properties

Name of field in Outlook Field Chooser	Name of equivalent Outlook object model property
% Complete	PercentComplete
Account	Account
Actual Work	ActualWork
Address Selected	N/A
Address Selector	N/A
All Day Event	AllDayEvent
Anniversary	Anniversary
Assigned	DelegationState
Assistant's Name	AssistantName
Assistant's Phone	AssistantTelephoneNumber
Attachment	Attachments
Bcc	BCC
Billing Information	BillingInformation
Birthday	Birthday
Business Address	BusinessAddress
Business Address City	BusinessAddressCity
Business Address Country	BusinessAddressCountry
Business Address PO Box	BusinessAddressPostOfficeBox
Business Address Postal Code	BusinessAddressPostalCode
Business Address State	BusinessAddressState
Business Address Street	BusinessAddressStreet
Business Fax	BusinessFaxNumber
Business Home Page	BusinessHomePage
Business Phone	BusinessTelephoneNumber
Business Phone 2	Business2TelephoneNumber

Callback	CallbackTelephoneNumber
Car Phone	CarTelephoneNumber
Categories	Categories
Cc	CC
Changed By	N/A
Children	Children
City	HomeAddressCity
Color	Color
Company	Companies
Company	CompanyName
Company Main Phone	CompanyMainTelephoneNumber
Complete	Complete
Computer Network Name	ComputerNetworkName
Contact	FormDescription .ContactName
Contacts	Links
Content	Body
Conversation	ConversationTopic
Country	HomeAddressCountry
Created	CreationTime
Customer ID	CustomerID
Date Completed	DateCompleted
Defer until	DeferredDeliveryTime
Department	Department
Distribution List Name	DLName
Do Not AutoArchive	NoAging
Download State	N/A
Due By	FlagDueBy
Due Date	DueDate
Duration	Duration

E-mail	Email1Address
E-mail 2	Email2Address
E-mail 3	Email3Address
E-mail Selected	N/A
E-mail Selector	N/A
End	End
Entry Type	Type
Expires	ExpiryTime
File As	FileAs
First Name	FirstName
Flag Status	FlagStatus
Follow-up Flag	FlagRequest
From	SentOnBehalfOfName
FTP Site	FTPSite
Full Name	FullName
Gender	Gender
Government ID Number	GovernmentIDNumber
Have Replies Sent To	ReplyRecipientNames
Hobbies	Hobby
Home Address	HomeAddress
Home Address City	HomeAddressCity
Home Address Country	HomeAddressCountry
Home Address PO Box	HomeAddressPostOfficeBox
Home Address Postal Code	HomeAddressPostalCode
Home Address State	HomeAddressState
Home Address Street	HomeAddressStreet
Home Fax	HomeFaxNumber
Home Phone	HomeTelephoneNumber

Home Phone 2	Home2TelephoneNumber
Icon	FormDescription .Icon
Importance	Importance
In Folder	Parent
Initials	Initials
Internet Free Busy Address	InternetFreeBusyAddress
ISDN	ISDNNumber
Job Title	JobTitle
Journal	Journal
Junk E-Mail Type	N/A
Language	Language
Last Name	LastName
Last Saved Time	N/A
Location	Location
Mailing Address	MailingAddress
Mailing Address Indicator	N/A
Manager's Name	ManagerName
Meeting Status	MeetingStatus
Message	Body
Message Class	MessageClass
Message Flag	FlagStatus
Middle Name	MiddleName
Mileage	Mileage
Mobile Phone	MobileTelephoneNumber
Modified	LastModificationTime
Nickname	NickName
Notes	Body
Office Location	OfficeLocation

Optional Attendees	OptionalAttendees
Organizational ID Number	OrganizationalIDNumber
Organizer	Organizer
Other Address	OtherAddress
Other Address City	OtherAddressCity
Other Address Country	OtherAddressCountry
Other Address PO Box	OtherAddressPostOfficeBox
Other Address Postal Code	OtherAddressPostalCode
Other Address State	OtherAddressState
Other Address Street	OtherAddressStreet
Other Fax	OtherFaxNumber
Other Phone	OtherTelephoneNumber
Outlook Internal Version	OutlookInternalVersion
Outlook Version	OutlookVersion
Owner	Owner
Pager	PagerNumber
Personal Home Page	PersonalHomePage
Phone <i>n</i> Selected	N/A
Phone <i>n</i> Selector	N/A
PO Box	HomeAddressPostOfficeBox
Primary Phone	PrimaryTelephoneNumber
Priority	Importance
Private	Sensitivity
Profession	Profession
Radio Phone	RadioTelephoneNumber
Read	UnRead
Received	ReceivedTime
Recurrence	RecurrencePattern

	.RecurrenceType
Recurrence Pattern	N/A
Recurrence Range End	RecurrencePattern .PatternEndDate
Recurrence Range Start	RecurrencePattern .PatternStartDate
Recurring	IsRecurring
Referred By	ReferredBy
Remind Beforehand	ReminderMinutesBeforeStart
Reminder	ReminderSet
Reminder Override Default	ReminderOverrideDefault
Reminder Sound	ReminderPlaySound
Reminder Sound File	ReminderSoundFile
Reminder Time	ReminderTime
Reminder Topic	N/A
Remote Status	RemoteStatus
RequestStatus	N/A
Requested By	N/A
Required Attendess	RequiredAttendees
Resources	Resources
Response Requested	ResponseRequested
Retrieval Time	N/A
Role	Role
Schedule+ Priority	SchedulePlusPriority
Send Plain Text Only	N/A
Sensitivity	Sensitivity
Sent	SentOn
Show Time As	BusyStatus
Size	Size

Spouse	Spouse
Start	Start
Start Date	StartDate
State	HomeAddressState
Status	Status
Street Address	HomeAddressStreet
Subject	Subject
Suffix	Suffix
Team Task	TeamTask
Telex	TelexNumber
Title	Title
To	To
Total Work	TotalWork
Tracking Status	TrackingStatus
TTY/TTDD Phone	TTYTDDTelephoneNumber
User Field 1	User1
User Field 2	User2
User Field 3	User3
User Field 4	User4
Web Page	WebPage
ZIP/Postal Code	HomeAddressPostalCode

Using Visual Basic with Outlook

You can use Visual Basic to customize and extend Microsoft Outlook. Outlook allows you to control Outlook by using Visual Basic, Visual Basic for Applications and VBScript. Which you use depends on what you want your program to do.

Visual Basic is a full-featured programming language you can use to create stand-alone applications or dynamic-link libraries (DLLs) that extend other applications. Visual Basic for Applications is a subset of Visual Basic that is run within an application to extend its capabilities. VBScript is a simplified version of Visual Basic for Applications and is run within an Outlook [item](#). In all cases, these programming languages control Outlook through its object model.

Learn about the [Outlook object model](#).

If you want to create a separate application that accesses data stored by Outlook and uses Outlook to send and receive messages, use Visual Basic to create the application (you can also use other programming languages, such as C++, to control Outlook through its object model). You can also use Visual Basic to create a DLL that can extend Outlook as a COM add-in.

You use Visual Basic for Applications in one of two ways: You can use Visual Basic for Applications in other applications (such as Microsoft Excel or Microsoft Word) to automate Outlook, or you can use Visual Basic for Applications within Outlook to control Outlook. If you expect your users to be using another application most of the time, and you want to give them the ability to send a message using Outlook or to access information stored by Outlook, write Visual Basic for Applications programs in that application that control Outlook through the Outlook object model. If, on the other hand, you want to write Visual Basic code that customizes how Outlook works (like a macro), use Visual Basic for Applications within Outlook.

You can extend the functionality of Outlook forms by using VBScript. VBScript programs are stored within a [form](#). Because the program code is contained within the form, it can be sent with an item to another user. An important consideration in choosing which kind of the Visual Basic

programming language you will use is the type of events you want your program to respond to. Because VBScript code is associated with a particular item, code that responds to events in specific items (such as when a particular item is opened or a value in a field is changed) is easiest to write using VBScript. If, on the other hand, you want your program to respond to events that occur in the application, in Windows Explorer, in folders, or in all items, then you should write your program using Visual Basic or Visual Basic for Applications.

Code written for Visual Basic or Visual Basic for Applications often does not work in VBScript without modification. For example, you must replace all built-in constants written in Visual Basic for Applications with the literal numeric values of those constants in VBScript. And VBScript uses only the **Variant** data type.

Learn about [constants and variables in VBScript](#).

In Outlook Visual Basic for Applications and VBScript, when you reference the **Application** object to use **CreateObject** or **GetObject**, you simply use **Application**. For example, the following code displays the Tasks folder:

```
Set olMAPI = Application.GetNameSpace("MAPI")  
olMAPI.GetDefaultFolder(13).Display
```

In Visual Basic or Visual Basic for Applications in other applications, you must explicitly create the **Application** object:

```
Set myOlApp = CreateObject("Outlook.Application")  
Set olMAPI = myOlApp.GetNameSpace("MAPI")  
olMAPI.GetDefaultFolder(olFolderTasks).Display
```

About using VBScript in Outlook

Microsoft VBScript is a powerful scripting language based on Microsoft Visual Basic that enables you to control objects, folders, forms, items, and controls within a form. For example, you can change properties and values of controls on a page, modify the default Microsoft Outlook item events, and even create automated procedures, such as mailing a notice to all the contacts in a Contacts folder.

You add VBScript code to an Outlook form to respond to [Click](#) events that are fired by controls on the form, or to respond to events fired by the items that have the same message class as the form. VBScript makes it especially easy to respond to item events because the VBScript code executes in the context of the item, so you don't have to set an object variable to point to the item. In addition, VBScript code is compact and can be contained within a form sent to other users.

With VBScript, you have full access to the Microsoft Outlook object model, except for two areas: VBScript code cannot respond to events other than item and form events, and you cannot use named constants defined in the Outlook object type library.

You can also use Visual Basic for Applications in Outlook to respond to Outlook events and to create macros that automate procedures. Unlike VBScript code, however, Visual Basic for Applications code cannot be contained in a form and so cannot accompany an item that is sent to other users.

For more information about using VBScript, see [Create custom forms by using VBScript](#).

How can I prevent the VBScript code from running?

To prevent any VBScript code from running, hold down the SHIFT key. For example, hold down SHIFT while you open an [item](#) to prevent the VBScript code for the Open event from running.

Variants supported in VBScript

Microsoft VBScript in Outlook uses only the **Variant** data type.

A **Variant** is a special kind of data type that can contain different kinds of information, depending on how it's used. Because **Variant** is the only data type in VBScript, it's also the data type returned by all functions in VBScript.

At its simplest, a **Variant** can contain either numeric or string information. A **Variant** behaves as a number when you use it in a numeric context and as a string when you use it in a string context. That is, if you're working with data that looks like numbers, VBScript assumes that it is numbers and does the thing that is most appropriate for numbers. Similarly, if you're working with data that can only be string data, VBScript treats it as string data. Of course, you can always make numbers behave as strings by enclosing them in quotation marks (" ").

Using the Script Editor

The Microsoft Script Editor allows you to add VBScript procedures that respond to events generated by [items](#) or form controls.

To open the Script Editor

1. Open an item of the type to which you want to add code.
2. On the **Tools** menu, point to **Forms** and then click **Design This Form**.
3. On the **Form** menu, click **View Code**.

The Script Editor makes it easy to insert the template for an item event handler.

To insert a blank item event handler

1. On the **Script** menu, click **Event Handler**.
2. Select the event you want to respond to, and then click **OK**.

The Script Editor can move the insertion point to a specific line of code. This makes it easy to debug the script when Microsoft Outlook reports an error at a specific location.

To move to a specific line

1. On the **Edit** menu, click **Go To**.
2. In the **Line Number** box, type in the number of the line of code to which you want to go.

Learn about [the Script Debugger](#).

Learn about [the Outlook object browser](#).

About the Outlook object browser

The Microsoft Outlook object browser displays the [classes](#), properties, methods, and events available from the Outlook object library. The object browser lets you view and insert these objects into the Script Editor and obtain information about the syntax for using the object.

Learn about [viewing and using the object browser](#).

About the Outlook script debugger

The Microsoft Script Debugger provides you with a comprehensive debugging environment for testing and correcting errors in the VBScript code that you created for your Microsoft Outlook [forms](#). The Script Debugger is a shared component that can also be used to track down errors in any Microsoft ActiveX-enabled scripting language and to debug Java applets, beans, and ActiveX components.

The Microsoft Script Debugger lets you debug both [client scripts](#) and [server scripts](#).

The Microsoft Script Debugger works the way many debuggers do, by allowing you to:

View the source code of the script that you are debugging.

Control the pace of the script execution.

View and change variable and property values.

View and control script flow.

The script debugger is only available in Outlook at [run time](#).

Learn about [using the Script Debugger in Outlook](#).

Set global variables for a form

A global [variable](#) is available to any procedure in a form while the script is running. To set a global variable, assign the value to the variables before any procedures.

Referencing controls on an Outlook form

If you need to refer to a control on an Outlook form within a procedure, you must also reference the inspector, page, and controls collection that contains the control, even if you are referencing the control with its own event procedure. The following example shows how to change the caption of a command button when it's clicked. To test this example, in [design mode](#) create a command button with the default name CommandButton1 on the page P.2.

```
Sub CommandButton1_Click
    Set myButton = Item.GetInspector.ModifiedFormPages("P.2")_
        .Controls("CommandButton1")
    myButton.Caption = "New Caption"
End Sub
```

Form Events

Form events occur when something happens to an item displayed in a form, such as when it's saved or opened or when a user-defined action is started.

Most often, form events are handled by VBScript code within the form itself.

Some events can be cancelled. That is, your event handler can prevent Microsoft Outlook from performing the default action associated with the event. For example, you can write an event handler for the **Forward** event to prevent an item from being sent to recipients who are not on a list of approved recipients. Learn about [canceling an event](#).

The following table lists the form events supported by Outlook.

Event	Cancelable?	Description
AttachmentAdd	No	Occurs when an attachment has been added to the item
AttachmentRead	No	Occurs when an attachment has been opened for reading
BeforeAttachmentSave	Yes	Occurs before an attachment is saved
BeforeCheckNames	Yes	Occurs before Outlook starts resolving names in the recipients collection of the item
Close	Yes	Occurs before Outlook closes the inspector displaying the item
CustomAction	Yes	Occurs before Outlook executes a custom action of an item
CustomPropertyChange	No	Occurs when a custom item property has changed
Forward	Yes	Occurs before Outlook executes the Forward action of an item

Open	Yes	Occurs before Outlook opens an inspector to display the item
PropertyChange	No	Occurs when an item property has changed
Read	No	Occurs when an item is opened for editing by a user
Reply	Yes	Occurs before Outlook executes the Reply action of an item
ReplyAll	Yes	Occurs before Outlook executes the Reply to All action of an item
Send	Yes	Occurs before Outlook sends the item
Write	Yes	Occurs before Outlook saves the item in a folder

Control Events

Outlook form controls support only one event, the [Click](#) event.

A control bound to a field does not fire the **Click** event. You must handle the appropriate [field event](#) to detect a user's interaction with a control bound to a field.

The following controls fire the **Click** event whenever a user clicks anywhere in the control.

[CheckBox](#)

[CommandButton](#)

[Frame](#)

[Image](#)

[Label](#)

[OptionButton](#)

[ToggleButton](#)

The following controls fire the **Click** event when the user selects an item in the list.

[ComboBox](#)

[ListBox](#)

The following controls do not support the **Click** event.

[MultiPage](#)

[ScrollBar](#)

[SpinButton](#)

[TabStrip](#)

[TextBox](#)

While the **MultiPage** control itself does not support the **Click** event, an individual [Page](#) on a **MultiPage** control will fire the **Click** event if the user clicks inside the client area of the page, but not if the user clicks the tab

associated with the page.

To detect a change in a **TextBox** control, bind the control to a field and then handle the appropriate [field event](#).

Field Events

Microsoft Outlook provides two events to notify your program that a field (property) in an item has changed. The **PropertyChange** event is fired whenever a standard Outlook field in an item has changed. Outlook fires the **CustomPropertyChange** event whenever a user-defined field changes.

A control that is bound to a field does not fire the [Click](#) event, whether the control was selected from the **Control Toolbox** and subsequently bound to a field, or was selected from the **Field Chooser**. Consequently, you must use the **PropertyChange** or **CustomPropertyChange** event to detect user interaction with a bound control.

Canceling an event

Microsoft Outlook calls event handlers in your program to allow your program to respond to such events as actions that the user takes or changes in the message store. Each event is accompanied by a default action that Outlook performs as a result of the event. For example, when the **Open** event occurs for an item, by default Outlook displays the item in an inspector window.

Some events only notify your program that a particular event has occurred. For these events, your event handler simply responds to the event. With other events, Outlook allows your event handler to cancel the event, that is, to instruct Outlook not to perform the default action associated with the event. In the case of the **Open** event, for example, your program can prevent Outlook from displaying the item in an inspector. If an event can be cancelled, the reference topic describing the event indicates how to cancel the event.

If an event can be cancelled, an event handler written in Microsoft Visual Basic or Microsoft Visual Basic for Applications receives a parameter that it sets before returning to indicate whether the event should be cancelled. For example, an event handler for the **Open** event written in Visual Basic for Applications might look like this. This example assumes that the value of `OpenOK` is set elsewhere.

```
Sub myItem_Open(byRef Cancel as Boolean)
    If OpenOK Then
        Cancel = False ' Outlook performs default action
    Else
        Cancel = True ' Outlook does not perform default action
    EndIf
```


End Sub

Because of limitations in VBScript, however, this syntax cannot be used. An event handler for the **Open** event in the script of an item must be written as a function. To cancel the event, the value of the function is set to **False** before returning, as in the following example.

```
Function Item_Open()  
    If OpenOK Then  
        Item_Open = True ' Outlook performs default action  
    Else  
        Item_Open = False ' Outlook does not perform default action  
    End If  
End Function
```


About the order of events

The following events occur in the order specified when a user completes an action.

Events	When
Open	A form is opened to compose an item .
Send, Write, Close	An item is sent.
Write, Close	An item is posted.
Write	An item is saved.
Close	An item is closed.
Read, Open	An item is opened in a folder.
Reply	A user replies to an item's sender.
ReplyAll	A user replies to an item's sender and all recipients.
Forward	The newly-created item is passed to the procedure after the user selects the Forward action for an item.
PropertyChange	One of the item's standard properties is changed.
CustomPropertyChange	One of the item's custom properties is changed.
CustomAction	A user-defined action is initiated.

The [Click](#) event occurs only when you have defined it for a control in the Script Editor.

Constants and variables in VBScript

In VBScript, constants must be referenced by their numeric values. The constant string does not work and returns a value of 0, which gives unpredictable results.

There are two types of variables. Procedure-level variables that are used only within a procedure and script-level variables that are available to all the procedures within your script. Declare script-level variables at the top of your script. Declare procedure-level variables inside procedures. You can use procedure-level variables with the same name in different procedures because each variable is recognized only by the procedure in which it's declared. When the procedure exits, the variable ends. Variables that refer to Outlook objects can be either procedure-level or script-level variables. However, the value of the variable must be set within a procedure. Do not attempt to access Outlook objects outside of a procedure.

Rules about variables:

Must begin with an alphanumeric character.

Cannot contain an embedded period.

Cannot exceed 255 characters.

Cannot use more than 127 procedure-level variables (arrays count as a single variable).

Cannot use more than 127 script-level variables.

Referencing fields

When you need to access the fields in an item, the method you use depends on whether the field is a standard, built-in Microsoft Outlook field, or a custom field.

In either case, you do not access the field directly. Instead, you refer to the field as a property of the item you're working with.

For example, to retrieve the text from the Subject field of a mail message, you use the **Subject** property of the item, as shown in the following VBScript example.

```
mySubject = Item.Subject
```

If the field is a custom (user-defined) field, you access it using the **UserProperties** property of the item, as shown in the following VBScript example. This example assumes that the item already contains a custom field named ReferredBy.

```
MyReferral = Item.UserProperties("ReferredBy")
```

Reference a folder

To reference a folder by the name of the folder, use the following code.

```
Application.GetNamespace("MAPI").Folders("Personal Folders").Fol
```

To reference a folder by a number, use the following code. In this example, the first folder in the folder collection Personal Folders is referenced.

```
Application.GetNamespace("MAPI").Folders("Personal Folders").Fol
```

To reference any of the default Outlook folders, use the **GetDefaultFolder** method. Use the value from the table below to specify the folder you want to create.

```
Application.GetNamespace("MAPI").GetDefaultFolder(6)
```

Default folder	Has the value
Deleted Items	3
Outbox	4
Sent Items	5
Inbox	6
Calendar	9
Contacts	10
Journal	11
Notes	12
Tasks	13

Creating a new item

To create a new item, use the **CreateItem** method of the **Application** object. This method returns an object that you can then use to work with the item.

The following Microsoft Visual Basic for Applications example shows how to create a mail message, add text to its subject and body, and display it. To use this sample, create a command button named Command1 on a form.

```
Private Sub Command1_Click()  
    Dim myOLApp As New Outlook.Application  
    Dim myOLItem As Outlook.MailItem  
    Set myOLItem = myOLApp.CreateItem(olMailItem)  
    With myOLItem  
        .Subject = "Sample item"  
        .Body = "This is a sample message."  
    End With  
    myOLItem.Display  
End Sub
```

The following example shows how to perform the same task using VBScript in a form.

```
Sub CommandButton1_Click()  
    Set myOLItem = Application.CreateItem(0)  
    myOLItem.Subject = "Sample item"  
    myOLItem.Body = "This is a sample message."
```



```
myOLItem.Display  
End Sub
```

Referencing existing items in a folder

There are a number of ways you can reference existing items in a folder using Microsoft Visual Basic. This topic provides information about:

Using a **For ... Next** or **For Each ... Next** loop

Using the **Items** collection

Using the **Find** method

Using the **Restrict** method

Using a For...Next or For Each...Next Loop

Typically these statements are used to loop through all of the items in a folder. The **Items** collection contains all the items in a particular folder, and you can specify which item to reference by using an index with the **Items** collection. This is typically used with the For I = 1 to n programming construct.

You can use For Each...Next to loop through the items in the collection without specifying an index. Both approaches achieve the same result.

The following examples use For...Next to loop through all the contacts in the Contacts folder and display the Full Name field in a dialog box.

```
' Visual Basic/Visual Basic for Applications code example.
```

```
Set ol = New Outlook.Application
```

```
Set olns = ol.GetNameSpace("MAPI")
```

```
' Set MyFolder to the default contacts folder.
```

```
Set MyFolder = olns.GetDefaultFolder(olFolderContacts)
```

```
' Get the number of items in the folder.
```

```

NumItems = MyFolder.Items.Count
' Set MyItem to the collection of items in the folder.
Set myItems = myFolder.Items.Restrict("[MessageClass] = 'IPM.Conta
' Loop through all of the items in the folder.
For I = 1 to NumItems
    MsgBox MyItems(I).FullName
Next

```

```

' VBScript code example.
Set olns = Item.Application.GetNameSpace("MAPI")
' Set MyFolder to the default contacts folder.
Set MyFolder = olns.GetDefaultFolder(10)
' Get the number of items in the folder.
NumItems = MyFolder.Items.Count
' Set MyItem to the collection of items in the folder.
Set myItems = myFolder.Items.Restrict("[MessageClass] = 'IPM.Conta
' Loop through all of the items in the folder.
For I = 1 to NumItems
    MsgBox MyItems(I).FullName
Next

```

The following examples use For Each...Next to achieve the same result as the preceding examples:

```

' Visual Basic/Visual Basic for Applications code example.
Set ol = New Outlook.Application
Set olns = ol.GetNameSpace("MAPI")
' Set MyFolder to the default contacts folder.
Set MyFolder = olns.GetDefaultFolder(olFolderContacts)
' Set MyItems to the collection of items in the folder.
Set myItems = myFolder.Items.Restrict("[MessageClass] = 'IPM.Conta
For Each SpecificItem in MyItems
    MsgBox SpecificItem.FullName

```

Next

```
' VBScript code example.  
Set olns = Item.Application.GetNamespace("MAPI")  
' Set MyFolder to the default contacts folder.  
Set MyFolder = olns.GetDefaultFolder(10)  
' Set MyItem to the collection of items in the folder.  
Set myItems = myFolder.Items.Restrict("[MessageClass] = 'IPM.Contact'  
For Each SpecificItem in MyItems  
    MsgBox SpecificItem.FullName  
Next
```

Using the Items Collection

You can also use the **Items** collection and specify a text string that matches the Subject field of an item. The following examples display an item in the Inbox whose subject contains "Please help on Friday!"

```
' Visual Basic/Visual Basic for Applications code example.  
Set ol = New Outlook.Application  
Set olns = ol.GetNamespace("MAPI")  
' Set MyFolder to the default Inbox.  
Set MyFolder = olns.GetDefaultFolder(olFolderInbox)  
Set MyItem = MyFolder.Items("Please help on Friday!")  
MyItem.Display
```

```
' VBScript code example.  
Set olns = Item.Application.GetNamespace("MAPI")  
' Set MyFolder to the default Inbox.  
Set MyFolder = olns.GetDefaultFolder(6)  
Set MyItem = MyFolder.Items("Please help on Friday!")  
MyItem.Display
```

Using the Find Method

Use the **Find** method to search for an item in a folder based on the value of one of its fields. If the search is successful, you can then use the **FindNext** method to check for additional items that meet the same search criteria.

The following examples search to see if you have any high priority tasks.

' Visual Basic/Visual Basic for Applications code example.

```
Set ol = New Outlook.Application
```

```
Set olns = ol.GetNamespace("MAPI")
```

```
Set myFolder = olns.GetDefaultFolder(olFolderTasks)
```

```
Set MyTasks = myFolder.Items
```

' Importance corresponds to Priority on the task form.

```
Set MyTask = MyTasks.Find("[Importance] = ""High""")
```

```
If MyTask Is Nothing Then ' the Find failed
```

```
    MsgBox "Nothing important. Go party!"
```

```
Else
```

```
    MsgBox "You have something important to do!"
```

```
End If
```

' VBScript code example.

```
Set olns = Item.Application.GetNamespace("MAPI")
```

```
Set myFolder = olns.GetDefaultFolder(13)
```

```
Set MyTasks = myFolder.Items
```

' Importance corresponds to Priority on the task form.

```
Set MyTask = MyTasks.Find("[Importance] = ""High""")
```

```
If MyTask Is Nothing Then ' the Find failed
```

```
    MsgBox "Nothing important. Go party!"
```

```
Else
```

```
    MsgBox "You have something important to do!"
```

```
End If
```

Using the Restrict Method

The **Restrict** method is similar to the **Find** method, but instead of

returning a single item, it returns a collection of items that meet the search criteria. For example, you could use this method to find all contacts that work at the same company.

The following examples display all of the contacts that work atProseWare Corporation:

' Automation code example.

```
Set ol = New Outlook.Application
```

```
Set olns = ol.GetNameSpace("MAPI")
```

```
Set MyFolder = olns.GetDefaultFolder(olFolderContacts)
```

```
Set myItems = myFolder.Items.Restrict("[MessageClass] = 'IPM.Conta
```

```
MyClause = "[CompanyName] = ""ProseWare"""
```

```
Set MyPWItems = MyItems.Restrict(MyClause)
```

```
For Each MyItem in MyPWItems
```

```
    MyItem.Display
```

```
Next
```

' VBScript code example.

```
Set olns = Item.Application.GetNameSpace("MAPI")
```

```
Set MyFolder = olns.GetDefaultFolder(10)
```

```
Set myItems = myFolder.Items.Restrict("[MessageClass] = 'IPM.Conta
```

```
MyClause = "[CompanyName] = ""ProseWare"""
```

```
Set MyPWItems = MyItems.Restrict(MyClause)
```

```
For Each MyItem in MyPWItems
```

```
    MyItem.Display
```

```
Next
```

Filtering items in a collection

You can use the Microsoft Outlook object model to return information about all items in a folder. Often, however, the desired objective is to search for a specific item or to retrieve a subset of the items in the folder. Consider the following examples:

You are developing a Microsoft Access database. When the user enters a new contact record, you want to give the user the ability to click a button to check whether a contact with the same name already exists in Outlook. If a match is found, you can retrieve all the fields for the contact and automatically fill in the Access database record. In this situation, if the user filled in the first and last name fields on the Access form, you can use the **Find** method in the Outlook object model to search for a match against the Outlook Full Name field. If you want to make sure there are no additional contacts in Outlook with the same name, you can then use the **FindNext** method to conduct the same search again.

You are writing a Microsoft Visual Basic program to automatically schedule appointments in users' calendars. In order to do this, you need to retrieve a user's appointments for a given day. In this case, you would use the **Restrict** method to retrieve all appointments that fall on a particular day.

While the **Find** and **Restrict** methods perform different functions, the syntax for both is similar. Following are some tips for using these methods:

The filters used with **Find** and **Restrict** are not case-sensitive.

It is not possible to use these methods if you need to search for a string that's contained within an Outlook field, commonly called a "Contains"

operation. If you need to perform a Contains operation, you can iterate through all of the items in the folder and use the Visual Basic **InStr** function to see if your search string is contained within an Outlook field.

To create a filter that performs a "begins with" operation, use the >, <, >=, and <= operators. For example, to search for all contacts whose last names begin with "Mc", use this filter: "[LastName] >= "Mc" and [LastName] < "Md"

You can use the contents of a variable as part of the filter.

Working with command bars

In Microsoft Outlook 2000, toolbars, menu bars, and shortcut menus are all controlled programmatically as one type of object: command bars. All the following items are represented in Microsoft Visual Basic by **CommandBar** objects:

Menu bars, toolbars, and shortcut menus

Menus on menu bars and toolbars

Submenus on menus, submenus, and shortcut menus

You can modify any built-in menu bar or toolbar, and you can create and modify custom toolbars, menu bars, and shortcut menus to deliver with your Visual Basic application. You can present the features of your application as buttons on toolbars or as groups of command names on menus. Because toolbars and menus are both command bars, you use the same kind of controls on both of them. Menu bars and toolbars can

both contain menus.

In Visual Basic, buttons and menu items are represented by **CommandBarButton** objects. The pop-up controls that display menus and submenus are represented by **CommandBarPopup** objects.

You can also add text boxes, drop-down list boxes, and combo boxes to any command bar. These three types of controls are all represented in Visual Basic by **CommandBarComboBox** objects.

Note Although they share similar appearances and behaviors, command bar controls and ActiveX controls are not the same. You cannot add ActiveX controls to command bars, and you cannot add command bar controls to documents or forms.

The built-in command bar controls in Outlook are also represented by **CommandBarButton**, **CommandBarPopup**, and **CommandBarComboBox** objects, but their appearances and behaviors may be different from those you can add yourself. You can modify the location and appearance of built-in controls, you cannot modify their built-in behavior.

CheckBox control

Displays the selection state of an item.

Remarks

Use a **CheckBox** to give the user a choice between two values such as *Yes/No*, *True/False*, or *On/Off*. When the user selects a **CheckBox**, it displays a special mark (such as an **X**) and its current setting is *Yes*, *True*, or *On*. If the user does not select the **CheckBox**, it is empty and its setting is *No*, *False*, or *Off*. Depending on the value of the [TripleState](#) property, a **CheckBox** can also have a [null](#) value.

If a **CheckBox** is [bound](#) to a [data source](#), changing the setting changes the value of that source. A disabled **CheckBox** shows the current value, but is dimmed and does not allow changes to the value from the user interface.

You can also use check boxes inside a group box to select one or more of a group of related items. For example, you can create an order form that contains a list of available items, with a **CheckBox** preceding each item. The user can select a particular item or items by checking the corresponding **CheckBox**.

The default property of a **CheckBox** is the [Value](#) property.

Note The [ListBox](#) also lets you put a check mark by selected options. Depending on your application, you can use the **ListBox** instead of using a group of **CheckBox** controls.

ComboBox control

Combines the features of a [ListBox](#) and a [TextBox](#). The user can enter a new value, as with a **TextBox**, or the user can select an existing value as with a **ListBox**.

Remarks

If a **ComboBox** is [bound](#) to a [data source](#), the **ComboBox** inserts the value entered or selected by the user into that data source. If a multicolumn combo box is bound, then the [BoundColumn](#) property determines which value is stored in the bound data source.

The list in a **ComboBox** consists of rows of data. Each row can have one or more columns, which can appear with or without headings. Some applications do not support column headings, others provide only limited support.

The default property of a **ComboBox** is the [Value](#) property.

Note If you want more than a single line of the list to appear at all times, you might want to use a **ListBox** instead of a **ComboBox**. If you want to use a **ComboBox** and limit values to those in the list, you can set the [Style](#) property of the **ComboBox** so the control looks like a drop-down list box.

CommandButton control

Starts, ends, or interrupts an action or series of actions.

Remarks

Requires VBScript.

Syntax

```
Sub CommandButton_Click( )
```

```
'write event code here
```

End Sub

The macro or event procedure assigned to the **CommandButton's** [Click](#) event determines what the **CommandButton** does. For example, you can create a **CommandButton** that opens another form. You can also display text, a picture, or both on a **CommandButton**.

The default property of a **CommandButton** is the [Value](#) property.

The only event for a **CommandButton** is the **Click** event.

Font Object

Defines the characteristics of the text used by a control.

Each control has its own **Font** object to let you set its text characteristics independently of the characteristics defined for other controls. Use font properties to specify the font name, to set bold, italic, or underlined text, or to adjust the size of the text.

The default property for the **Font** object is the **Name** property.

Frame control

Creates a functional and visual [control group](#).

Remarks

All option buttons in a **Frame** are mutually exclusive, so you can use the **Frame** to create an option group. You can also use a **Frame** to group controls with closely related contents. For example, in an application that processes customer orders, you might use a **Frame** to group the name, address, and account number of customers.

You can also use a **Frame** to create a group of [ToggleButton](#)s, but the toggle buttons are not mutually exclusive.

To create a group of mutually exclusive [OptionButton](#) controls, you can put the buttons in a **Frame** on your form, or you can use the [GroupName](#) property.

Image control

Displays a picture on a form.

Remarks

The **Image** control lets you display a picture as part of the data in a form. For example, you might use an **Image** to display employee photographs in a personnel form.

The **Image** lets you crop, size, or zoom a picture, but does not allow you to edit the contents of the picture. For example, you cannot use the

Image to change the colors in the picture, to make the picture [transparent](#), or to refine the image of the picture. You must use image editing software for these purposes.

The **Image** supports the following file formats:

*.bmp

*.cur

*.gif

*.ico

*.jpg

*.wmf

Note You can also display a picture on a [Label](#). However, a **Label** does not let you crop, size, or zoom the picture.<P>

Label control

Displays descriptive text.

Remarks

A **Label** control on a form displays descriptive text such as titles, captions, pictures, or brief instructions. For example, labels for an address book might include a **Label** for a name, street, or city.

The default property for a **Label** is the [Caption](#) property.

Note You can also display a picture on a **Label**. However, a **Label** does

not let you crop, size, or zoom the picture.

ListBox control

Displays a list of values and lets you select one or more.

Remarks

If the **ListBox** is [bound](#) to a [data source](#), the **ListBox** stores the selected value in that data source.

The **ListBox** can either appear as a list or as a group of [OptionButton](#) controls or [CheckBox](#) controls.

The default property for a **ListBox** is the [Value](#) property.

The default event for a **ListBox** is the [Click](#) event.

Note You can't drop text into a drop-down **ListBox**.

ListBox styles

You can choose between two presentation styles for a **ListBox**. Each style provides different ways for users to select items in the list.

If the style is **Plain**, each item is on a separate row; the user selects an item by highlighting one or more rows.

If the style is **Option**, an [OptionButton](#) or [CheckBox](#) appears at the beginning of each row. With this style, the user selects an item by clicking the option button or check box. Check boxes appear only when the [MultiSelect](#) property is **True**.

MultiPage control

Presents multiple screens of information as a single set.

Remarks

A **MultiPage** is useful when you work with a large amount of information that can be sorted into several categories. For example, use a **MultiPage** to display information from an employment application. One page might contain personal information such as name and address; another page might list previous employers; a third page might list references. The **MultiPage** lets you visually combine related information, while keeping

the entire record readily accessible.

New pages are added to the right of the currently selected page rather than adjacent to it.

A **MultiPage** is a control that contains a [collection](#) of one or more pages.

Each [Page](#) of a **MultiPage** is a form that contains its own controls, and as such, can have a unique layout. Typically, the pages in a **MultiPage** have tabs so the user can select the individual pages.

By default, a **MultiPage** includes two pages, called Page1 and Page2. Each of these is a **Page** object, and together they represent the [Pages](#) collection of the **MultiPage**. If you add more pages, they become part of the same **Pages** collection.

The default property for a **MultiPage** is the [Value](#) property, which returns the index of the currently active **Page** in the **Pages** collection of the **MultiPage**.

Note The **MultiPage** control does not support the [Click](#) event.

OptionButton control

Shows the selection status of one item in a group of choices.

Remarks

Use an **OptionButton** to show whether a single item in a group is selected. Note that each **OptionButton** in a [Frame](#) is mutually exclusive.

If an **OptionButton** is [bound](#) to a [data source](#), the **OptionButton** can show the value of that data source as either *Yes/No*, *True/False*, or *On/Off*. If the user selects the **OptionButton**, the current setting is *Yes*,

True, or *On*. If the user does not select the **OptionButton**, the setting is *No*, *False*, or *Off*. For example, an **OptionButton** in an inventory-tracking application might show whether an item is discontinued. If the **OptionButton** is bound to a data source, then changing the setting changes the value of that data source. A disabled **OptionButton** is dimmed and does not show a value.

Depending on the value of the [TripleState](#) property, an **OptionButton** can also have a [null](#) value.

You can also use an **OptionButton** inside a group box to select one or more of a group of related items. For example, you can create an order form with a list of available items, with an **OptionButton** preceding each item. The user can select a particular item by checking the corresponding **OptionButton**.

The default property for an **OptionButton** is the [Value](#) property.

Page Object

One page of a [MultiPage](#) or a single member of a [Pages](#) collection.

Remarks

Each **Page** object contains its own set of controls and does not necessarily rely on other pages in the collection for information. A **Page** inherits some properties from its container; the value of each inherited property is set by the container.

You can reference a **Page** by its index value. The index value reflects the

ordinal position of the **Page** within the collection. The index of the first **Page** in a collection is 0; the index of the second **Page** is 1; and so on.

The default name for the first **Page** is Page1. The default name for the second **Page** is Page2.

ScrollBar control

Returns or sets the value of another control based on the position of the scroll box.

Remarks

Requires VBScript.

A **ScrollBar** is a stand-alone control you can place on a form. It is visually like the scroll bar you see in certain objects such as a [ListBox](#) or the drop-down portion of a [ComboBox](#). However, unlike the scroll bars in

these controls, the stand-alone **ScrollBar** is not an integral part of any other control.

To use the **ScrollBar** to set or read the value of another control, you must write code that uses the **ScrollBar's Value** property. For example, to use the **ScrollBar** to update the value of a **TextBox**, you can write code that reads the **Value** property of the **ScrollBar** and then sets the **Value** property of the **TextBox**.

The default property for a **ScrollBar** is the **Value** property.

Note To create a horizontal or vertical **ScrollBar**, drag the sizing handles of the **ScrollBar** horizontally or vertically on the form.

SpinButton control

Increments and decrements numbers.

Remarks

Requires VBScript.

Clicking a **SpinButton** changes only the value of the **SpinButton**. You can write code that uses the **SpinButton** to update the displayed value of another control. For example, you can use a **SpinButton** to change the month, the day, or the year shown on a date. You can also use a

SpinButton to scroll through a range of values or a list of items, or to change the value displayed in a text box.

To display a value updated by a **SpinButton**, you must assign the value of the **SpinButton** to the displayed portion of a control, such as the [Caption](#) property of a [Label](#) or the [Text](#) property of a [TextBox](#). To create a horizontal or vertical **SpinButton**, drag the sizing handles of the **SpinButton** horizontally or vertically on the form.

The default property for a **SpinButton** is the [Value](#) property.

Tab Object

A **Tab** is an individual member of a [Tabs](#) collection.

Remarks

Visually, a **Tab** object appears as a rectangle protruding from a larger rectangular area, or as a button adjacent to a rectangular area.

In contrast to a [Page](#), a **Tab** does not contain any controls. Controls that appear within the region bounded by a [TabStrip](#) are contained on the form, as is the **TabStrip**.

You can reference a **Tab** by its index value. The index value reflects the ordinal position of the **Tab** within the collection. The index of the first **Tab** in a collection is 0; the index of the second **Tab** is 1; and so on.

TabStrip control

Presents a set of related controls as a visual group.

Remarks

You can use a **TabStrip** to view different sets of information for related controls.

A **TabStrip** is a control that contains a [collection](#) of one or more tabs.

Each [Tab](#) of a **TabStrip** is a separate object that users can select. Visually, a **TabStrip** also includes a client area that all the tabs in the

TabStrip share.

By default, a **TabStrip** includes two pages, called Tab1 and Tab2. Each of these is a **Tab** object, and together they represent the **Tabs** collection of the **TabStrip**. If you add more pages, they become part of the same **Tabs** collection.

For example, the controls might represent information about a daily schedule for a group of individuals, with each set of information corresponding to a different individual in the group. Set the title of each tab to show one individual's name. Then, you can write code that, after you click a tab, updates the controls to show information about the person identified on the tab.

Note The **TabStrip** is implemented as a **container** of a **Tabs** collection, which in turn contains a group of **Tab** objects. The **TabStrip** control does not support the **Click** event.

The default property for a **TabStrip** is the **SelectedItem** property.

TextBox control

Displays information from a user or from an organized set of data.

Remarks

A **TextBox** is the control most commonly used to display information entered by a user. Also, it can display a set of data, such as a table, query, worksheet, or a calculation result. If a **TextBox** is [bound](#) to a [data source](#), then changing the contents of the **TextBox** also changes the value of the bound data source.

Formatting applied to any piece of text in a **TextBox** will affect all text in the control. For example, if you change the font or point size of any character in the control, the change will affect all characters in the control.

The default property for a **TextBox** is the [Value](#) property.

Tips on using text boxes

The **TextBox** is a flexible control governed by the following properties: [Text](#), [MultiLine](#), [WordWrap](#), and [AutoSize](#).

Text contains the text that's displayed in the text box.

MultiLine controls whether the **TextBox** can display text as a single line or as multiple lines. Newline characters identify where one line ends and another begins. If **MultiLine** is **False** (the default value), the text is truncated instead of wrapped.

WordWrap allows the **TextBox** to wrap lines of text that are longer than the width of the **TextBox** into shorter lines that fit. The default value is **True**.

If you do not use **WordWrap**, the **TextBox** starts a new line of text when it encounters a newline character in the text. If **WordWrap** is turned off, you can have text lines that do not fit completely in the **TextBox**. The **TextBox** displays the portions of text that fit inside its width and truncates the portions of text that do not fit. **WordWrap** is not applicable unless **MultiLine** is **True**.

AutoSize controls whether the **TextBox** adjusts to display all of the text. When using **AutoSize** with a **TextBox**, the width of the **TextBox** shrinks or expands according to the amount of text in the **TextBox** and the font size used to display the text. The default value is **False**.

AutoSize works well in the following situations:

Displaying a caption of one or more lines.

Displaying the contents of a single-line **TextBox**.

Displaying the contents of a multiline **TextBox** that is read-only to the user.

Note Avoid using **AutoSize** with an empty **TextBox** that also uses the **MultiLine** and **WordWrap** properties. When the user enters text into a **TextBox** with these properties, the **TextBox** automatically sizes to a long narrow box one character wide and as long as the line of text.

ToggleButton control

Shows the selection state of an item.

Remarks

Use a **ToggleButton** to show whether an item is selected. If a **ToggleButton** is [bound](#) to a [data source](#), the **ToggleButton** shows the current value of that data source as either *Yes/No*, *True/False*, *On/Off*, or some other choice of two settings. If the user selects the **ToggleButton**, the current setting is *Yes*, *True*, or *On*. If the user does not select the **ToggleButton**, the setting is *No*, *False*, or *Off*. If the **ToggleButton** is

bound to a data source, changing the setting changes the value of that data source. A disabled **ToggleButton** shows a value, but is dimmed and does not allow changes from the user interface.

You can also use a **ToggleButton** inside a [Frame](#) to select one or more of a group of related items. For example, you can create an order form with a list of available items, with a **ToggleButton** preceding each item. The user can select a particular item by selecting the appropriate **ToggleButton**.

The default property of a **ToggleButton** is the [Value](#) property.

The only event for a **ToggleButton** is the [Click](#) event.

ActiveControl Property

Identifies and allows manipulation of the control that has the focus.

Syntax

object.**ActiveControl**

The **ActiveControl** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.

Remarks

The **ActiveControl** property is read-only and is set when you select a control in the interface. You can use **ActiveControl** as a substitute for the control name when setting properties or calling methods.

CanPaste Property

Specifies whether the Clipboard contains data that the object supports.

Syntax

object.**CanPaste**

The **CanPaste** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.

Return Values

The **CanPaste** property return values are:

Value	Description
-------	-------------

True	The object can receive information pasted from the Clipboard.
False	The object cannot receive information pasted from the Clipboard.

Remarks

CanPaste is read-only.

If the Clipboard data is in a format that the object does not support, the **CanPaste** property is **False**. For example, if you try to paste a bitmap into an object that only supports text, **CanPaste** will be **False**.

CanRedo Property

Indicates whether the most recent Undo can be reversed.

Syntax

object.**CanRedo**

The **CanRedo** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.

Return Values

The **CanRedo** property syntax return values are:

Value	Description
-------	-------------

True	The most recent Undo can be reversed.
-------------	---------------------------------------

False	The most recent Undo is irreversible.
--------------	---------------------------------------

Remarks

CanRedo is read-only.

To Redo an action means to reverse an Undo; it does not necessarily mean to repeat the last user action.

CanUndo Property

Indicates whether the last user action can be undone.

Syntax

object.**CanUndo**

The **CanUndo** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.

Return Values

The **CanUndo** property syntax return values are:

Value	Description
-------	-------------

True	The most recent user action can be undone.
False	The most recent user action cannot be undone.

Remarks

CanUndo is read-only.

Many user actions can be undone with the **Undo** command. The **CanUndo** property indicates whether the most recent action can be undone.

CurLine Property

Specifies the current line of a control.

Syntax

object.**CurLine** [= *Long*]

The **CurLine** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.
<i>Long</i>	Optional. Specifies the current line of a control.

Remarks

The current line of a control is the line that contains the insertion point. The number of the first line is 0.

The **CurLine** property is valid when the control has focus.

IMEMode Property

This feature is available in the Simplified Chinese, Traditional Chinese, Korean, and Japanese language versions of Microsoft Office.

Specifies the default run-time mode of the [Input Method Editor](#) (IME) for a control. This property applies only to applications written for Asian languages and is ignored in other applications.

Syntax

object.**IMEMode** [= *fmIMEMode*]

The **IMEMode** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.
<i>fmIMEMode</i>	Optional. The mode of the Input Method Editor (IME).

Settings

The settings for *fmIMEMode* are:

Value	Description
0	Does not control IME (default).
1	IME on.
2	IME off. English mode.
3	IME off. User can't turn on IME by keyboard.
4	IME on with Full-width Hiragana mode.
5	IME on with Full-width Katakana mode.
6	IME on with Half-width Katakana mode.
7	IME on with Full-width Alphanumeric mode.
8	IME on with Half-width Alphanumeric mode.
9	IME on with Full-width Hangul mode.
10	IME on with Half-width Hangul mode.

A setting of 0 indicates that the mode of the IME does not change when the control receives focus at run time. For any other value, the mode of the IME is set to the value specified by the **IMEMode** property when the control receives focus at run time.

ListRows Property

Specifies the maximum number of rows to display in the list.

Syntax

object.**ListRows** [= *Long*]

The **ListRows** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.
<i>Long</i>	Optional. An integer indicating the maximum number of rows. The default value is 8.

Remarks

If the number of items in the list exceeds the value of the **ListRows** property, a scroll bar appears at the right edge of the list-box portion of

the combo box.

VerticalScrollBarSide Property

Specifies whether a vertical scroll bar appears on the right or left side of a frame.

Syntax

object.**VerticalScrollBarSide** [= *VerticalScrollBarSide*]

The **VerticalScrollBarSide** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.
<i>VerticalScrollBarSide</i>	Optional. Where the scroll bar should appear.

Settings

Value	Description
0	Puts the scroll bar on the right side

(default).

1

Puts the scroll bar on the left side.

Zoom Property

Specifies how much to change the size of a displayed object.

Syntax

object.**Zoom** [= *Integer*]

The **Zoom** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.
<i>Integer</i>	The percentage to increase or decrease the displayed image.

Remarks

The value of the **Zoom** property specifies a percentage of image enlargement or reduction by which an image display should change.

Values from 10 to 400 are valid. The value specified is a percentage of the object's original size; thus, a setting of 400 means you want to enlarge the image to four times its original size (or 400 percent), while a setting of 10 means you want to reduce the image to one-tenth of its original size (or 10 percent).

Create an instant workgroup solution using public folders

By using custom views in a [public folder](#), you can take an existing [item](#), such as a contact, and turn it into a workgroup [form](#) from which any user of the public folder can get information or to which they can add data. With this method, you create an appointment, task, or contact in a public folder, create a custom view for the information in the folder, and then give permission to those who you want to use the folder.

Click a step below to begin:

[Step 1: Create a public folder](#)

[Step 2: Create an Outlook item in a public folder](#)

[Step 3: Create and use a custom view in a public folder](#)

Create custom forms by using VBScript

Microsoft VBScript is a subset of Visual Basic for Applications. You can use VBScript to create procedures that control Microsoft Outlook folders, objects, [items](#), and properties. VBScript in Outlook requires a special object syntax that has some differences from referencing objects in Visual Basic for Applications.

Learn about [the Outlook object model](#).

You can choose the Outlook [item](#) on which to base your custom [form](#).

You can extend Outlook forms by using custom controls from the **Control Toolbox**. Outlook forms can use most of the properties and methods that come with the controls. Since controls cannot store values, to store the value you need to [bind](#) the control to an Outlook field.

The Outlook object browser displays the [classes](#), properties, methods, events, and [constants](#) available from the Outlook object library. The object browser lets you view and use objects in the Microsoft Script Editor and obtain information about the syntax for using the object.

Learn about [viewing and using the object browser](#).

The Microsoft Script Debugger provides you with a comprehensive debugging environment for testing and correcting errors in the VBScript code for your forms.

Learn about [the Script Debugger](#).

About the object environment

There are two ways to write code for Outlook:

From outside the application, such as by using Microsoft Visual Basic or Microsoft Visual Basic for Applications in Microsoft Excel or another application.

From inside the application, such as by using Visual Basic for Applications or by using VBScript with an Outlook form.

Learn more about [the differences between using Visual Basic for Applications and VBScript](#)

The major components of the Outlook object model are:

Application The top of the object hierarchy that represents the entire application and creates items and objects. For example, in Outlook Visual Basic for Applications or VBScript:

```
Application.CreateItem(1).Display
```

NameSpace Represents the MAPI message store where all the Outlook folders are stored on and off Outlook and for referencing the default folders. For example, this code references the active user in Outlook Visual Basic for Applications or VBScript:

```
Application.GetNameSpace("MAPI").CurrentUser
```

Explorer Represents the Outlook window. Enables you to show, retrieve, and delete messages. The following code shows the active Outlook window in Outlook Visual Basic for Applications or VBScript:

```
Application.ActiveExplorer.Display
```

Folders There are two folder objects, the **Folders** collection object and the **MAPIFolder** object that enables you to work with a collection of folders named Personal Folders in Outlook Visual Basic for Applications or VBScript:

```
Application.GetNameSpace("MAPI").Folders("Personal Folders")
```

Outlook items There are two item objects, the **Items** collection object that contains the item objects that represent the standard item types in a message. In VBScript, the active item is assumed, so you can refer to it directly. For example, this code sets the Subject field of the active item:

```
Item.Subject = "New Subject"
```

Inspector References forms. Use to show forms and pages. For example, in Outlook Visual Basic for Applications or VBScript:

Application.ActiveInspector.SetCurrentFormPage("Opt

AddressEntry	Each AddressEntry object in the AddressEntries collection process to which the messaging system can deliver messages.
AddressList	The AddressList object is an address book that contains a hierarchy is available through the parent AddressLists collection.
Exception	The Exception object holds information about one instance of an exception to a recurring series. Unlike most of the other objects.
Control	There are two control objects, the Controls collection object on a page and the specific control object that enables you to set the Caption of a CommandButton control named "CommandButton". Item.GetInspector.ModifiedFormPages("Test").Controls

Click Event

The **Click** event occurs only if it has been defined for a control in the Script Editor.

The [TabStrip](#) and [MultiPage](#) controls do not support the **Click** event. However, individual the [Page](#) objects of the **MultiPage** control do support the click event.

[ScrollBars](#) and [SpinButtons](#) do not support the **Click** event but you can [bind](#) them to fields and use the **CustomPropertyChange** event.

The **Click** event occurs in one of two cases:

The user clicks a control with the mouse.

The user definitively selects a value for a control with more than one possible value.

Syntax

Sub *object_Click()*

The **Click** event syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.

Example:

```
Sub CommandButton1_Click()  
    MsgBox "You just clicked my button! "  
End Sub
```

Remarks

Of the two cases where the **Click** event occurs, the first case applies to the [CommandButton](#), [Frame](#), [Image](#), [Label](#), and [Page](#).

The second case applies to the [CheckBox](#), [ComboBox](#), [ListBox](#), and [ToggleButton](#). It also applies to an [OptionButton](#) when the value changes to **True**.

The following are examples of actions that initiate the **Click** event:

Clicking a blank area of a form or a disabled control (other than a list box) on the form.

Clicking a **CommandButton**.

Pressing the SPACEBAR when a **CommandButton** has the focus.

Clicking a control with the left mouse button (left-clicking).

Pressing ENTER on a form that has a command button whose **Default** property is set to **True**, as long as no other command button has the focus.

Pressing ESC on a form that has a command button whose **Cancel** property is set to **True**, as long as no other command button has the focus.

Pressing a control's [accelerator key](#).

For some controls, the **Click** event occurs when the [Value](#) property changes. However, using the **PropertyChange** or **CustomPropertyChange** event is the preferred technique for detecting a new value for a property. The following are examples of actions that initiate the **Click** event due to assigning a new value to a control:

Clicking a **CheckBox** or **ToggleButton**, pressing the SPACEBAR when one of these controls has the focus, pressing the accelerator key for one of these controls, or changing the value of the control in code.

Changing the value of an **OptionButton** to **True**. Setting one **OptionButton** in a group to **True** sets all other buttons in the group to **False**, but the **Click** event occurs only for the button whose value changes to **True**.

Selecting a value for a **ComboBox** or **ListBox** so that it unquestionably matches an item in the control's drop-down list. For example, if a list is not sorted, the first match for characters typed in the edit region may not be the only match in the list, so choosing such a value does not initiate the **Click** event. In a sorted list, you can use entry-matching to ensure that a selected value is a unique match for text the user types.

The **Click** event is not initiated when **Value** is set to **Null**.

Note Left-clicking changes the value of a control, thus it initiates the **Click** event. Right-clicking does not change the value of the control, so it does not initiate the **Click** event.

Also Note If you bind a **ListBox**, **ComboBox**, **OptionButton**, or **CheckBox** to a field, then the **Click** event does not fire. You need to use the **PropertyChange** or **CustomPropertyChange** event to detect the change via code.

Example:

```
Sub Item_PropertyChange(ByVal Name)
Set MyListBox = Item.GetInspector.ModifiedFormPages("Message").(
Select Case Name
    Case "Mileage"
        Item.CC = MyListBox.Value
```

```
        Item.Subject = MyListBox.Value  
    Case Else  
End Select  
End Sub
```


How to use the Outlook object browser

To view the Outlook object browser:

- 1 Open a [form](#) in [design mode](#).
- 2 In the **Form** menu, click **View Code** to view the Script Editor.
- 3 In the Script Editor, click **Object Browser** on the **Script** menu or press **F2**.

All of the available Outlook objects are listed in the **Classes** pane of the object browser in alphabetical order.

To view the members of an object, select the object in the **Classes** pane. The members of this object appear in alphabetical order in the **Members of** pane. The heading at the top of this pane will reflect the name of the object that you select. For example, if you select the **AppointmentItem** object in the **Classes** pane, the heading of the **Members of** pane will appear as **Members of AppointmentItem**.

The details pane shows the definition of the selected member. This text is read-only and cannot be copied and pasted into the Script Editor.

To insert an item from the object browser into the Script Editor:

- 1 In the Script Editor, position your cursor at the location for insertion.
- 2 Select the desired object in the **Classes** pane.
- 3 Select the desired member of this object in the **Members of** pane.
- 4 Click the **Insert** button.

Note The **Insert** button remains unavailable until a member of the object is selected.

How to use the Outlook script debugger

To use the script debugger, open a [form](#) in [design mode](#).

To add VBScript to the form, on the **Form** menu, click on **View Code**. In the Script Editor window, add the necessary VBScript for your form.

For more information about adding VBScript to the form, click here [»](#)

On the **Form** menu, click **Run This Form**.

While your form is in [run mode](#), on the **Tools** menu, select **Forms** and click **Script Debugger**.

You can also use the Stop statement in code to launch the debugger.

Note If there is no VBScript for this form, the **Script Debugger** menu item will be unavailable.

The VBScript for this form will appear in a read-only window. You cannot change text in this window.

On the Help menu of the Microsoft Script Debugger, select Help Topics for more detailed information about:

Debugging scripts

Viewing source code

Controlling program execution

Viewing and changing values

Viewing and controlling program flow

TriState Property

Determines whether a user can specify, from the user interface, the [Null](#) state for a [CheckBox](#) or [ToggleButton](#).

Syntax

object.**TriState** [= *Boolean*]

The **TriState** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.
<i>Boolean</i>	Optional. Whether the control supports the Null state.

Settings

The settings for *Boolean* are:

Value	Description
True	The button clicks through three states.
False	The button only supports True and False (default).

Remarks

Although the **TriState** property exists on the [OptionButton](#), the property does not affect the action of the control. Regardless of the value of **TriState**, you cannot set the control to **Null** through the user interface.

When the **TriState** property is **True**, a user can choose from the values of **Null**, **True**, and **False**. The null value is displayed as a shaded button.

When **TriState** is **False**, the user can choose either **True** or **False**.

A control set to **Null** does not initiate the [Click](#) event.

Regardless of the property setting, the **Null** value can always be assigned programmatically to an **OptionButton**, **CheckBox** or **ToggleButton**, causing that control to appear shaded.

Value Property

Specifies the state or content of a given control.

Syntax

object.**Value** [= *Variant*]

The **Value** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.
<i>Variant</i>	Optional. The state or content of the control.

Settings

Control	Description
CheckBox	An integer value indicating whether the item

	is selected:
	Null Indicates the item is in a null state, neither selected nor cleared .
	-1 True. Indicates the item is selected.
	0 False. Indicates the item is cleared.
OptionButton	Same as CheckBox .
ToggleButton	Same as CheckBox .
ScrollBar	An integer between the values specified for the Max and Min properties.
SpinButton	Same as ScrollBar .
ComboBox , ListBox	The value in the BoundColumn of the currently selected rows.
CommandButton	Always False .
MultiPage	An integer indicating the currently active page. Zero (0) indicates the first page. The maximum value is one less than the number of pages.
TextBox	The text in the edit region.

Remarks

For a **CommandButton**, setting the **Value** property to **True** in a macro or procedure initiates the button's [Click](#) event.

For a **ComboBox**, changing the contents of **Value** does not change the value of [BoundColumn](#). To add or delete entries in a **ComboBox**, you can use the [AddItem](#) or [RemoveItem](#) method.

For a **TextBox**, any value you assign to the [Text](#) property is also assigned to the **Value** property.

Value cannot be used with a multi-select **ListBox**.

BoundColumn Property

Identifies the source of data in a multicolumn [ComboBox](#) or [ListBox](#).

Syntax

object.**BoundColumn** [= *Variant*]

The **BoundColumn** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.
<i>Variant</i>	Optional. Indicates how the BoundColumn value is selected.

Settings

The settings for *Variant* are:

--	--

Value	Description
0	Assigns the value of the ListIndex property to the control.
1 or greater	Assigns the value from the specified column to the control. Columns are numbered from 1 when using this property (default).

Remarks

When the user chooses a row in a multicolumn **ListBox** or **ComboBox**, the **BoundColumn** property identifies which item from that row to store as the value of the control. For example, if each row contains 8 items and **BoundColumn** is 3, the system stores the information in the third column of the currently-selected row as the value of the object.

You can display one set of data to users but store different, associated values for the object by using the **BoundColumn** and the [TextColumn](#) properties. **TextColumn** identifies the column of data displayed in a **ComboBox** or **ListBox**; **BoundColumn** identifies the column of associated data values stored for the control. For example, you could set up a multicolumn **ListBox** that contains the names of holidays in one column and dates for the holidays in a second column. To present the holiday names to users, specify the first column as the **TextColumn**. To store the dates of the holidays, specify the second column as the **BoundColumn**.

If the control is [bound](#) to a [data source](#), the value in the column specified by **BoundColumn** is stored in the data source named in the **ControlSource** property.

The **ListIndex** value retrieves the number of the selected row. For example, if you want to know the row of the selected item, set **BoundColumn** to 0 to assign the number of the selected row as the value of the control. Be sure to retrieve a current value, rather than relying on a previously saved value, if you are referencing a list whose contents might change.

The **Column**, **List**, and **ListIndex** properties all use zero-based

numbering. That is, the value of the first item (column or row) is zero; the value of the second item is one, and so on. This means that if **BoundColumn** is set to 3, you could access the value stored in that column using the expression `Column(2)`.

Style Property

For [ComboBox](#), specifies how the user can choose or set the control's value. For [MultiPage](#) and [TabStrip](#), identifies the style of the tabs on the control.

Syntax

object.**Style** [=*Style*]

The **Style** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.
<i>Style</i>	Optional. Specifies how a user sets the value of a ComboBox .

Settings

The settings for **ComboBox** are:

Value	Description
0	The ComboBox behaves as a drop-down combo box. The user can type a value in the edit region or select a value from the drop-down list (default).
2	The ComboBox behaves as a list box. The user must choose a value from the list.

The settings for **MultiPage** and **TabStrip** are::

Value	Description
0	Displays tabs on the tab bar (default).
1	Displays buttons on the tab bar.
2	Does not display the tab bar.

GroupName Property

Creates a group of mutually exclusive [OptionButton](#) controls.

Syntax

object.**GroupName** [= *String*]

The **GroupName** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid OptionButton .
<i>String</i>	Optional. The name of the group that includes the OptionButton . Use the same setting for all buttons in the group. The default setting is an empty string.

Remarks

To create a group of mutually exclusive **OptionButton** controls, you can

put the buttons in a [Frame](#) on your form, or you can use the **GroupName** property. **GroupName** is more efficient for the following reasons:

You do not have to include a **Frame** for each group. By not using a **Frame**, you reduce the number of controls on the form, and in turn, improve performance and reduce the size of the form.

You have more design flexibility. If you use a **Frame** to create the group, all the buttons must be inside the **Frame**. If you want more than one group, you must have one **Frame** for each group. However, if you use **GroupName** to create the group, the group can include option buttons anywhere on the form. If you want more than one group, specify a unique name for each group; you can still place the individual controls anywhere on the form.

You can create buttons with [transparent](#) backgrounds, which can improve the visual appearance of your form. The **Frame** is not a transparent control.

Regardless of which method you use to create the group of buttons, clicking one button in a group sets all other buttons in the same group to **False**. All option buttons with the same **GroupName** within a single [container](#) are mutually exclusive. You can use the same group name in two containers, but doing so creates two groups (one in each container) rather than one group that includes both containers.

For example, assume your form includes some option buttons and a [MultiPage](#) that also includes option buttons. The option buttons on the **MultiPage** are one group and the buttons on the form are another group. The two groups do not affect each other. Changing the setting of a button on the **MultiPage** does not affect the buttons on the form.

Caption Property

Descriptive text that appears on an object to identify or describe it.

Syntax

object.**Caption** [= *String*]

The **Caption** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.
<i>String</i>	Optional. A string expression that evaluates to the text displayed as the caption.

Settings

The default setting for a control is a unique name based on the type of control. For example, `CommandButton1` is the default caption for the first

command button in a form.

Remarks

The text identifies or describes the object with which it is associated. For buttons and labels, the **Caption** property specifies the text that appears in the control. For **Page** and **Tab** objects, it specifies the text that appears on the tab. For the **Explorer** object, it specifies the text that appears in the explorer's title bar.

If a control's caption is too long, the caption is truncated. If a form's caption is too long for the title bar, the title is displayed with an ellipsis.

The [ForeColor](#) property of the control determines the color of the text in the caption.

Tip If a control has both the **Caption** and [AutoSize](#) properties, setting **AutoSize** to **True** automatically adjusts the size of the control to frame the entire caption.

MultiSelect Property

Indicates whether the object permits multiple selections.

Syntax

object.**MultiSelect** [=MultiSelect]

The **MultiSelect** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.
<i>MultiSelect</i>	Optional. The selection mode that the control uses.

Settings

The settings for *MultiSelect* are:

--	--

Value	Description
0	Only one item can be selected (default).
1	Pressing the SPACEBAR or clicking selects or deselects an item in the list.
2	Pressing SHIFT and clicking the mouse, or pressing SHIFT and one of the arrow keys, extends the selection from the previously selected item to the current item. Pressing CTRL and clicking the mouse selects or deselects an item.

Remarks

When the **MultiSelect** property is set to *Extended* or *Simple*, you must use the list box's **Selected** property to determine the selected items. Also, the **Value** property of the control is always **Null**.

The **ListIndex** property returns the index of the row with the keyboard focus.<P>

Pages Collection

A **Pages** collection includes all the pages of a [MultiPage](#).

Remarks

Each **Pages** collection provides the features to manage the number of pages in the collection and to identify the page that is currently in use.

The default value of the **Pages** collection identifies the current **Page** of a collection.

You can reference a **Page** by its index value. The index value reflects the

ordinal position of the **Page** within the collection. The index of the first **Page** in a collection is 0; the index of the second **Page** is 1; and so on.

Text Property

Returns or sets the text in a [TextBox](#). Changes the selected row in a [ComboBox](#) or [ListBox](#).

Syntax

object.**Text** [= *String*]

The **Text** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.
<i>String</i>	Optional. A string expression specifying text. The default value is a zero-length string ("").

Remarks

For a **TextBox**, any value you assign to the **Text** property is also

assigned to the [Value](#) property.

For a **ComboBox**, you can use **Text** to update the value of the control. If the value of **Text** matches an existing list entry, the value of the [ListIndex](#) property (the index of the current row) is set to the row that matches **Text**. If the value of **Text** does not match a row, **ListIndex** is set to -1 .

For a **ListBox**, the value of **Text** must match an existing list entry. Specifying a value that does not match an existing list entry causes an error.

When the **Text** property of a **ComboBox** changes (such as when a user types an entry into the control), the new text is compared to the column of data specified by [TextColumn](#).

You cannot use **Text** to change the value of an entry in a **ComboBox** or **ListBox**; use the **Column** or **List** property for this purpose.

The [ForeColor](#) property determines the color of the text.

Tabs Collection

A **Tabs** collection includes all [Tabs](#) of a [TabStrip](#).

Remarks

Each **Tabs** collection provides the features to manage the number of tabs in the collection and to identify the tab that is currently in use.

The default value of the **Tabs** collection identifies the current **Tab** of a collection.

You can reference a **Tab** by its index value. The index value reflects the

ordinal position of the **Tab** within the collection. The index of the first **Tab** in a collection is 0; the index of the second **Tab** is 1; and so on.

SelectedItem Property

Returns the currently selected **Tab** or **Page** object.

Syntax

object.**SelectedItem**

The **SelectedItem** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid TabStrip or MultiPage .

Remarks

The **SelectedItem** property is read-only. Use **SelectedItem** to programmatically control the currently selected **Tab** or **Page** object. For example, you can use **SelectedItem** to assign values to properties of a

Tab or Page object.

MultiLine Property

Specifies whether a control can accept and display multiple lines of text.

Syntax

object.**MultiLine** [= *Boolean*]

The **MultiLine** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.
<i>Boolean</i>	Optional. Whether the control supports more than one line of text.

Settings

The settings for *Boolean* are:

--	--

Value	Description
True	The text is displayed across multiple lines (default).
False	The text is not displayed across multiple lines.

Remarks

A multiline [TextBox](#) allows absolute line breaks and adjusts its quantity of lines to accommodate the amount of text it holds. If needed, a multiline control can have vertical scroll bars.

A single-line **TextBox** doesn't allow absolute line breaks and doesn't use vertical scroll bars.

For controls that support the **MultiLine** property as well as the [WordWrap](#) property, **WordWrap** is ignored when **MultiLine** is **False**.

Single-line controls ignore the value of the **WordWrap** property.

Note If you change **MultiLine** to **False** in a multiline **TextBox**, all the characters in the **TextBox** will be combined into one line, including non-printing characters (such as carriage returns and new-lines).

The [EnterKeyBehavior](#) and **MultiLine** properties are closely related. The **EnterKeyBehavior** values of **True** and **False** only apply if **MultiLine** is **True**. If **MultiLine** is **False**, pressing ENTER always moves the **focus** to the next control in the [tab order](#) regardless of the value of **EnterKeyBehavior**.

The effect of pressing CTRL+ENTER also depends on the value of **MultiLine**. If **MultiLine** is **True**, pressing CTRL+ENTER creates a new line regardless of the value of **EnterKeyBehavior**. If **MultiLine** is **False**, pressing CTRL+ENTER has no effect.

The [TabKeyBehavior](#) and **MultiLine** properties are closely related. The values described above only apply if **MultiLine** is **True**. If **MultiLine** is **False**, pressing TAB always moves the **focus** to the next control in the [tab order](#) regardless of the value of **TabKeyBehavior**.

The effect of pressing CTRL+TAB also depends on the value of

MultiLine. If **MultiLine** is **True**, pressing CTRL+TAB creates a new line regardless of the value of **TabKeyBehavior**. If **MultiLine** is **False**, pressing CTRL+TAB has no effect.

WordWrap Property

Indicates whether the contents of a control automatically wrap at the end of a line.

Syntax

object.**WordWrap** [= *Boolean*]

The **WordWrap** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.
<i>Boolean</i>	Optional. Whether the control expands to fit the text.

Settings

The settings for *Boolean* are:

Value	Description
True	The text wraps (default).
False	The text does not wrap.

Remarks

For controls that support the [MultiLine](#) property as well as the **WordWrap** property, **WordWrap** is ignored when **MultiLine** is **False**.

AutoSize Property

Specifies whether an object automatically resizes to display its entire contents.

Syntax

object.**AutoSize** [= *Boolean*]

The **AutoSize** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.
<i>Boolean</i>	Optional. Whether the control is resized.

Settings

The settings for *Boolean* are:

Value	Description
True	Automatically resizes the control to display its entire contents.
False	Keeps the size of the control constant. Contents are clipped when they exceed the area of the control (default).

Remarks

For controls with captions, the **AutoSize** property specifies whether the control automatically adjusts to display the entire caption.

For controls without captions, this property specifies whether the control automatically adjusts to display the information stored in the control. In a [ComboBox](#), for example, setting **AutoSize** to **True** automatically sets the width of the display area to match the length of the current text.

For a single-line [TextBox](#), setting **AutoSize** to **True** automatically sets the width of the display area to the length of the text in the text box.

For a multiline **TextBox** that contains no text, setting **AutoSize** to **True** automatically displays the text as a column. The width of the text column is set to accommodate the widest letter of that font size. The height of the text column is set to display the entire text of the **TextBox**.

For a multiline **TextBox** that contains text, setting **AutoSize** to **True** automatically enlarges the **TextBox** vertically to display the entire text. The width of the **TextBox** does not change.

Note If you manually change the size of a control while **AutoSize** is **True**, the manual change overrides the size previously set by **AutoSize**.

Step 1: Create a public folder

To create a [public folder](#), you must have permission to create folders in an existing public folder. For information about how to obtain permission, see your administrator.

1. On the **File** menu, select **New**, and then click **Folder**.
(CTRL+SHIFT+E)
2. In the **Name** box, enter a name for the folder.
3. In the **Folder contains** box, click the type of [item](#) that you want the folder to contain. A folder can only contain one type of item.
4. Click the **Select Folder** button, and then click the public folder in which you want your new public folder to appear.
5. If you do not want to add a [Shortcut](#) for the public folder to your [Outlook Bar](#), click **No** in the **Add shortcut to Outlook Bar?** box.

Note You can copy a private folder to a public folder for quick creation of a public folder with existing items.

To go on to Step 2, click 

Step 2: Create an Outlook item in a public folder

1. If the [public folder](#), that you want to use does not exist, you can create it.

[How?](#)

2. In the Folder List or on the [Outlook Bar](#), select the public folder in which you want to add the new [item](#).
3. On the **File** menu, point to **New**, and then click the item that you want to create.

Tip To quickly let others add the public folder to their Public Folders folder, you can send a shortcut in a mail message.

To go on to Step 3, click .

Step 3: Create and use a custom view in a public folder

With custom views, you can arrange information in a [public folder](#), exactly how you want it. You can set which view is initially shown when a user opens the public folder, and you can remove all the standard views from a public folder and only show the custom views you create. By setting the permission for a public folder, you can determine who has access to the folder.

AddItem Method

For a single-column [ListBox](#) or [ComboBox](#), the **AddItem** method adds an item to the list. For a multicolumn **ListBox** or **ComboBox**, this method adds a row to the list.

Syntax

Variant = *object*.**AddItem**([*item* [, *varIndex*]])

The **AddItem** method syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.
<i>item</i>	Optional. Specifies the item or row to add. The number of the first item or row is 0; the number of the second item or row is 1, and so on.
<i>varIndex</i>	Optional. Integer specifying the position within the

object where the new item or row is placed.

Remarks

If you supply a valid value for *varIndex*, the **AddItem** method places the item or row at that position within the list. If you omit *varIndex*, the method adds the item or row at the end of the list.

The value of *varIndex* must not be greater than the value of the [ListCount](#) property.

For a multicolumn **ListBox** or **ComboBox**, **AddItem** inserts an entire row, that is, it inserts an item for each column of the control. To assign values to an item beyond the first column, use the [List](#) or [Column](#) property and specify the row and column of the item.

If the control is [bound](#) to data, the **AddItem** method fails.

Note You can add more than one row at a time to a **ComboBox** or **ListBox** by using **List**.

RemoveItem Method

Removes a row from the list in a list box or combo box.

Syntax

Boolean = *object*.**RemoveItem**(*index*)

The **RemoveItem** method syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.
<i>index</i>	Required. Specifies the row to delete. The number of the first row is 0; the number of the second row is 1, and so on.

This method will not remove a row from the list if the [ListBox](#) is data bound (that is, when the [RowSource](#) property specifies a data source for the **ListBox**).

ListIndex Property

Identifies the currently selected item in a [ListBox](#) or [ComboBox](#).

Syntax

object.ListIndex [= *Variant*]

The **ListIndex** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.
<i>Variant</i>	Optional. The currently selected item in the control.

Remarks

The **ListIndex** property contains an index of the selected row in a list. Values of **ListIndex** range from -1 to one less than the total number of

rows in a list (that is, **ListCount** – 1). When no rows are selected, **ListIndex** returns –1. When the user selects a row in a **ListBox** or **ComboBox**, the system sets the **ListIndex** value. The **ListIndex** value of the first row in a list is 0, the value of the second row is 1, and so on.

Note If you use the **MultiSelect** property to create a **ListBox** that allows multiple selections, the **Selected** property of the **ListBox** (rather than the **ListIndex** property) identifies the selected rows. The **Selected** property is an **array** with the same number of values as the number of rows in the **ListBox**. For each row in the list box, **Selected** is **True** if the row is selected and **False** if it is not. In a **ListBox** that allows multiple selections, **ListIndex** returns the index of the row that has **focus**, regardless of whether that row is currently selected.

The **ListIndex** value is also available by setting the **BoundColumn** property to 0 for a combo box or list box. If **BoundColumn** is 0, the underlying **data source** to which the combo box or list box is **bound** contains the same list index value as **ListIndex**.

TextColumn Property

Identifies the column in a [ComboBox](#) or [ListBox](#) to display to the user.

Syntax

object.**TextColumn** [= *Variant*]

The **TextColumn** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.
<i>Variant</i>	Optional. The column to be displayed.

Settings

Values for the **TextColumn** property range from -1 to the number of columns in the list. The **TextColumn** value for the first column is 1, the

value of the second column is 2, and so on. Setting **TextColumn** to 0 displays the [ListIndex](#) values. Setting **TextColumn** to -1 displays the first column that has a [ColumnWidths](#) value greater than 0.

Remarks

When the user selects a row from a **ComboBox** or **ListBox**, the column referenced by **TextColumn** is stored in the [Text](#) property. For example, you could set up a multicolumn **ListBox** that contains the names of holidays in one column and dates for the holidays in a second column. To present the holiday names to users, specify the first column as the **TextColumn**. To store the dates of the holidays, specify the second column as the [BoundColumn](#).

When the **Text** property of a **ComboBox** changes (such as when a user types an entry into the control), the new text is compared to the column of data specified by **TextColumn**.

ForeColor Property

Specifies the [foreground color](#) of an object.

Syntax

object.ForeColor [= *Long*]

The **ForeColor** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.
<i>Long</i>	Optional. A value or constant that determines the foreground color of an object.

Settings

You can use any integer that represents a valid color. You can also specify a color by using the **RGB** function with red, green, and blue color

components. The value of each color component is an integer that ranges from zero to 255. For example, you can specify teal blue as the integer value 4966415 or as red, green, and blue color components 15, 200, 75, as shown in the following example.

RGB(15,200,75)

Remarks

Use the **ForeColor** property for controls on forms to make them easy to read or to convey a special meaning. For example, if a text box reports the number of units in stock, you can change the color of the text when the value falls below the reorder level.

For a [ScrollBar](#) or [SpinButton](#), **ForeColor** sets the color of the arrows. For a [Frame](#), **ForeColor** changes the color of the caption. For a **Font** object, **ForeColor** determines the color of the text.

Selected Property

Returns or sets the selection state of items in a [ListBox](#).

Syntax

object.**Selected**(*index*) [= *Boolean*]

The **Selected** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.
<i>index</i>	Required. An integer with a range from 0 to one less than the number of items in the list.
<i>Boolean</i>	Optional. Whether an item is selected.

Settings

The settings for *Boolean* are:

Value	Description
True	The item is selected.
False	The item is not selected.

Remarks

The **Selected** property is useful when users can make multiple selections. You can use this property to determine the selected rows in a multi-select list box. You can also use this property to select or deselect rows in a list from code.

The default value of this property is based on the current selection state of the **ListBox**.

For single-selection list boxes, the [Value](#) or [ListIndex](#) properties are recommended for getting and setting the selection. In this case, **ListIndex** returns the index of the selected item. However, in a multiple selection, **ListIndex** returns the index of the row contained within the [focus](#) rectangle, regardless of whether the row is actually selected.

When a list box control's [MultiSelect](#) property is set to *None*, only one row can have its **Selected** property set to **True**.

Entering a value that is out of range for the index does not generate an error message, but does not set a property for any item in the list.

EnterKeyBehavior Property

Defines the effect of pressing ENTER in a [TextBox](#).

Syntax

object.EnterKeyBehavior [= *Boolean*]

The **EnterKeyBehavior** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.
<i>Boolean</i>	Optional. Specifies the effect of pressing ENTER.

Settings

The settings for *Boolean* are:

--	--

Value	Description
True	Pressing ENTER creates a new line.
False	Pressing ENTER moves the focus to the next object in the tab order (default).

Remarks

The **EnterKeyBehavior** and **MultiLine** properties are closely related. The values described above only apply if **MultiLine** is **True**. If **MultiLine** is **False**, pressing ENTER always moves the **focus** to the next control in the **tab order** regardless of the value of **EnterKeyBehavior**.

The effect of pressing CTRL+ENTER also depends on the value of **MultiLine**. If **MultiLine** is **True**, pressing CTRL+ENTER creates a new line regardless of the value of **EnterKeyBehavior**. If **MultiLine** is **False**, pressing CTRL+ENTER has no effect.

TabKeyBehavior Property

Determines whether tabs are allowed in the edit region.

Syntax

object.**TabKeyBehavior** [= *Boolean*]

The **TabKeyBehavior** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.
<i>Boolean</i>	Optional. The effect of pressing TAB.

Settings

The settings for *Boolean* are:



Value	Description
True	Pressing TAB inserts a tab character in the edit region.
False	Pressing TAB moves the focus to the next object in the tab order (default).

Remarks

The **TabKeyBehavior** and **MultiLine** properties are closely related. The values described above only apply if **MultiLine** is **True**. If **MultiLine** is **False**, pressing TAB always moves the **focus** to the next control in the **tab order** regardless of the value of **TabKeyBehavior**.

The effect of pressing CTRL+TAB also depends on the value of **MultiLine**. If **MultiLine** is **True**, pressing CTRL+TAB creates a new line regardless of the value of **TabKeyBehavior**. If **MultiLine** is **False**, pressing CTRL+TAB has no effect.

ListCount Property

Returns the number of list entries in a control.

Syntax

object.ListCount

The **ListCount** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.

Remarks

The **ListCount** property is read-only. **ListCount** is the number of rows over which you can scroll. **ListCount** is always one greater than the largest value for the [ListIndex](#) property, because index numbers begin

with 0 and the count of items begins with 1. If no item is selected, **ListCount** is 0 and **ListIndex** is -1.

List Property

Returns or sets the list entries of a [ListBox](#) or [ComboBox](#).

Syntax

object.List(*row*, *column*) [= *Variant*]

The **List** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.
<i>row</i>	Required. An integer with a range from 0 to one less than the number of entries in the list.
<i>column</i>	Required. An integer with a range from 0 to one less than the number of columns.
<i>Variant</i>	Optional. The contents of the specified entry in the ListBox or ComboBox .

Settings

Row and column numbering begins with zero. That is, the row number of the first row in the list is zero; the column number of the first column is zero. The number of the second row or column is 1, and so on.

Remarks

The **List** property works with the [ListCount](#) and [ListIndex](#) properties. Use **List** to access list items. A list is a variant [array](#); each item in the list has a row number and a column number.

Initially, **ComboBox** and **ListBox** contain empty lists.

Note To specify items you want to display in a **ComboBox** or **ListBox**, use the [AddItem](#) method. To remove items, use the [RemoveItem](#) method.

Use **List** to copy an entire two-dimensional array of values to a control. Use **AddItem** to load a one-dimensional array or to load an individual element.

Column Property

Specifies one or more items in a [ListBox](#) or [ComboBox](#).

Syntax

object.**Column**(*column*, *row*) [= *Variant*]

The **Column** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.
<i>column</i>	Optional. An integer with a range from 0 to one less than the total number of columns.
<i>row</i>	Optional. An integer with a range from 0 to one less than the total number of rows.
<i>Variant</i>	Optional. Specifies a single value, a column of values, or a two-dimensional array to load into a

ListBox or ComboBox.

Settings

If you specify both the column and row values, **Column** reads or writes a specific item.

If you specify only the column value, the **Column** property reads or writes the specified column in the current row of the object. For example, `MyListBox.Column(3)` reads or writes the third column in `MyListBox`.

Column returns a *Variant* from the cursor. When a built-in [cursor](#) provides the value for *Variant* (such as when using the [AddItem](#) method), the value is a string. When an external cursor provides the value for *Variant*, formatting associated with the data is not included in the *Variant*.

Remarks

You can use **Column** to assign the contents of a combo box or list box to another control, such as a text box. For example, you can set the **ControlSource** property of a text box to the value in the second column of a list box.

If the user makes no selection when you refer to a column in a combo box or list box, the **Column** setting is **Null**. You can check for this condition by using the **IsNull** function.

You can also use **Column** to copy an entire two-dimensional [array](#) of values to a control. This syntax lets you quickly load a list of choices rather than individually loading each element of the list using **AddItem**.

Note When copying data from a two-dimensional array, **Column** transposes the contents of the array in the control so that the contents of `ListBox1.Column(X, Y)` is the same as `MyArray(Y, X)`. You can also use [List](#) to copy an array without transposing it.

RowSource Property

Specifies the source providing a list for a [ComboBox](#) or [ListBox](#).

Syntax

object.RowSource [= *String*]

The **RowSource** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.
<i>String</i>	Optional. The source of the list for the ComboBox or ListBox .

Remarks

The **RowSource** property accepts worksheet ranges from Microsoft Excel.

ColumnWidths Property

Specifies the width of each column in a multicolumn [ComboBox](#) or [ListBox](#).

Syntax

object.ColumnWidths [= *String*]

The **ColumnWidths** property syntax has these parts:

Part	Description
<i>object</i>	Required. A valid object.
<i>String</i>	Optional. Sets the column width in points. A setting of -1 or blank results in a calculated width. A width of 0 hides a column. To specify a different unit of measurement, include the unit of measure. A value greater than 0 explicitly specifies the width of the

column.

Settings

To separate column entries, use semicolons (;) as list separators. Or use the list separator specified in the Regional Settings section of the Windows Control Panel.

Any or all of the **ColumnWidths** property settings can be blank. You create a blank setting by typing a list separator without a preceding value.

If you specify a -1 in the property page, the displayed value in the property page is a blank.

To calculate column widths when **ColumnWidths** is blank or -1, the width of the control is divided equally among all columns of the list. If the sum of the specified column widths exceeds the width of the control, the list is left-aligned within the control and one or more of the rightmost columns are not displayed. Users can scroll the list using the horizontal scroll bar to display the rightmost columns.

The minimum calculated column width is 72 [points](#) (1 inch). To produce columns narrower than this, you must specify the width explicitly.

Unless specified otherwise, column widths are measured in points. To specify another unit of measure, include the units as part of the values. The following examples specify column widths in several units of measure and describe how the various settings would fit in a three-column list box that is 4 inches wide.

Setting	Effect
90;72;90	The first column is 90 points (1.25 inch); the second column is 72 points (1 inch); the third column is 90 points.
6 cm;0;6 cm	The first column is 6 centimeters; the second column is hidden; the third column is 6 centimeters. Because part of the third column is visible, a horizontal scroll bar appears.
1.5 in;0;2.5 in	The first column is 1.5 inches, the second column is hidden, and the third column is 2.5 inches.

2 in;;2 in	The first column is 2 inches, the second column is 1 inch (default), and the third column is 2 inches. Because only half of the third column is visible, a horizontal scroll bar appears.
(Blank)	All three columns are the same width (1.33 inches).

Remarks

In a **ComboBox**, the system displays the column designated by the [TextColumn](#) property in the text box portion of the control. Setting **TextColumn** to -1 displays the first column that has a **ColumnWidths** value greater than 0.