# Visual Basic: DataCombo/DataList Controls

**Visual Studio 6.0**

# AboutBox Method

See Also Example Applies To

Displays the About box for the control.

**Syntax**

*object*.**AboutBox**

The *object* placeholder represents an object expression that evaluates to an object in the Applies To list.

**Remarks**

This is the same as clicking About in the Properties window.

# Visual Basic: Winsock Control

**Visual Studio 6.0**

# Accept Method

See Also Example Applies To

For TCP server applications only. This method is used to accept an incoming connection when handling a ConnectionRequest event.

**Syntax**

*object*.**Accept** *requestID*

The *object* placeholder represents an object expression that evaluates to an object in the Applies To list.

**Data Type**

Long

**Return Value**

Void

**Remarks**

The **Accept** method is used in the ConnectionRequest event. The ConnectionRequest event has a corresponding argument, the **RequestID** parameter, that should be passed to the **Accept** method. The **Accept** method should be used on a new control instance (other than the one that is in the listening state.) To do this, at design time, set the **Index** of a single Winsock control to 0. An example is shown below:

```
Private Sub Winsock1_ConnectionRequest _
(Index As Integer, ByVal requestID As Long)
   ' Close the connection if it is currently open
   ' by testing the State property.
   If Winsock1.State <> sckClosed Then Winsock1.Close

   ' Load a new instance of the control to service the connection.
   ' The variable newInstanceIndex maintains the current index of the
   ' next connection to load - this way we don't accidently use an
   ' index already created. Remember to Unload the control when the
   ' connection is closed
   Load Winsock1(newInstanceIndex)
   ' Pass the value of the requestID parameter to the
   ' Accept method.
   Winsock1.Accept requestID
End Sub
```

# Visual Basic: Winsock Control

# Accept Method, ConnectionRequest Event Example

The example shows the code necessary to connect a **Winsock** control using the TCP protocol. The code runs on the machine that is accepting the connection request. The **RequestID** parameter identifies the request. This is passed to the **Accept** method which accepts the particular request.

```
Private Sub WinsockTCP_ConnectionRequest _
(requestID As Long)
   If Winsock1.State <> sckClosed Then Winsock1.Close
   Winsock.Accept requestID
End Sub
```

# Visual Basic Extensibility Reference

**Visual Studio 6.0**

# Activate Method

See Also Example Applies To

Causes the currently selected component in the project window to be activated as if it were double-clicked.

**Syntax**

*object*.**Activate**

The *object* placeholder represents an object expression that evaluates to an object in the Applies To list.

# Visual Basic Reference

**Visual Studio 6.0**

# Add Method (BindingCollection)

See Also Example Applies To

Adds a **Binding** object to the **BindingCollection** object.

**Syntax**

*object*.**Add**(*object*, *PropertyName*, *DataField*, *DataFormat*, *Key*)

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

| Part | Description |
|---|---|
| *object* | An object expression that evaluates to an object in the Applies To list. |
| *object* | Required. The control or other data consumer which will be bound. |
| *PropertyName* | Required. The property of the data consumer to which the data field will be bound. |
| *DataField* | Required. The column of the data source that will be bound to the property specified in the *PropertyName* argument. |
| *DataFormat* | Optional. A **DataFormat** object or a reference to a **DataFormat** variable that will be used to format the bound property. |
| *Key* | Optional. A unique string that identifies the member of the collection. |

**Remarks**

The **Binding** object represents a property of an object bound to a data field of a data source. Use the **BindingCollection** object to bind a data source that has no design time interface, such as a **Class** configured as a data source, to a data consumer. You can also bind an OLE Simple Provider (OSP) data source to a data consumer using the **Binding** object.

You cannot use the **Binding** object to bind a complex bound control (such as the **DataGrid** control) to a data source. Instead, simply set the **DataSource** property of the control to the source.

# Visual Basic Reference

# Binding Object, BindingCollection Object Example

The example uses the **BindingCollection** object to bind a data source to two **TextBox** controls. The example first opens an ADODB recordset object, then sets the **DataSource** property of the **BindingCollection** to the recordset. The code then adds two **Binding** objects to the collection, thereby binding two **TextBox** controls to different fields of the recordset.

To try the example, in the **References** dialog box set a reference to the **Microsoft Data Binding Collection**. In the same dialog box, set a reference to the **Microsoft ActiveX Data Objects Library**. Draw two **TextBox** controls on a Form, and paste the code into the Declarations section. Press F5, and click the form to advance through the recordset.

```
Option Explicit
Private colBndNwind As New BindingCollection
Private rsNwind As New ADODB.Recordset
Private cn As New ADODB.Connection

Private Sub Form_Load()

   ' Set the Connection object parameters.
   With cn
      ' The following connection may or may not work on your computer.
      ' Alter it to find the Nwind.mdb file, which is included with
      ' Visual Basic.
      .Provider = "Microsoft.Jet.OLEDB.3.51"
      .Open "C:\Program Files\DevStudio\VB\Nwind.mdb"
   End With

   ' Open the recordset object.
   rsNwind.Open "Select * From Products", cn

   ' Set the DataSource of the Bindings collection to the recordset.
   Set colBndNwind.DataSource = rsNwind

   ' Add to the Bindings collection.
   With colBndNwind
      .Add Text1, "Text", "ProductName", , "product"
      .Add Text2, "Text", "SupplierID", , "ID"
   End With

   ' Print the properties of the objects in the collection.
   Dim bndObj As Binding
   For Each bndObj In colBndNwind
      Debug.Print "DataField", "PropertyName", "Key"
      Debug.Print bndObj.DataField, bndObj.PropertyName, bndObj.Key
      Debug.Print
   Next
End Sub

Private Sub Form_Click()
   ' Move to the next record by clicking the form.
```

```
    rsNwind.MoveNext
End Sub
```

# Visual Basic: Windows Controls

**Visual Studio 6.0**

# Add Method (ButtonMenus Collection)

See Also Example Applies To

Adds a **ButtonMenu** object to the **ButtonMenus** collection and returns a reference to the newly created object.

**Syntax**

*object*.**Add** (*index, key, text)*

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

| Part | Description |
|---|---|
| *object* | Required. An object expression that evaluates to an object in the Applies To list. |
| *index* | Optional. An integer specifying the position where you want to insert the object. If no index is specified, the object is added to the end of the collection. |
| *key* | Optional. A unique string that identifies the object. Use this value to retrieve a specific object in the collection. |
| *text* | Required. The string that will appear in the menu. |

**Remarks**

To see **ButtonMenu** objects, you must set the **Button** object's **Style** property to **tbrDropDown**.

To detect when the user has clicked a **ButtonMenu** object, use the ButtonMenuClick event of the **Toolbar** control.

# Visual Basic: Windows Controls

**Visual Studio 6.0**

# Add Method (Buttons Collection)

See Also Example Applies To

Adds a **Button** object to a **Buttons** collection and returns a reference to the newly created object.

**Syntax**

*object*.**Add(***index, key, caption, style, image***)**

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

| Part | Description |
|---|---|
| *object* | Required. An object expression that evaluates to a **Buttons** collection. |
| *index* | Optional. An integer specifying the position where you want to insert the **Button** object. If no *index* is specified, the **Button** is added to the end of the **Buttons** collection. |
| *key* | Optional. A unique string that identifies the **Button** object. Use this value to retrieve a specific **Button** object. |
| *caption* | Optional. A string that will appear beneath the **Button** object. |
| *style* | Optional. The style of the **Button** object. The available styles are detailed in the **Style** Property (**Button** Object). |
| *image* | Optional. An integer or unique key that specifies a **ListImage** object in an associated **ImageList** control. |

**Remarks**

You can add **Button** objects at design time using the Buttons tab of the Properties Page of the **Toolbar** control. At run time, use the **Add** method to add **Button** objects as in the following code:

```
Dim btnButton as Button
Set btnButton = Toolbar1.Buttons.Add(, "open", , tbrDefault, "open")
```

You associate an **ImageList** control with the **Toolbar** through the **Toolbar** control's **ImageList** property.

# Visual Basic: Windows Controls

**Visual Studio 6.0**

# Add Method (ColumnHeaders Collection)

See Also Example Applies To

Adds a **ColumnHeader** object to a **ColumnHeaders** collection in a **ListView** control.

**Syntax**

*object*.**Add**(*index, key, text, width, alignment, icon*)

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

| Part | Description |
|---|---|
| *object* | Required. An object expression that evaluates to a **ColumnHeaders** collection. |
| *index* | Optional. An integer that uniquely identifies a member of an object collection. |
| *key* | Optional. A unique string expression that can be used to access a member of the collection. |
| *text* | Optional. A string that appears in the **ColumnHeader** object. |
| *width* | Optional. A numeric expression specifying the width of the object using the scale units of the control's container. |
| *alignment* | Optional. An integer that determines the alignment of text in the **ColumnHeader** object. For settings, choose the **Alignment** property from the See Also list. |
| *icon* | Optional. The key or index of an image in the **Smallicons** ImageList. |

**Remarks**

The **Add** method returns a reference to the newly inserted **ColumnHeader** object.

Use the *index* argument to insert a column header in a specific position in the **ColumnHeaders** collection.

When the members of a **ColumnHeaders** collection can change dynamically, you may want to reference them using the **Key** property, because the **Index** property for any **ColumnHeader** object may be changing.

# Visual Basic: DataGrid Control

**Visual Studio 6.0**

# Add Method (Columns, SelBookmarks, Splits Collections)

See Also Example Applies To

Adds a new column to the **Columns** collection, a new bookmark to the **SelBookmarks** collection, or a new split to the **Splits** collection of the **DataGrid** control. Doesn't support named arguments.

**Syntax**

*object*.**Add** *colindex*

*object*.**Add** *bookmark*

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

| Part | Description |
|---|---|
| *object* | An object expression that evaluates to an object in the Applies To list. |
| *colindex* | Required. An integer that specifies where the new **Column** or **Split** object is inserted in the **Columns** collection or **Splits** collection, as described in Settings. |
| *bookmark* | The bookmark to be added to the collection. |

**Settings**

The settings for *colindex* are:

| Setting | Description |
|---|---|
| 0 | Inserts new column as leftmost column. |
| **Count** | If the *colindex* argument is the same as the **Count** property setting, the new column is inserted as the rightmost column. |
| n | Inserts the new column to the left of the nth column in the **Columns** collection. The *n*th column and all subsequent columns are incremented accordingly. |

### Remarks

The **Add** method inserts a new **Column** object into the **Columns** collection based on the *colindex* argument. New columns are added with their **Visible** property set to **False** and all other properties set to their default values. Initially, new columns are unbound because the **DataField** property is set to a zero-length string (""). The **Count** property of the **Columns** collection is incremented to reflect the new column.

**Important** If you have previously deleted a column using the **Remove** method, after adding new columns, you may need to refresh the display with the **Rebind** and **Refresh** methods. This instructs the **DataGrid** control to rebuild its internal column layout matrix to correctly reflect the true status of the control.

Use the **Add** method to add bookmarks to the **SelBookmarks** collection. Once a bookmark is appended to the **SelBookmarks** collection, it appears selected in the **DataGrid** control.

# Visual Basic: Windows Controls

**Visual Studio 6.0**

# Add Method (ComboItems Collection)

See Also Example Applies To

Adds a **ComboItem** object to the collection and returns a reference to the newly created object.

**Syntax**

*object*.**Add(***Index* **As Variant,** *Key* **As Variant,** *Text* **As Variant,** *Image* **As Variant,** *SelImage* **As Variant,** *Indentation* **As Variant) As** *ComboItem*

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

| Part | Description |
|---|---|
| *object* | An object expression that evaluates to an object in the Applies To list. |
| *Index* | Optional. The position within the collection to create the new object. |
| *Key* | Optional. A unique string which identifies the item within the collection. May be used in place of *Index* to designate the object. |
| *Text* | Optional. The text of the item, as it will appear in the list and text portions of the combo box. |
| *Image* | Optional. An index or key into an **ImageList** control that identifies the picture to use with the list item. |
| *SelImage* | Optional. An index or key into an **ImageList** control that identifies the picture to use with the list item when it is selected. |
| *Indentation* | Optional. The level of indentation that will be applied to the item. The amount of space applied to each level of indentation is determined by the **Indentation** property. |
| *ComboItem* | A reference to the new **ComboItem** object returned as a result of the successful completion of the function. |

**Remarks**

If no arguments are passed to the **Add** method, the new object is created as the last object in the collection. No default values are specified, and *Indentation* is set to 0.

The **Add** method returns a reference to the newly created **ComboItem** object. You can use this to assign the new **ComboItem** to an object variable and then access or change any of its properties.

# Visual Basic: Windows Controls

# Add Method Example

This example shows how to add a **ComboItem** using the **Add** method, and use the reference returned to change the properties of the new object.

```
Dim ci As ComboItem

Set ci = ImageCombo1.ComboItems. _
Add(1, "Signal1", "Signal", "RedLight", "GreenLight", 0)

ci.ToolTipText = "Traffic Light"
ci.Indentation = 2
```

# Visual Basic Reference

**Visual Studio 6.0**

# Add Method (Controls Collection)

See Also Example Applies To

Adds a control to the **Controls** collection and returns a reference to the control.

**Syntax**

*object*.**Add (***ProgID*, *name*, *container***)**

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

| Part | Description |
|---|---|
| *object* | Required. An object expression that evaluates to an object in the Applies To list. |
| *ProgID* | Required. A string that identifies the control. The *ProgID* of most controls can be determined by viewing the Object Browser. The *ProgID* is composed of the Library and Class of the control. For example, the **CommandButton** control's *ProgID* is VB.CommandButton. In cases where the *ProgID* differs from that shown in the Object Browser, Visual Basic displays an error message that contains the correct *ProgId*. |
| *name* | Required. A string that identifies the member of the collection. |
| *container* | Optional. An object reference that specifies a container of the control. If not specified or NULL, defaults to the container to which the Controls collection belongs. You can put a control in any existing container control (such as the **Frame** control) by specifying this argument. A user control or an ActiveX document can also be a container. |

**Remarks**

**Note** The **Controls** collection is a late-bound collection. This means the compiler cannot determine in advance which controls are contained by the collection, their objects or their interfaces. Without this information, the Auto Statement Builder will not function.

This method allows you to add controls to an application at run time. Dynamic control addition can be used to add the functionality of a control to an application, even after the application has been compiled and deployed. For example, you may have several complex user controls, each suited to a different task. Depending on an external factor, such as time or date or user input, a different user control could be added to an existing form in an application. You can also use the *container* argument of the method to specify a container control (such as the **Frame** control) to position the control. Or you can design an application that automatically reads a file, database, or registry entry for new controls to load. In this way, you can modify an application without having to redeploy it.

**Important** When you add an unreferenced control that requires a license to an existing (deployed) application, you must also add the license key for the control before using the **Add** method. For information on when and how to add licenses, see

"Licenses Collection" in the See Also list.

### Adding Unreferenced Controls at Run Time

You can also use the **Add** method to dynamically add a control that is not referenced in the project. (An "unreferenced" control is a control that is not present in the Toolbox.) To do so, you must also add the control's License key to the Licenses collection as well. The example below adds a control's license key before adding the control itself:

```
Option Explicit
Private WithEvents extCtl As VBControlExtender

Private Sub Form_Load()
   Licenses.Add "prjWeeks.WeeksCtl", "xydsfasfjewfe"
   Set extCtl = Form1.Controls.Add("prjWeeks.WeeksCtl", "ctl1")
   extCtl.Visible = True ' The control is invisible by default.
End Sub
```

**Note** See Add Method (Licenses Collection) in the See Also list for more information about retrieving a control's license key.

In order to program the events of such an unreferenced control, however, you must declare an object variable using the **WithEvents** keyword as a **VBControlExtender** object (shown above), and set the object variable to the reference returned by the **Add** method. Then use the **VBControlExtender** object's ObjectEvent event to program the control's events. An abbreviated example is shown below.

```
Option Explicit
Dim WithEvents objExt As VBControlExtender ' Declare VBControlExtender variable

Private Sub LoadControl()
   Licenses.Add "Project1.Control1", "xydsfasfjewfe"
   Set objExt = Controls.Add("Project1.Control1", "myCtl")
   objExt.Visible = True
End Sub

Private Sub extObj_ObjectEvent(Info As EventInfo)
   ' Program the events of the control using Select Case.
   Select Case Info.Name
   Case "Click"
      ' Handle Click event here.
   ' Other cases now shown
   Case Else ' Unknown Event
      ' Handle unknown events here.
   End Select
End Sub
```

**Note** You can't assign an intrinsic control to the VBControlExtender variable; any attempt will result in a type mismatch error.

You can also program the events of a dynamically added control by declaring an object variable using the **WithEvents** keyword, and setting the reference returned by the method to the variable, as shown below:

```
Option Explicit
' Declare object variable as CommandButton.
Private WithEvents cmdObject As CommandButton

Private Sub Form_Load()
   Set cmdObject = Form1.Controls.Add("VB.CommandButton", "cmdOne")
   cmdObject.Visible = True
   cmdObject.Caption = "Dynamic CommandButton"
End Sub
```

```
Private Sub cmdObject_Click()
    Print "This is a dynamically added control"
End Sub
```

If you intend to add a user control or any ActiveX control to your form, you must either add the control to the Toolbox, or add its License key to the Licenses collection. See the Add Method (Licenses Collection) for more information.

**Note** If you add an ActiveX or user control to your project but don't use it on a form, you must also uncheck the Remove Information About Unused ActiveX Controls option on the Make tab of the Project Properties dialog box. If your application attempts to add the control, the Add method will fail because the necessary information has been discarded.

### Removing Controls

To remove any controls added dynamically, use the **Remove** method. It should be noted that you can only remove controls added using the **Add** method (in contrast to controls added using the **Load** statement). The example below removes a dynamically added control:

```
Form1.Controls.Remove "ctl1" ' The control's name is ctl1.
```

# Visual Basic Reference

# Add Method (Controls Collection) Examples

```
Private Sub Form_Load()
   Form1.Controls.Add "VB.CommandButton", "cmdObj1", Frame1
   With Form1!cmdObj1
      .Visible = True
      .Width = 2000
      .Caption = "Dynamic Button"
   End With
End Sub
```

**Note** The code example above uses ! as a syntax element. You can also use standard collection syntax such as Form1.Controls("cmdObj1") to reference the control.

This second example declares an object variable of type CommandButton using the **WithEvents** keyword, allowing you to program the events of the control. The object variable is set to the reference returned by the **Add** method. To try the example, paste the code into the Declarations section and run the project.

```
Option Explicit
Private WithEvents btnObj As CommandButton

Private Sub btnObj_Click()
   MsgBox "This is a dynamically added button."
End Sub

Private Sub Form_Load()
   Set btnObj = Controls.Add("VB.CommandButton", "btnObj")
   With btnObj
      .Visible = True
      .Width = 2000
      .Caption = "Hello"
      .Top = 1000
      .Left = 1000
   End With
End Sub
```

The third example adds an unreferenced control to the **Controls** collection. To program such a control's events, however, you must declare an object variable of type **VBControlExtender**, and set the reference returned by the method to the object. Then program the control's events using the ObjectEvent event.

```
Option Explicit
Dim ctlExtender As VBControlExtender

Private Sub Form_Load()
   Set ctlExtender = Controls.Add("Project1.UserControl1", "MyControl")
   With ctlExtender
      .Visible = True
      .Top = 1000
      .Left = 1000
   End With
End Sub
```

```
Private Sub extObj_ObjectEvent(Info As EventInfo)
   ' Program the events of the control using Select Case.
   Select Case Info.Name
   Case "UserName"
      ' Check User name value.
      MsgBox Info.EventParameters("UserName").Value
   ' Other cases now shown
   Case Else ' Unknown Event
      ' Handle unknown events here.
   End Select
End Sub
```

# Visual Basic Reference

**Visual Studio 6.0**

# Add Method (DataMembers Collection)

See Also Example Applies To

Adds a data member to the **DataMembers** Collection.

**Syntax**

*object*.**Add** (*DataMember*)

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

| Part | Description |
|---|---|
| *object* | Required. An object expression that evaluates to an object in the Applies To list. |
| *DataMember* | Required. A unique string that can be used to identify a specific data member. |

**Remarks**

A data member can be either ADO recordset, an object that implements the OLE Simple Provider (OSP) interface using the **Implements** statement with the OSP class, or an OLEDB provider created with Visual Basic.

# Visual Basic Reference

# BindingCollection Object, DataMembers Collection Example

The example uses a class module as a data source. When code to set the **DataSource** and **DataMember** properties of two **Binding** objects executes, the class module's Initialize event occurs; two ADO recordsets are created in that event, and the names of the recordsets are added to the **DataMembers** collection. The GetDataMember event and its arguments are used to return data to the data consumer.

To try the example, on the **Project** menu, click **References**, and set a reference to **Microsoft Data Binding Collection** and **Microsoft ActiveX Data Objects**. On the Project menu, click **Add Class Module**. Change the name of the class to MyDataClass, and set the **DataSourceBehavior** property to **vbDataSource**. Then draw two **TextBox** controls on a form. Paste the code into the **Form** object's code module.

```
Option Explicit
' Declare the object variables, one for a Class module named MyDataClass,
' and two more for each BindingCollection object one for each
' recordset).

Private clsData As New MyDataClass               ' Class module
Private bndColProducts As New BindingCollection  ' Bindings Collection
Private bndColSuppliers As New BindingCollection ' Bindings Collection

Private Sub Form_Load()
   ' Set DataSource and DataMember properties for each Bindings
   ' collection object.
   With bndColProducts
      .DataMember = "Products"
      Set .DataSource = clsData
      .Add Text1, "Text", "ProductName" ' Bind to a TextBox.
   End With

   With bndColSuppliers
      .DataMember = "Suppliers"
      Set .DataSource = clsData
      .Add Text2, "Text", "CompanyName" ' Bind to a TextBox.
   End With

   ' Change the Caption of Command1
   Command1.Caption = "MoveNext"

End Sub

Private Sub Command1_Click()
   clsData.MoveNext
End Sub
```

Paste the code below into the MyDataClass module. The **DataSourceBehavior** property must be set to **vbDataSource** in order to see the GetDataMember event. Run the project.

```
Option Explicit
' Declare object variables for ADO Recordset and Connection objects.
```

```
Private WithEvents rsProducts As ADODB.Recordset
Private WithEvents rsSuppliers As ADODB.Recordset
Private cnNwind As ADODB.Connection

Private Sub Class_Initialize()
   ' Add strings to the DataMembers collection.
   With DataMembers
      .Add "Products"
      .Add "Suppliers"
   End With

   ' Set Recordset objects.
   Set rsProducts = New ADODB.Recordset
   Set rsSuppliers = New ADODB.Recordset
   Set cnNwind = New ADODB.Connection

   ' Set the Connection object parameters.
   With cnNwind
      ' The Nwind.mdb that comes with Visual Basic must be installed on
      ' the computer or the code will fail. Otherwise alter the path to
      ' find the file on the computer.
      .Provider = "Microsoft.Jet.OLEDB.3.51"
      .Open "C:\Program Files\DevStudio\VB\Nwind.mdb"
   End With

   ' Open the recordset objects.
   rsSuppliers.Open "SELECT * FROM Suppliers", cnNwind, _
   adOpenStatic, adLockOptimistic
   rsProducts.Open "SELECT * FROM Products", cnNwind, _
   adOpenStatic, adLockOptimistic

End Sub

' The GetDataMember occurs when the DataSource property of a data
' consumer is set. In this case, the Bindings collection object is
' the consumer.
Private Sub Class_GetDataMember(DataMember As String, Data As Object)
   Select Case DataMember
   Case "Products"
      Set Data = rsProducts
   Case "Suppliers"
      Set Data = rsSuppliers
   Case ""
      ' Provide a default record source when no Data Member is specified.
      Set Data = rsProducts
   End Select

End Sub

Public Function MoveNext()
   If rsProducts.EOF Then
      rsProducts.MoveFirst
   Else
      rsProducts.MoveNext
   End If
End Function

Private Sub rsProducts_MoveComplete(ByVal adReason As _
ADODB.EventReasonEnum, ByVal pError As ADODB.Error, adStatus As _
ADODB.EventStatusEnum, ByVal pRecordset As ADODB.Recordset)
   ' Keep the two recordsets in sync. The first textbox displays
```

```
    ' the supplier of the product. If the SupplierID for both
    ' recordsets are equivalent, no change needed. Otherwise,
    ' move to first record and test for SupplierID. This example
    ' is for demonstration only as the method is not the most
    ' efficient.

    If rsSuppliers("SupplierID").Value = _
    pRecordset("SupplierID").Value Then Exit Sub

    rsSuppliers.MoveFirst
    Do While Not rsSuppliers.EOF
       If rsSuppliers("SupplierID").Value = _
       pRecordset("SupplierID").Value Then
          Exit Sub
       Else
          rsSuppliers.MoveNext
       End If
    Loop
End Sub
```

# Visual Basic Reference

**Visual Studio 6.0**

# Add Method (DataObjectFiles Collection)

See Also Example Applies To

Adds a filename to the **Files** collection of a **DataObject** object.

**Syntax**

*object*.**Add (***filename*, [*index*]**)**

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

| Part | Description |
|---|---|
| *object* | An object expression that evaluates to an object in the Applies To list. |
| *filename* | Required. A string that sets the name of the file. |
| *index* | Optional. An integer specifying where in the collection you want to insert the filename. If you don't specify an index value, the filename is added to the end of the collection. |

**Remarks**

The **Files** collection can be filled only with filenames that are of type **vbCFFiles** (as found in the **ClipBoardConstants** list in the Object Browser). The **DataObject** object itself, however, can contain several different types of data. To retrieve a list of file names, iterate through the **Files** collection.

# Visual Basic Reference

**Visual Studio 6.0**

# Add Method (DEDesigner Extensibility)

See Also Example Applies To

Adds a member to a **DECommands** or **DEConnections** collection.

**Syntax**

*object*.**Add(***string, Boolean***)**

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

| Part | Description |
|---|---|
| *object* | An object expression that evaluates to an item in the Applies To list. |
| *string* | This specifies a unique name for the **DEConnection** or **DECommand** object.<br>**Note** If the name already exists in the collection, an error occurs and a **DEConnection** or **DECommand** object is not added. |
| *Boolean* | Optional. A Boolean expression that specifies whether the **Command** or **Connection Properties** dialog box displays after the object is added to the Data Environment. |

**Remarks**

For the **DEAggregates**, **DEGroupingFields**, and **DERelationConditions** collections, the *Boolean* is not used. Thus, for these three collections, the **Add** method is the same as Visual Basic's Add method.

# Visual Basic for Applications Reference

**Visual Studio 6.0**

# Add Method (Dictionary)

See Also Example Applies To Specifics

**Description**

Adds a key and item pair to a **Dictionary** object.

**Syntax**

*object*.**Add** *key*, *item*

The **Add** method has the following parts:

| Part | Description |
|---|---|
| *object* | Required. Always the name of a **Dictionary** object. |
| *key* | Required. The key associated with the item being added. |
| *item* | Required. The item associated with the key being added. |

**Remarks**

An error occurs if the *key* already exists.

# Visual Basic Reference

**Visual Studio 6.0**

# Add Method (ExportFormats Collection)

See Also Example Applies To

Adds an **ExportFormat** object to the **ExportFormats** collection and returns a reference to the newly created object.

**Syntax**

*object*.**Add (***Key*, *FormatType*, *FileFormatString*, *FileFilter*, *Template***)**

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

| Part | Description |
|---|---|
| *object* | Required. An object expression that evaluates to an object in the Applies To list. |
| *Key* | Required. A unique string that identifies the member of the collection. |
| *FormatType* | Required. Sets the report type of the object, as shown in Settings. |
| *FileFormatString* | Required. Sets the text displayed in the Save As combo box of the Export dialog box. |
| *FileFilter* | Required. Sets the file extension to be used if the user selects the ExportFormat object in Export dialog box. If multiple file filters are used, the first becomes the default extension. |
| *Template* | Optional. Sets the template to be used for the report. |

**Settings**

The settings for *FormatType* are:

| Constant | Value | Description |
|---|---|---|
| **rptFmtHTML** | 0 | HTML |
| **rptFmtText** | 1 | Text |
| **rptFmtUnicodeText** | 2 | Unicode |
| **rptFmtUnicodeHTML_UTF8** | 3 | HTML encoded using Universal Character Set (UTF 8) |

**Return Type**

**ExportFormat** object

**Remarks**

If you do not specify a **Template**, Visual Basic supplies a default template appropriate to the **FormatType**.

**Return Type**

**ExportFormat** object

**Remarks**

If you do not specify a **Template**, Visual Basic supplies a default template appropriate to the **FormatType**.

# Visual Basic Reference

**Visual Studio 6.0**

# Add Method (Files Collection)

See Also Example Applies To

Adds a filename to the **Files** collection of a **DataObject** object.

**Syntax**

*object*.**Add** (*filename*, *index*)

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

| Part | Description |
|---|---|
| *object* | An object expression that evaluates to an object in the Applies To list. |
| *filename* | Required. A string that sets the name of the file. |
| *index* | Optional. An integer specifying the position where you want to insert the member. If no index is specified, the member is added to the end of the collection. |

**Remarks**

The **Files** collection is filled with filenames only when the **DataObject** object contains data of type **vbCFFiles**. (The **DataObject** object can contain several different types of data.) You can iterate through the collection to retrieve the list of file names.

# Visual Basic Reference

**Visual Studio 6.0**

# Add Method (Licenses Collection)

See Also Example Applies To

Adds a license to the **Licenses** collection and returns the license key.

**Syntax**

*object*.**Add** (*ProgID*, *LicenseKey*)

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

| Part | Description |
|---|---|
| *object* | Required. An object expression that evaluates to an object in the Applies To list. |
| *ProgID* | Required. A string that specifies the control for which a license key will be added. |
| *LicenseKey* | Optional. A string that specifies the license key. |

**Remarks**

Use the **Add** method whenever you want to dynamically add a control that requires a license key. For more information about controls that require license keys, see "Licensing Issues for Controls" in the See Also list.

When you compile a user control that requires a license key, and you want to dynamically add that control to an existing application, you must use the **Add** method for the **Licenses** collection in two different ways.

First, use the method to return the license key that is hard-coded into a user control. Second, use the method to add the same license key to the **Licenses** collection before adding the user control to the **Controls** collection.

In most cases, you will have to use the method in both ways in order to properly deploy a compiled user control. The steps for this are outlined below.

After you have compiled a user control that requires a license key, use the **Add** method to return the license key. Store that license key where it can be retrieved by the deployed application. For example, the example below writes the key to a file. You could also store it in a database, or in the Windows registry.

```
Private Sub GenerateLicenseKey()
   Dim intFile As Integer
   intFile = FreeFile
   ' Open a file to write the license key to.
   Open "c:\Temp\Ctl_Licenses.txt" For Output As #intFile
   Dim strLicense As String
```

```
   strLicense = Licenses.Add("prjWeeks.WeeksCtl")
   ' Write the license key to the file.
   Write #intFile, strLicense
   Close #intFile
End Sub
```

When you deploy your control, the deployed application adds the license key to the **Licenses** collection before adding the control to the **Controls** collection. (Of course, the control must have been installed on the machine as well.) The code example below adds the license key, then adds the control:

```
Option Explicit
Dim WithEvents extObj As VBControlExtender

Private Sub LoadDynamicControl()
   Dim intFile As Integer
   intFile = FreeFile
   Open "c:\Download\Ctl_Licenses.txt" For Input As #intFile
   Dim strKey As String
   ' On the client machine, read the license key from the file.
   Input #intFile, strKey
   Licenses.Add "prjWeeks.WeeksCtl", strKey
   Close #intFile
   Set extObj = Controls.Add("prjWeeks.WeeksCtl", "ctl1")
   With Controls("ctl1")
      .Visible = True
   End With
End Sub
```

## When To Add a License Key

When you create a user control and you want to distribute the control for dynamic control addition, you must consider the following question: Does the user control contain only intrinsic controls? If the answer is yes, then ask, "Do I want to require that the end user have a license key in order to use the control?" If the answer to both questions is "yes," then be sure to check the Require License Key option on the General tab of the Project Properties dialog box.

**Note** Even if you clear the Require License Key option, a user control that contains third-party controls will still require a license key.

## When No License Key Is Needed

There are two circumstances when you do not have to add a license key in order to add a control to the **Controls** collection:

1. When the control is an intrinsic control and you have not checked the Require License Key option.
2. When you add a control that is referenced in the project. In other words, if the control is present in the Toolbox.

**Note** When you do have a control in the Toolbox and you plan to add that control only at run time, be sure to clear the Remove Information About Unused ActiveX Controls option on the Make tab of the Project Properties dialog box; otherwise, trying to add the control will fail.

**Visual Studio 6.0**

*Visual Basic: MSChart Control*

# Add Method (LightSources Collection)

See Also Example Applies To

Adds a **LightSource** object to the **LightSources** collection.

**Syntax**

*object*.**Add** (*x,y,z,intensity)*

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

| Part | Description |
|---|---|
| *collection* | A object expression that evaluates to an object in the Applies To list. |
| *x, y, z* | Integers. Indicate the light source location. |
| *intensity* | Single. Indicates the light source intensity. |

**Remarks**

Setting *x*, *y*, and *z* to zero generates a **VtChInvalidArgument** error.

# Visual Basic: Windows Controls

**Visual Studio 6.0**

# Add Method (ListImages Collection)

See Also Example Applies To

Adds a **ListImage** object to a **ListImages** collection.

**Syntax**

*object*.**Add(***index*, *key*, *picture***)**

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

| Part | Description |
|---|---|
| *object* | Required. An object expression that evaluates to an object in the Applies To list. |
| *index* | Optional. An integer specifying the position where you want to insert the **ListImage**. If no *index* is specified, the **ListImage** is added to the end of the **ListImages** collection. |
| *key* | Optional. A unique string that identifies the **ListImage** object. Use this value to retrieve a specific **ListImage** object. An error occurs if the key is not unique. |
| *picture* | Required. Specifies the picture to be added to the collection. |

**Remarks**

The **ListImages** collection is a 1-based collection.

You can load either bitmaps or icons into a **ListImage** object. To load a bitmap or icon, you can use the **LoadPicture** function, as follows:

```
Set imgX = ImageList1.ListImages.Add(,,LoadPicture("file name"))
```

You can also load a **Picture** object directly into the **ListImage** object. For example, this example loads a **PictureBox** control's picture into the **ListImage** object:

```
Set imgX = ImageList1.ListImages.Add(,,Picture1.Picture)
```

If no **ListImage** objects have been added to a **ListImages** collection, you can set the **ImageHeight** and **ImageWidth** properties before adding the first **ListImage** object. You can then add images of any size to the collection. However, when the **ImageList** control is bound to another Windows Common Control, all images you add to the collection will be displayed (in the bound Windows Common Control) at the size specified by the **ImageHeight** and **ImageWidth** properties. Once a

**ListImage** object has been added to the collection, the **ImageHeight** and **ImageWidth** properties become read-only properties.

**Note** You can the use the **ImageList** control with any control by setting the **Picture** property of the second control to the **Picture** object of any image contained by the **ImageList** control. However, the size of the displayed image will not be affected by the **ImageHeight** and **ImageWidth** properties. In other words, the second control will display the image at its original size.

You should use the **Key** property to reference a **ListImage** object if you expect the value of the **Index** property to change. For example, if you allow users to add and delete their own images to the collection, the value of the **Index** property may change.

**Note** When using the ImageList control on a DHTML Page designer, images cannot be added at design time. If you try to use the Add method in an uncompiled .dll project, you will get the run-time error: -2147418113 (8000ffff), "Method 'Add' of object images failed". However, the code will work when the .dll project is compiled.

# Visual Basic: Windows Controls

**Visual Studio 6.0**

# Add Method (ListItems Collection)

See Also Example Applies To

Adds a **ListItem** object to a **ListItems** collection in a **ListView** control and returns a reference to the newly created object.

**Syntax**

*object*.**Add**(*index, key, text, icon, smallIcon*)

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

| Part | Description |
|---|---|
| *object* | Required. An object expression that evaluates to a **ListItems** collection. |
| *index* | Optional. An integer specifying the position where you want to insert the **ListItem**. If no index is specified, the **ListItem** is added to the end of the **ListItems** collection. |
| *key* | Optional. A unique string expression that can be used to access a member of the collection. |
| *text* | Optional. A string that is associated with the **ListItem** object control. |
| *icon* | Optional. An integer that sets the icon to be displayed from an **ImageList** control, when the **ListView** control is set to Icon view. |
| *smallIcon* | Optional. An integer that sets the icon to be displayed from an **ImageList** control, when the **ListView** control is set to SmallIcon view. |

**Remarks**

Before setting either the **Icons** or **SmallIcons** properties, you must first initialize them. You can do this at design time by specifying an ImageList object with the General tab of the **ListView** Control Properties dialog box, or at run time with the following code:

```
ListView1.Icons = ImageList1 'Assuming the Imagelist is ImageList1.
ListView1.SmallIcons = ImageList2
```

If the list is not currently sorted, a **ListItem** object can be inserted in any position by using the *index* argument. If the list is sorted, the *index* argument is ignored and the **ListItem** object is inserted in the appropriate position based upon the sort order.

If *index* is not supplied, the **ListItem** object is added with an index that is equal to the number of **ListItem** objects in the collection + 1.

Use the **Key** property to reference a member of the **ListItems** collection if you expect the value of an object's **Index** property to change, such as by dynamically adding objects to or removing objects from the collection.

# Visual Basic: Windows Controls

**Visual Studio 6.0**

# Add Method (ListSubItems Collection)

See Also Example Applies To

Adds a **ListSubItem** object to the ListSubItems collection and returns a reference to the newly created object.

**Syntax**

*object*.**Add (***index, key, text, ReportIcon, ToolTipText)*

| Part | Description |
|---|---|
| *object* | Required. An object expression that evaluates to an object in the Applies To list. |
| *index* | Optional. An integer specifying the position where you want to insert the object. If no index is specified, the object is added to the end of the collection. |
| *key* | Optional. A unique string that identifies the object. Use this value to retrieve a specific object in the collection. |
| *text* | Optional. The string displayed by the **ListView** control in Report view. |
| *ReportIcon* | Optional. The **Index** or **Key** property value specifying a **ListImage** object in the associated **ImageList** control. |
| *ToolTipText* | Optional. The string displayed when the mouse pointer hovers over the **ListSubItem**. |

**Remarks**

The **ListSubItems** collection replaces the **SubItems** array of strings. The **SubItems** array is still available in the **ListView** control, but it's recommended that new applications use the **ListSubItems** collection because of the increased flexibility.

# Visual Basic: Windows Controls

**Visual Studio 6.0**

# Add Method (Nodes Collection)

See Also Example Applies To

Adds a **Node** object to a **Treeview** control's **Nodes** collection.

**Syntax**

*object*.**Add**(*relative, relationship, key, text, image, selectedimage*)

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

| Part | Description |
|---|---|
| *Object* | Required. An object expression that evaluates to an object in the Applies To list. |
| *Relative* | Optional. The index number or key of a pre-existing **Node** object. The relationship between the new node and this pre-existing node is found in the next argument, *relationship*. |
| *Relationship* | Optional. Specifies the relative placement of the **Node** object, as described in Settings. |
| *Key* | Optional. A unique string that can be used to retrieve the **Node** with the **Item** method. |
| *Text* | Optiona. The string that appears in the **Node**. |
| *Image* | Optional. The index of an image in an associated **ImageList** control. |
| *Selectedimage* | Optional. The index of an image in an associated **ImageList** control that is shown when the **Node** is selected. |

**Settings**

The settings for *relationship* are:

| Constant | Value | Description |
|---|---|---|
| **TvwFirst** | 0 | First. The **Node** is placed before all other nodes at the same level of the node named in *relative*. |
| **TvwLast** | 1 | Last. The **Node** is placed after all other nodes at the same level of the node named in *relative*. Any Node added subsequently may be placed after one added as Last. |
| | | |

| | | |
|---|---|---|
| **TvwNext** | 2 | (Default) Next. The **Node** is placed after the node named in *relative*. |
| **TvwPrevious** | 3 | Previous. The **Node** is placed before the node named in *relative*. |
| **TvwChild** | 4 | Child. The **Node** becomes a child node of the node named in *relative*. |

**Note** If no **Node** object is named in *relative*, the new node is placed in the last position of the top node hierarchy.

**Remarks**

The **Nodes** collection is a 1-based collection.

As a **Node** object is added it is assigned an index number, which is stored in the **Node** object's **Index** property. This value of the newest member is the value of the **Node** collection's **Count** property.

Because the **Add** method returns a reference to the newly created **Node** object, it is most convenient to set properties of the new **Node** using this reference. The following example adds several **Node** objects with identical properties:

```
Private Sub CreateNodes()
  ' Set CheckBoxes property to True to see checked nodes:
  TreeView1.CheckBoxes = True
  Dim nodX As Node   ' Declare the object variable.
  Dim i as Integer   ' Declare a counter variable.
  For i = 1 to 4
    Set nodX = TreeView1.Nodes.Add(,,,"Node " & Cstr(i))
    ' Use the reference to set other properties, such as Bold.
    NodX.Bold = True
    ' Set image property to image 3 in an associated ImageList.
    nodX.Checked = True
  Next i
End Sub
```

# Visual Basic: Windows Controls

# Add Method Example (Nodes Collection)

The following example adds two **Node** objects to a **TreeView** control. To try the example, place a **TreeView** control on a form, and paste the code into the form's Declarations section. Run the example, and click the **Node** object to expand it.

```
Private Sub Form_Load()
   ' Set Treeview control properties.
   TreeView1.LineStyle = tvwRootLines  ' Linestyle 1

   ' Add Node objects.
   Dim nodX As Node  ' Declare Node variable.
   ' First node with 'Root' as text.
   Set nodX = TreeView1.Nodes.Add(, , "r", "Root")

   ' This next node is a child of Node 1 ("r").
   Set nodX = TreeView1.Nodes.Add("r", tvwChild, "child1", "Child")
End Sub
```

# Visual Basic: RichTextBox Control

**Visual Studio 6.0**

# Add Method (OLEObjects Collection)

See Also Example Applies To

Adds an **OLEObject** object to an **OLEObjects** collection.

**Syntax**

*object*.**Add** *index, key, sourcedoc, class*

The **Add** Method syntax has these parts:

| Part | Description |
|---|---|
| *object* | Required. An object expression that evaluates to an object in the Applies To list. |
| *index* | Optional. An integer that identifies a member of the object collection. If supplied, the new member will be inserted after the member specified by the index. |
| *key* | Optional. A unique string expression that can be used to access a member of the collection. The *key* and *index* arguments can be used interchangeably with the **Item** method of the collection to retrieve the **OLEObject** object. |
| *sourcedoc* | Required. The filename of a document used as a template for the embedded object. The **RichTextBox** control doesn't support linking, so the contents of the file will be copied into the **OLEObject** object. Must be a zero-length string ("") if you don't specify a source document. |
| *class* | Optional. The OLE class name for the object to be embedded. This argument is the ProgID used by OLE in the system registry. This argument is ignored if you specify a filename for *sourcedoc*. |

**Remarks**

The following code adds a Microsoft Excel worksheet to the **RichTextBox** and sets its Key property to "SalesData":

```
RichTextBox1.OLEObjects.Add , "SalesData", , "Excel.Sheet.5"
```

When an object is added to the collection, it immediately becomes in-place active so the user can add data to it.

# Visual Basic: Windows Controls

**Visual Studio 6.0**

# Add Method (Panels Collection)

See Also Example Applies To

Adds a **Panel** object to a **Panels** collection and returns a reference to the newly created **Panel** object.

**Syntax**

*object*.**Add***(index, key, text, style, picture)*

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

| Part | Description |
|---|---|
| *object* | An object expression that evaluates to a **Panels** collection. |
| *index* | Optional. An integer specifying the position where the **Panel** object is to be inserted. If no *index* is specified, the **Panel** is added to the end of the **Panels** collection. |
| *key* | Optional. A unique string that identifies the **Panel**. Use *key* to retrieve a specific **Panel**. This is equivalent to setting the **Key** property of the new **Panel** object after the object has been added. |
| *text* | Optional. A string that appears in the **Panel**. This is equivalent to setting the **Text** property of the new **Panel** object after the object has been added. |
| *style* | Optional. The style of the panel. The available styles are detailed in the **Style** Property (**Panel** Object). This is equivalent to setting the **Style** property of the new **Panel** object after the object has been added. |
| *picture* | Optional. Specifies the bitmap displayed in the active **Panel**. For more information, see the **LoadPicture** function. This is equivalent to setting the **Picture** property of the new **Panel** object after the object has been added. |

**Remarks**

At run time, the **Add** method returns a reference to the newly inserted **Panel** object. With this reference, you can set properties for every new **Panel** in the following manner:

```
Dim pnlX As Panel
Dim i As Integer
For i = 1 To 6  ' Add six Panel objects.
  ' Create a panel and get a reference to it simultaneously.
  Set pnlX = StatusBar1.Panels.Add(, "Panel" & i) ' Set Key property.
  pnlX.Style = i  ' Set Style property.
```

```
  pnlX.AutoSize = sbrContents ' Set AutoSize property.
Next i
```

The value of the **Text** property is displayed in a **Panel** object when the **Panel** object's **Style** property is set to **sbrText**.

The **Panels** collection is a 1-based collection. In order to get a reference to the first (default) **Panel** in a collection, you can use its **Index** or **Key** (if there is one) properties, or the **Item** method. The following code references the first **Panel** object using its index.

```
Dim pnlX As Panel
' Get a reference to first Panel.
Set pnlX = StatusBar1.Panels(1) ' Use the index
pnlX.Text = "Changed text"    ' Alter the Panel object's text.
```

By default, one **Panel** already exists on the control. Therefore, after adding panels to a collection, the **Count** will be one more than the number of panels added. For example:

```
Dim i as Integer
For i = 1 to 4  ' Add four panels.
   StatusBar1.Panels.Add ' Add panels without any properties.
Next i
MsgBox StatusBar1.Panels.Count ' Returns 5 panels.
```

# Visual Basic: RDO Data Control

**Visual Studio 6.0**

# Add Method (Remote Data)

See Also Example Applies To

Adds a member to a Remote Data **Collection** object.

**Syntax**

*object*.**Add** *item, key, before, after*

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

| Part | Description |
|---|---|
| *object* | An object expression that evaluates to the **rdoResultset** object containing the **rdoColumns** collection. |
| *item* | Required. An expression of any type that specifies the member to add to the collection. |
| *key* | Optional. A unique string expression that specifies a key string that can be used, instead of a positional index, to access a member of the collection. |
| *before* | Optional. An expression that specifies a relative position in the collection. The member to be added is placed in the collection before the member identified by the before argument. If a numeric expression, before must be a number from 1 to the value of the collection's Count property. If a string expression, before must correspond to the key specified when the member being referred to was added to the collection. You can specify a before position or an after position, but not both. |
| *after* | Optional. An expression that specifies a relative position in the collection. The member to be added is placed in the collection after the member identified by the after argument. If numeric, after must be a number from 1 to the value of the collection's Count property. If a string, after must correspond to the key specified when the member referred to was added to the collection. You can specify a before position or an after position, but not both. |

**Remarks**

Whether the before or after argument is a string expression or numeric expression, it must refer to an existing member of the collection, or an error occurs.

An error also occurs if a specified key duplicates the key for an existing member of the collection.

# Visual Basic: DataRepeater Control

**Visual Studio 6.0**

# Add Method (RepeaterBindings Collection)

See Also Example Applies To

Adds a **RepeaterBinding** object to the **RepeaterBindings** collection and returns a reference to the new object.

**Syntax**

*object*.**Add(***PropertyName*, *DataField*, *DataFormat*, *key***)**

| Part | Description |
|---|---|
| *object* | An object expression that evaluates to an object in the Applies To list. |
| *PropertyName* | Required. A string that sets the data-bound property of the user control. |
| *DataField* | Required. A string that sets the data source field to bind to the property specified in *PropertyName*. |
| *DataFormat* | Optional. Sets the **DataFormat** object to be used. |
| *key* | Optional. A unique string that identifies the object. Use this value to retrieve a specific member of the collection. |

# Visual Basic: DataRepeater Control

# RepeaterBinding Object Example

The example below first prints the existing property names of the **RepeaterBindings** collection, then adds a **DataBinding** object to the collection, and finally changes the format of a **DataBinding** object.

```
Private Sub DataRepeater1_RepeatedControlLoaded()
   Dim rb As RepeaterBinding

   For Each rb In DataRepeater1.RepeaterBindings
      Debug.Print rb.PropertyName ' Print all property names.
   Next

   With DataRepeater1
      ' Add a new RepeaterBinding object.
      .RepeaterBindings.Add "cleared", "cleared"
      ' Change the Format to all caps.
      .RepeaterBindings(2).DataFormat.Format = ">"
   End With
End Sub
```

# Visual Basic: Windows Controls

**Visual Studio 6.0**

# Add Method (Tabs Collection)

See Also Example Applies To

Adds a **Tab** object to a **Tabs** collection in a **TabStrip** control.

**Syntax**

*object*.**Add(***index, key, caption, image***)**

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

| Part | Description |
|---|---|
| *object* | An object expression that evaluates to a **Tabs** collection. |
| *index* | Optional. An integer specifying the position where you want to insert the **Tab**. If you don't specify an index, the **Tab** is added to the end of the **Tabs** collection. |
| *key* | Optional. A unique string that identifies the **Tab**. Use key to retrieve a specific Tab. This is equivalent to setting the **Key** property of the new **Tab** object after the object has been added to the **Tabs** collection. |
| *caption* | Optional. The string that appears on the **Tab**. This is equivalent to setting the **Caption** property of the new **Tab** object after the object has been added to the **Tabs** collection. |
| *image* | Optional. The index of an image in an associated **ImageList** control. This image is displayed on the tab. This is equivalent to setting the **Image** property of the new **Tab** object after the object has been added to the **Tabs** collection. |

**Remarks**

To add tabs to the **TabStrip** control at design time, click the Insert Tab button on the Tab tab in the Properties Page of the **TabStrip** control, and then fill in the appropriate fields for the new tab.

To add tabs to the **TabStrip** control at run time, use the **Add** method, which returns a reference to the newly inserted **Tab** object. For example, the following code adds a tab with the *caption*, "Howdy!" whose *key* is "MyTab," as the second tab (its *index* is 2):

```
Set X = TabStrip1.Tabs.Add(2,"MyTab","Howdy!")
```

# Visual Basic: Windows Controls

# Add Method (Tabs Collection) Example

This example adds three **Tab** objects, each with captions and images from an **ImageList** control, to a **TabStrip** control. To try this example, put an **ImageList** and a **TabStrip** control on a form. The **ImageList** control supplies the images for the **Tab** objects, so add three images to the **ImageList** control. Paste the following code into the Load event of the Form object, and run the program.

```
Private Sub Form_Load()
   Dim X As Integer
   Set TabStrip1.ImageList = ImageList1
   TabStrip1.Tabs(1).Caption = "Time"
   TabStrip1.Tabs.Add 2, , "Date"
   TabStrip1.Tabs.Add 3, , "Mail"
   For X = 1 To TabStrip1.Tabs.Count
      TabStrip1.Tabs(X).Image = X
   Next X
End Sub
```

# Visual Basic Extensibility Reference

**Visual Studio 6.0**

# Add Method (VBA Add-In Object Model)

See Also Example Applies To Specifics

Adds an object to a collection.

**Syntax**

*object*.**Add(***component***)**

The **Add** syntax has these parts:

| Part | Description |
|---|---|
| *object* | Required. An object expression that evaluates to an object in the Applies To list. |
| *component* | Required. For the **LinkedWindows** collection, an object. For the **VBComponents** collection, an enumerated constant representing a class module, a form, or a standard module.<br>For the **VBProjects** collection, an enumerated constant representing a project type. |

You can use one of the following constants for the *component* argument:

| Constant | Description |
|---|---|
| **vbext_ct_ClassModule** | Adds a class module to the collection. |
| **vbext_ct_MSForm** | Adds a form to the collection. |
| **vbext_ct_StdModule** | Adds a standard module to the collection. |
| **vbext_pt_StandAlone** | Adds a standalone project to the collection. |

**Remarks**

For the **LinkedWindows** collection, the **Add** method adds a window to the collection of currently linked windows.

**Note** You can add a window that is a pane in one linked window frame to another linked window frame; the window is simply moved from one pane to the other. If the linked window frame that the window was moved from no longer contains any panes, it's destroyed.

For the **VBComponents** collection, the **Add** method creates a new standard component and adds it to the project.

For the **VBComponents** collection, the **Add** method returns a **VBComponent** object. For the **LinkedWindows** collection, the **Add** method returns **Nothing**.

For the **VBProjects** collection, the **Add** method returns a **VBProject** object and adds a project to the **VBProjects** collection.

# Visual Basic Extensibility Reference

**Visual Studio 6.0**

# Add Method (Visual Basic Extensibility)

See Also Example Applies To

- ContainedVBControls collection: Adds a new **VBControl** object to the ContainedVBControls collection.
- VBControls collection: Adds a new **VBControl** object to the VBControls collection.
- VBProjects collection: adds a new, empty project to the set of projects in the VBProjects collection.

**Syntax**

*object*.**Add** (*progid* **As String**, **[***relativevbcontrol* **As VBControl] [***before* **As Boolean]**) **As VBControl**

*object*.**Add** (*projecttype* **As vbext_ProjectType**, **[***exclusive* **As Boolean]**) **As VBProject**

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

| Part | Description |
|---|---|
| *object* | An object expression that evaluates to an object in the Applies To list. |
| *progid* | Required. A string expression specifying the ProgID of the component to be added. |
| *relativevbcontrol* | Optional. An existing **VBControl** object specifying the point where the new component is to be inserted. |
| *before* | Optional. Default = **False**. A Boolean expression specifying whether the new **VBControl** is to be placed before or after the *relativevbcontrol*. |
| *projecttype* | Required. A **VBProject** object specifying the type of the new project. For a list of kinds of projects, see the **Kind** property. |
| *exclusive* | Optional. Default = **False**. A Boolean expression specifying whether a new project is added to an existing set of projects, or added as the only project. |

**Remarks**

If the *exclusive* parameter is specified as **True**, then the existing group project is closed and the new project becomes the only project in the collection.

# Visual Basic for Applications Reference

**Visual Studio 6.0**

# Add Method

See Also Example Applies To Specifics

Adds a member to a **Collection** object.

**Syntax**

*object*.**Add** ***item, key, before, after***

The **Add** method syntax has the following object qualifier and named arguments:

| Part | Description |
|---|---|
| *object* | Required. An object expression that evaluates to an object in the Applies To list. |
| ***item*** | Required. An expression of any type that specifies the member to add to the collection. |
| ***key*** | Optional. A unique string expression that specifies a key string that can be used, instead of a positional index, to access a member of the collection. |
| ***before*** | Optional. An expression that specifies a relative position in the collection. The member to be added is placed in the collection before the member identified by the ***before*** argument. If a numeric expression, ***before*** must be a number from 1 to the value of the collection's **Count** property. If a string expression, ***before*** must correspond to the ***key*** specified when the member being referred to was added to the collection. You can specify a ***before*** position or an ***after*** position, but not both. |
| ***after*** | Optional. An expression that specifies a relative position in the collection. The member to be added is placed in the collection after the member identified by the ***after*** argument. If numeric, ***after*** must be a number from 1 to the value of the collection's **Count** property. If a string, ***after*** must correspond to the ***key*** specified when the member referred to was added to the collection. You can specify a ***before*** position or an ***after*** position, but not both. |

**Remarks**

Whether the ***before*** or ***after*** argument is a string expression or numeric expression, it must refer to an existing member of the collection, or an error occurs.

An error also occurs if a specified ***key*** duplicates the ***key*** for an existing member of the collection.

# Visual Basic for Applications Reference

## Add Method Example

This example uses the **Add** method to add `Inst` objects (instances of a class called `Class1` containing a **Public** variable `InstanceName`) to a collection called `MyClasses`. To see how this works, insert a class module and declare a public variable called `InstanceName` at module level of `Class1` (type **Public** `InstanceName`) to hold the names of each instance. Leave the default name as `Class1`. Copy and paste the following code into the Form_Load event procedure of a form module.

```
Dim MyClasses As New Collection   ' Create a Collection object.
Dim Num As Integer   ' Counter for individualizing keys.
Dim Msg
Dim TheName   ' Holder for names user enters.
Do
   Dim Inst As New Class1   ' Create a new instance of Class1.
   Num = Num + 1   ' Increment Num, then get a name.
   Msg = "Please enter a name for this object." & Chr(13) _
    & "Press Cancel to see names in collection."
   TheName = InputBox(Msg, "Name the Collection Items")
   Inst.InstanceName = TheName   ' Put name in object instance.
   ' If user entered name, add it to the collection.
   If Inst.InstanceName <> "" Then
      ' Add the named object to the collection.
      MyClasses.Add item := Inst, key := CStr(Num)
   End If
   ' Clear the current reference in preparation for next one.
   Set Inst = Nothing
Loop Until TheName = ""
For Each x In MyClasses
   MsgBox x.instancename, , "Instance Name"
Next
```

# Visual Basic Reference

**Visual Studio 6.0**

# Add Method (Format Objects)

See Also Example Applies To

Adds a **StdDataFormat** object to a **StdDataFormats** collection.

**Syntax**

*object*.**Add**(*dataformat*, [*index*])

| Part | Description |
|---|---|
| *object* | An object expression that evaluates to an object in the Applies To list. |
| *dataformat* | Required. A string expression specifying the name of the object to add to the collection. |
| *index* | Optional. An integer that uniquely identifies a member of the collection. |

# Visual Basic Extensibility Reference

**Visual Studio 6.0**

# AddCustom Method

See Also Example Applies To

Returns a **VBComponent** object, or creates a new custom component and adds it to the project.

**Syntax**

*object*.**AddCustom** (**ByVal** *progid* **As String**) **As VBComponent**

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

| Part | Description |
|---|---|
| *object* | An object expression that evaluates to an object in the Applies To list. |
| *progid* | Required. The ProgID of the custom component to be created. |

# Visual Basic Extensibility Reference

**Visual Studio 6.0**

# AddFile Method

See Also Example Applies To

Returns the newly added component.

**Syntax**

*object*.**AddFile** (**ByVal** *pathname* **As String**, [*relateddocument* **As Boolean**]) **As VBComponent**

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

| Part | Description |
|---|---|
| *object* | An object expression that evaluates to an object in the Applies To list. |
| *pathname* | Required. A string expression specifying the path and filename of the file to open as a template. |
| *relateddocument* | Optional (for text files only). Default = **False**. A Boolean expression specifying whether the file is to be treated as a standard module or a document. If set to **True**, then the file added is treated as a document file. |

**Remarks**

Files that are normally Visual Basic project components, such as forms, cause an error if the *relateddocument* parameter is set to **True**. The *relateddocument* parameter is required only when adding text files that can be treated as either standard modules or documents.

# Visual Basic for Applications Reference

**Visual Studio 6.0**

# Add Method (Folders)

See Also Example Applies To Specifics

**Description**

Adds a new **Folder** to a **Folders** collection.

**Syntax**

*object*.**Add** *folderName*

The **Add** method has the following parts:

| Part | Description |
|---|---|
| *object* | Required. Always the name of a **Folders** collection. |
| *folderName* | Required. The name of the new **Folder** being added. |

**Remarks**

An error occurs if the *folderName* already exists.

# Visual Basic Extensibility Reference

**Visual Studio 6.0**

# AddFromFile Method

See Also Example Applies To Specifics

For the **References** collection, adds a reference to a project from a file. For the **CodeModule** object, adds the contents of a file to a module.

**Syntax**

*object*.**AddFromFile(***filename***)**

The **AddFromFile** syntax has these parts:

| Part | Description |
|---|---|
| *object* | Required. An object expression that evaluates to an object in the Applies To list. |
| *filename* | Required. A string expression specifying the name of the file you want to add to the project or module. If the file name isn't found and a path name isn't specified, the directories searched by the **Windows OpenFile** function are searched. |

**Remarks**

For the **CodeModule** object, the **AddFromFile** method inserts the contents of the file starting on the line preceding the first procedure in the code module. If the module doesn't contain procedures, **AddFromFile** places the contents of the file at the end of the module.

# Visual Basic Extensibility Reference

## AddFromFile Method Example

The following example uses the **AddFromFile** method to add the contents of a file to a specified code pane.

```
Application.VBE.CodePanes(3).CodeModule.AddFromFile "c:\Code Files\book2.frm"
```

# Visual Basic Extensibility Reference

**Visual Studio 6.0**

# AddFromFile Method

See Also Example Applies To

Adds or opens a project or group project.

**Syntax**

*object*.**AddFromFile** (**ByVal** *pathname* **As String**, [*exclusive* **As Boolean**]) **As VBNewProjects**

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

| Part | Description |
|---|---|
| *object* | An object expression that evaluates to an object in the Applies To list. |
| *pathname* | Required. A string expression specifying the path to the file to use as the template. |
| *exclusive* | Optional. Default = **False**. A Boolean expression. If set to **True**, then the existing group project is closed and the new project is created as the only open project. |

**Remarks**

If the file is a group project file and *exclusive* is set to **False**, then all projects in that group project are added to the current group project. If the file is a group project file and *exclusive* is set to **True**, then the current group project is replaced by the specified one.

# Visual Basic Extensibility Reference

**Visual Studio 6.0**

# AddFromGuid Method

See Also Example Applies To Specifics

Adds a reference to the **References** collection using the globally unique identifier (GUID) of the reference.

**Syntax**

*object*.**AddFromGuid(***guid*, *major*, *minor***) As Reference**

The **AddFromGuid** syntax has these parts:

| Part | Description |
|---|---|
| *object* | Required. An object expression that evaluates to an object in the Applies To list. |
| *guid* | Required. A string expression representing the GUID of the reference. |
| *major* | Required. A Long specifying the major version number of the reference. |
| *minor* | Required. A **Long** specifying the minor version number of the reference. |

**Remarks**

The **AddFromGuid** method searches the registry to find the reference you want to add. The GUID can be a type library, control, class identifier, and so on.

# Visual Basic Extensibility Reference

## AddFromGUID Method Example

The following example uses the **AddFromGUID** method to add a reference to the current project, identifying the reference using the globally unique ID value of the **Reference** object.

```
Application.VBE.ActiveVBProject.References.AddFromGuid("{000204EF-0000-0000-C000-000000000046}", 5, 0)
```

# Visual Basic Extensibility Reference

**Visual Studio 6.0**

# AddFromString Method

See Also Example Applies To Specifics

Adds text to a module.

**Syntax**

*object*.**AddFromString**

The *object* placeholder is an object expression that evaluates to an object in the Applies To list.

**Remarks**

The **AddFromString** method inserts the text starting on the line preceding the first procedure in the module. If the module doesn't contain procedures, **AddFromString** places the text at the end of the module.

# Visual Basic Extensibility Reference

## AddFromString Method Example

The following example uses the **AddFromString** method to add a line, Dim intJack As Integer, to the specified code pane.

```
Application.VBE.CodePanes(3).CodeModule.AddFromString "Dim intJack As Integer"
```

# Visual Basic Extensibility Reference

**Visual Studio 6.0**

# AddFromTemplate Method

See Also Example Applies To

- VBComponents collection: Returns the newly created component, and creates a new component from a template.
- VBProjects collection: Returns a collection of all projects added as a result of a call to this method, or creates a new project using an existing project as a template.

**Syntax**

*object*.**AddFromTemplate** (*filename* **As String**) **As VBComponent**

*object*.**AddFromTemplate** (**ByVal** *pathname* **As String**, [*exclusive* **As Boolean**]) **As VBNewProjects**

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

| Part | Description |
|---|---|
| *object* | An object expression that evaluates to an object in the Applies To list. |
| *filename* | Required. A string expression specifying the path and filename of the file to open as a template. |
| *exclusive* | Optional. Default = **False**. A Boolean expression. If set to **True**, then the existing group project is closed and the new project is created as the only open project. |
| *pathname* | Required. A string expression specifying the path to the file to use as the template. |

**Remarks**

If the file type referenced is a group project file, and *exclusive* is set to **False**, then all projects in that file are created as templates and added to the current set of open projects. If *exclusive* is set to **True**, however, the current group project is closed and a new group project created, and all projects within the group project template are created as project templates. The object returned by the method is **Nothing**.

New project or projects are given the usual default names.

# Visual Basic: MSFlexGrid/MSHFlexGrid Controls

**Visual Studio 6.0**

# AddItem Method (MSHFlexGrid)

SeeAlso Example Applies To

Adds a row to an **MSHFlexGrid**. This property doesnt support named arguments.

**Syntax**

*object*.**AddItem(***string*, *index***)**

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

| Part | Description |
|---|---|
| *object* | An object expression that evaluates to an object in the Applies To list. |
| *string* | Required. A string expression displayed in the newly added row. To add multiple strings (for multiple columns in the row), use the tab character (**vbTab**) to separate each string. |
| *index* | Optional. A Long value indicating the position within the control. This position is where the new row is placed. For the first row, *index*=0. If *index* is omitted, the new row becomes the last row in the band. Note that *index* is **BandColIndex** in the **MSHFlexGrid**. |

**Remarks**

Not available when the control is bound to a hierarchical recordset.

# Visual Basic: MSFlexGrid/MSHFlexGrid Controls

# AddItem, RemoveItem Methods (MSHFlexGrid) Example

This example uses the **AddItem** method to add 100 items to an **MSHFlexGrid**. To use this example, paste the code into the Declarations section of a form with an **MSHFlexGrid** named MSHFlexGrid1, press F5, and then click the form.

**Note** If you are using the **MSFlexGrid**, substitute "MSHFlexGrid1" with "MSFlexGrid1."

```
Private Sub Form_Click ()
   Dim Entry, i, Msg               ' Declare variables.
   Msg =_
   "Choose OK to add 100 items to your MSHFlexGrid."
   MsgBox Msg   ' Display message.
   MSHFlexGrid1.Cols =2         ' Two strings per row.
   For i =1 To 100   ' Count from 1 to 100.
      Entry ="Entry " & Chr(9) & I    ' Create entry.
      MSHFlexGrid1.AddItem Entry      ' Add entry.
   Next i
   Msg ="Choose OK to remove every other entry."
   MsgBox Msg                     ' Display message.
   For i =1 To 50               ' Determine how to
      MSHFlexGrid1.RemoveItem i    ' remove every other
   Next I                        ' item.
   Msg ="Choose OK to clear all items."
   MsgBox Msg                     ' Display message.
   MSHFlexGrid1.Clear               ' Clear list box.
End Sub
```

# Visual Basic Reference

**Visual Studio 6.0**

# AddItem Method

See Also Example Applies To

Adds an item to a **ListBox** or **ComboBox** control or adds a row to a **MS Flex Grid** control. Doesn't support named arguments.

**Syntax**

*object*.**AddItem** *item*, *index*

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

| Part | Description |
|---|---|
| *object* | Required. An object expression that evaluates to an object in the Applies To list. |
| *item* | Required. string expression specifying the item to add to the object. For the **MS Flex Grid** control only, use the tab character (character code 09) to separate multiple strings you want to insert into each column of a newly added row. |
| *index* | Optional. Integer specifying the position within the object where the new item or row is placed. For the first item in a **ListBox** or **ComboBox** control or for the first row in a **MS Flex Grid** control, *index* is 0. |

**Remarks**

If you supply a valid value for *index*, *item* is placed at that position within the *object*. If *index* is omitted, *item* is added at the proper sorted position (if the **Sorted** property is set to **True**) or to the end of the list (if **Sorted** is set to **False**).

A **ListBox** or **ComboBox** control that is bound to a **Data** control doesn't support the **AddItem** method.

# Visual Basic Reference

# AddItem Method Example

This example uses the **AddItem** method to add 100 items to a list box. To try this example, paste the code into the Declarations section of a form with a **ListBox** control named List1, and then press F5 and click the form.

```
Private Sub Form_Click ()
   Dim Entry, I, Msg   ' Declare variables.
   Msg = "Choose OK to add 100 items to your list box."
   MsgBox Msg   ' Display message.
   For I = 1 To 100   ' Count from 1 to 100.
      Entry = "Entry " & I   ' Create entry.
      List1.AddItem Entry   ' Add the entry.
   Next I
   Msg = "Choose OK to remove every other entry."
   MsgBox Msg   ' Display message.
   For I = 1 To 50   ' Determine how to
      List1.RemoveItem I   ' remove every other
   Next I   ' item.
   Msg = "Choose OK to remove all items from the list box."
   MsgBox Msg   ' Display message.
   List1.Clear   ' Clear list box.
End Sub
```

# Visual Basic: RDO Data Control

**Visual Studio 6.0**

# AddNew Method (Remote Data)

See Also Example Applies To

Creates a new row for an updatable **rdoResultset** object.

**Syntax**

*object*.**AddNew**

The *object* placeholder represents an object expression that evaluates to an object in the Applies To list.

**Remarks**

The **AddNew** method prepares a new row you can edit and subsequently add to the **rdoResultset** object named by *object* using the **Update** method. This method initializes the columns to SQL_IGNORE to ensure columns not specifically referenced are not included in the update operation.

When the **AddNew** method is executed, the **EditMode** property is set to **rdEditAdd** until you execute the **Update** method.

After you modify the new row, use the **Update** method to save the changes and add the row to the result set. No changes are made to the database until you use the **Update** method unless you are using the Client Batch cursor library which does not write to the database until the **BatchUpdate** method is used.

The **AddNew** method does not return an error if the **rdoResultset** is not updatable. A trappable error is triggered when the **Update** method is used against an object that is not updatable. For an object to be updatable, the **rdoColumn**, **rdoResultset**, and **rdoConnection** objects must all be updatable check the **Updatable** property of each of these objects before performing an update. There are a variety of reasons why an **rdoResultset** is not updatable as discussed in the **Update** method topic.

**Caution** If you use the **AddNew** method on a row and then perform any operation that moves to another row without using **Update**, your changes are lost without warning. In addition, if you close the *object* or end the procedure which declares the *object* or its **rdoConnection** object, the new row and the changes made to it are discarded without warning.

A newly added row might be visible as a part of the **rdoResultset** if your data source and type of cursor support it. For example, newly added rows are not included in a static-type **rdoResultset**.

When newly added rows are included in the **rdoResultset**, the row that was current *before* you used **AddNew** remains current. When the row is added to the cursor keyset, and you want to make the new row current, you can set the **Bookmark** property to the bookmark identified by the **LastModified** property setting.

If you need to cancel a pending **AddNew** operation, use the **CancelUpdate** method.

When you use the **Update** method after using the **AddNew** method, the RowCurrencyChange event is fired.

# Visual Basic: RDO Data Control

# AddNew, Update, CancelUpdate Method Example

The following example illustrates use of the **AddNew** method to add new rows to a base table. This example assumes that you have read-write access to the table, that the column data provided meets the rules and other constraints associated with the table, and there is a unique index on the table. The data values for the operation are taken from three **TextBox** controls on the form. Note that the unique key for this table is not provided here as it is provided automatically it is an *identity* column.

```
Option Explicit
Dim er As rdoError
Dim cn As New rdoConnection
Dim qy As New rdoQuery
Dim rs As rdoResultset
Dim col As rdoColumn

Private Sub AddNewJob_Click()
On Error GoTo ANEH

With rs
   .AddNew
   !job_desc = JobDescription
   !min_lvl = MinLevel
   !max_lvl = MaxLevel
   .Update
End With
Exit Sub

UpdateFailed:
MsgBox "Update did not suceed."
rs.CancelUpdate
Exit Sub
A
NEH:
Debug.Print Err, Error
For Each er In rdoErrors
   Debug.Print er
Next
Resume UpdateFailed

End Sub

Private Sub Form_Load()

cn.CursorDriver = rdUseOdbc
cn.Connect = "uid=;pwd=;server=sequel;" _
   & "driver={SQL Server};database=pubs;dsn=;"
cn.EstablishConnection
With qy
   .Name = "JobsQuery"
   .SQL = "Select * from Jobs"
```

```
    .RowsetSize = 1
    Set .ActiveConnection = cn
    Set rs = .OpenResultset(rdOpenKeyset, _
       rdConcurRowver)
    Debug.Print rs.Updatable
End With

Exit Sub
End Sub
```

# Visual Basic Reference

**Visual Studio 6.0**

# AddToAddInToolbar Method

See Also Example Applies To

Inserts a button on the Add-In toolbar which references an add-in or Wizard.

**Syntax**

*object*.**AddToAddInToolbar** (*sfilename* **As String**, *sprogid* **As String**, *showontoolbar* **As Boolean**, *forceaddintoolbar* **As Boolean**)

| Part | Description |
|---|---|
| *object* | An object expression that evaluates to an object in the Applies To list. |
| *sfilename* | Required. A string expression specifying the path to the add-in or Wizard and the name of its .Exe or .Dll file. |
| *sprogid* | Required. A string expression specifying the programmatic ID (ProgID) of the add-in or Wizard. |
| *saddinname* | Required. A string expression specifying the title of the add-in or Wizard. |
| *showontoolbar* | Required. A Boolean expression specifying whether the add-in or Wizard referred to will appear on the Add-In toolbar. **True** = yes, **False** = no. |
| *forceaddintoolbar* | Required. A Boolean expression specifying whether the Add-In toolbar is automatically displayed the next time Visual Basic is started. **True** = yes, **False** = no. |

# Visual Basic Reference

# AddToAddInToolbar Method Example

This example uses the **AddToAddInToolbar** method to add a button to the Add-In toolbar for a fictitious add-in called MyAdd.Dll. Setting ForceAddInToolbar to **True** ensures that the Add-In toolbar is loaded the next time Visual Basic is started.

You could modify the following in a small Visual Basic application to serve as a Setup for your add-in.

```
Sub Main()
 dim x as Object
 Set x=CreateObject("AddInToolbar.Manager")
 x.AddToAddInToolbar sFileName:="C:\VB5\MyAdd.DLL", _
  sProgID:="MyAddIn.Connect", _
  sAddInName:="MyAddIn Title", _
  ShowOnToolBar:=True, _
  ForceAddInToolbar:=True
End Sub
```

# Visual Basic Extensibility Reference

**Visual Studio 6.0**

# AddToolboxProgID Method

See Also Example Applies To

Places the control or embedded component in the toolbox and adds a control reference to the project.

**Syntax**

*object*.**AddToolboxProgID** (**ByVal** *progid* **As String**, [*filename* **As String**])

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

| Part | Description |
|---|---|
| *object* | An object expression that evaluates to an object in the Applies To list. |
| *progid* | Required. A string expression specifying the programmatic identifier (ProgID) of the compound document object to add to the Visual Basic toolbox. Either a version-independent or version-dependent ProgID can be used. If a version-independent progid is specified, the most recent version is used. If the compound document object has an associated type library, this type library is referenced as well. |
| *filename* | Optional. A string expression specifying the filename of the desired type library to be added to Visual Basic. A complete pathname can be used, but if the file isn't found, the directories searched by the Windows **OpenFile** function are searched, even if a complete pathname is specified. |

# Visual Basic: RDO Data Control

**Visual Studio 6.0**

# AppendChunk Method (Remote Data)

See Also Example Applies To

Appends data from a Variant expression to an **rdoColumn** object with a data type of **rdTypeLONGVARBINARY** or **rdTypeLONGVARCHAR**.

**Syntax**

*object* **!** *column*.**AppendChunk** *source*

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

| Part | Description |
|---|---|
| *object* | An object expression that evaluates to the **rdoResultset** object containing the **rdoColumns** collection. |
| *column* | An object expression that evaluates to an **rdoColumn** object whose **ChunkRequired** property is set to **True**. |
| *source* | A string expression or variable containing the data you want to append to the **rdoColumn** object specified by *column*. |

**Remarks**

*Chunk* data columns are designed to store large binary (BLOB) or text values that can range in size from a few characters to over 1.2GB and are stored in the database on successive data pages. In many cases, *chunk* data cannot be managed with a single operation, so you must use the *chunk* methods to save and write data. If the **ChunkRequired** property is **True** for a column, you should use the **AppendChunk** method to manipulate column data. However, if there is sufficient internal memory available, RDO might be able to carry out the operation without use of the **AppendChunk** method. In other words, you might be able to simply assign a value to a BLOB column.

Use the **AppendChunk** method to write successive blocks of data to the database column and **GetChunk** to extract data from the database column. Certain operations (copying, for example) involve temporary strings. If string space is limited, you may need to work with smaller segments of a *chunk* column instead of the entire column.

Use the **BindThreshold** property to specify the largest column size that will be automatically bound.

Use the **ColumnSize** property to determine the number of bytes in a *chunk* column. Note that for variable-sized columns, it is not necessary to write back the same number of bytes as returned by the **ColumnSize** property as **ColumnSize** reflects the size of the column before changes are made.

If there is no current row when you use **AppendChunk**, a trappable error occurs.

**Note** The initial **AppendChunk** (after the first **Edit** method), even if the row already contains data, replaces existing column data. Subsequent **AppendChunk** calls within a single **Edit** session appends data to existing column data.

# Visual Basic: RDO Data Control

# AppendChunk, GetChunk Method Example

This example illustrates use of the **AppendChunk** and **GetChunk** methods to write page-based binary large object (BLOB) data to a remote data source. The code expects a table with a char, text, and image field named *Chunks*. To create this table, submit the following as an action query against your test database:

```
CREATE TABLE Chunks (ID integer identity NOT NULL, PName char(10) NULL,
Description TEXT NULL,
Photo IMAGE NULL)
CREATE UNIQUE INDEX ChunkIDIndex on Chunks(ID)
```

Once the table is created, you will need to locate one or more .BMP or other suitable graphics images that can be loaded by the **PictureBox** control.

```
'
Option Explicit
Dim en As rdoEnvironment
Dim Qd As rdoQuery
Dim Cn As rdoConnection
Dim Rs As rdoResultset
Dim SQL As String
Dim DataFile As Integer, Fl As Long, Chunks As Integer
Dim Fragment As Integer, Chunk() As Byte, I As Integer
Const ChunkSize As Integer = 16384

Private Sub Form_Load()
Set en = rdoEnvironments(0)
Set Cn = en.OpenConnection(dsname:="", _
   Connect:="UID=;PWD=;DATABASE=WorkDB;" _
   & "Driver={SQL Server};SERVER=Betav486", _
   prompt:=rdDriverNoPrompt)
Set Qd = Cn.CreateQuery("TestChunk", "Select * from
   Chunks Where PName = ?")
End Sub

Private Sub LoadFromFile_Click()
'
'  Locates a file and sets the Filename to this file.
'
With CommonDialog1
   .Filter = "Pictures(*.bmp;*.ico)|*.bmp;*.ico"
   .ShowOpen
   FileName = .FileName
End With
End Sub

Private Sub ReadFromDB_Click()
If Len(NameWanted) = 0 Then _
   NameWanted = InputBox("Enter name wanted", "Animal")
   Qd(0) = NameWanted
   Set Rs = Qd.OpenResultset(rdOpenKeyset, _
   rdConcurRowver)
```

```
If Rs Is Nothing Or Rs.Updatable = False Then
   MsgBox "Cant open or write to result set"
   Exit Sub
End If
If Rs.EOF Then
   MsgBox "Cant find picture by that name"
   Exit Sub
End If
Description = Rs!Description
DataFile = 1
Open "pictemp" For Binary Access Write As DataFile
Fl = Rs!Photo.ColumnSize
Chunks = Fl \ ChunkSize
Fragment = Fl Mod ChunkSize
ReDim Chunk(Fragment)
Chunk() = Rs!Photo.GetChunk(Fragment)
Put DataFile, , Chunk()
For I = 1 To Chunks
   ReDim Buffer(ChunkSize)
   Chunk() = Rs!Photo.GetChunk(ChunkSize)
   Put DataFile, , Chunk()
Next I
Close DataFile
FileName = "pictemp"
End Sub

Private Sub SaveToDB_Click()
If Len(NameWanted) = 0 Then _
   NameWanted = InputBox("Enter name for this" _
   & " picture", "Animal")
   Qd(0) = NameWanted
   Set Rs = Qd.OpenResultset(rdOpenKeyset, _
   rdConcurRowver)
If Rs Is Nothing Or Rs.Updatable = False Then
   MsgBox "Cant open or write to result set"
   Exit Sub
End If
If Rs.EOF Then
   Rs.AddNew
   Rs!PName = NameWanted
If Description = "" Then  _
   Description = InputBox("Describe the picture", _
   "Dont care")
   'Rs!Description = Description
Else
   Rs.Edit
End If
DataFile = 1
Open FileName For Binary Access Read As DataFile
Fl = LOF(DataFile)    ' Length of data in file
If Fl = 0 Then Close DataFile: Exit Sub
Chunks = Fl \ ChunkSize
Fragment = Fl Mod ChunkSize
Rs!Photo.AppendChunk Null
ReDim Chunk(Fragment)
Get DataFile, , Chunk()
Rs!Photo.AppendChunk Chunk()
ReDim Chunk(ChunkSize)
For I = 1 To Chunks
   Get DataFile, , Chunk()
   Rs!Photo.AppendChunk Chunk()
```

```
Next I
Close DataFile
Rs.Update
End Sub

Private Sub FileName_Change()
Picture1.Picture = LoadPicture(FileName)
End Sub
```

# Visual Basic Reference

**Visual Studio 6.0**

# Arrange Method

See Also Example Applies To

Arranges the windows or icons within an **MDIForm** object. Doesn't support named arguments.

### Syntax

*object*.**Arrange** *arrangement*

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

| Part | Description |
|---|---|
| *object* | Required. An object expression that evaluates to an object in the Applies To list. |
| *arrangement* | Required. A value or constant that specifies how to arrange windows or icons on an **MDIForm** object, as described in Settings. |

### Settings

The settings for *arrangement* are:

| Constant | Value | Description |
|---|---|---|
| **vbCascade** | 0 | Cascades all nonminimized MDI child forms |
| **vbTileHorizontal** | 1 | Tiles all nonminimized MDI child forms horizontally |
| **vbTileVertical** | 2 | Tiles all nonminimized MDI child forms vertically |
| **vbArrangeIcons** | 3 | Arranges icons for minimized MDI child forms |

### Remarks

These constants are listed in the Visual Basic (VB) object library in the Object Browser.

Windows or icons are arranged even if the **MDIForm** object is minimized. Results are visible when the **MDIForm** is maximized.

# Visual Basic Reference

# Arrange Method Example

This example uses the **Arrange** method to arrange windows and icons in an MDI form. To try this example, paste the code into the Declarations section of an MDI form named MDIForm1 that has an MDI child form (named Form1, with its **MDIChild** property set to **True**) and a picture box on the MDI Form (named Picture1). Press F5 and click anywhere in the picture box to see the effects of the **Arrange** method.

```
Const FORMCOUNT = 5
Dim F(1 To FORMCOUNT) As New Form1
Private Sub MDIForm_Load ()
   Dim I   ' Declare local variable.
   Load Form1   ' Load original Form1.
   For I = 1 To FORMCOUNT
      F(I).Caption = "Form" & I + 1   ' Change caption on copies.
   Next I
End Sub

Private Sub Picture1_Click ()
   Static ClickCount   ' Declare variables.
   Dim I, PrevWidth, Start
   ClickCount = ClickCount + 1   ' Increment click counter.
   Select Case ClickCount
      Case 1
         MDIForm1.Arrange 1   ' Tile horizontally.
      Case 2
         MDIForm1.Arrange 2   ' Tile vertically.
      Case 3   ' Minimize each form.
         PrevWidth = MDIForm1.Width   ' Get MDI form width.
         MDIForm1.Width = PrevWidth / 2  ' Divide it in half.
         Form1.WindowState = 1   ' Minimize the original.
         For I = 1 To FORMCOUNT   ' Look at each instance of F.
            F(I).WindowState = 1   ' Minimize each copy of F.
         Next I
         Start = Timer
         Do
         Loop Until Timer = Start + 5
         MDIForm1.Width = PrevWidth   ' Resize to original size.
         MDIForm1.Arrange 3   ' Arrange icons.
   End Select
End Sub
```

# Visual Basic for Applications Reference

**Visual Studio 6.0**

# Assert Method

See Also Example Applies To Specifics

Conditionally suspends execution at the line on which the method appears.

**Syntax**

*object*.**Assert** *booleanexpression*

The **Assert** method syntax has the following object qualifier and argument:

| Part | Description |
|---|---|
| *object* | Required. Always the **Debug** object. |
| *booleanexpression* | Required. An expression that evaluates to either **True** or **False**. |

**Remarks**

**Assert** invocations work only within the development environment. When the module is compiled into an executable, the method calls on the **Debug** object are omitted.

All of *booleanexpression* is always evaluated. For example, even if the first part of an **And** expression evaluates **False**, the entire expression is evaluated.

# Visual Basic for Applications Reference

## Assert Method Example

The following example shows how to use the **Assert** method. The example requires a form with two button controls on it. The default button names are Command1 and Command2.

When the example runs, clicking the Command1 button toggles the text on the button between 0 and 1. Clicking Command2 either does nothing or causes an assertion, depending on the value displayed on Command1. The assertion stops execution with the last statement executed, the Debug.Assert line, highlighted.

```
Option Explicit
Private blnAssert As Boolean
Private intNumber As Integer

Private Sub Command1_Click()
    blnAssert = Not blnAssert
    intNumber = IIf(intNumber <> 0, 0, 1)
    Command1.Caption = intNumber
End Sub

Private Sub Command2_Click()
    Debug.Assert blnAssert
End Sub

Private Sub Form_Load()
    Command1.Caption = intNumber
    Command2.Caption = "Assert Tester"
End Sub
```

# Visual Basic Reference

**Visual Studio 6.0**

# AsyncRead Method

See Also Example Applies To

Initiates the asynchronous reading of data from a file or URL by a container, control, or ActiveX component.

**Syntax**

*object*.**AsyncRead** *Target*, *AsyncType* [, *PropertyName*], [*AsyncReadOptions*]

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

| Part | Description |
|---|---|
| *object* | An object expression that evaluates to an object in the Applies To list. |
| *Target* | A string expression specifying the location of the data. This can be a path or a URL. |
| *AsyncType* | An integer expression specifying how the data is presented, as described in Settings. |
| *PropertyName* | An optional string expression specifying the name of the property to be loaded. Required for differentiating between simultaneous downloads. |
| *AsyncReadOptions* | An optional string expression specifying additional options for **AsyncRead** as described in Settings. |

**Settings**

The settings for *AsyncType* are:

| Setting | Description |
|---|---|
| **vbAsyncTypeFile** | The data is provided in a file created by Visual Basic. |
| **vbAsyncTypeByteArray** | The data is provided as a byte array that contains the retrieved data. It is assumed that the component author knows how to handle the data. |
| **vbAsyncTypePicture** | The data is provided in a **Picture** object. |

The settings for *AsyncReadOptions* are:

| Constant | Setting | Description |
|---|---|---|
| **vbAsyncReadSynchronousDownload** | 1 | **AsyncRead** doesn't return until the AsyncReadComplete event has occurred. |
| **vbAsyncReadOfflineOperation** | 8 | **AsyncRead** uses only the locally cached resource. |
| **vbAsyncReadForceUpdate** | 16 | **AsyncRead** forces a download of the resource from the server, ignoring a locally cached copy. |
| **vbAsyncReadResynchronize** | 512 | **AsyncRead** updates the locally cached copy only if the server version is newer. |
| **vbAsyncReadGetFromCacheIfNetFail** | 524288 | **AsyncRead** uses the cached copy if the server connection is unsuccessful. |

**Remarks**

The progress of a download that is requested by the **AsyncRead** method can be tracked by the AsyncReadProgress event. Once the data is available, the AsyncReadComplete event is raised in the object. The asynchronous read can be canceled before it is completed by calling the **CancelAsyncRead** method.

The *PropertyName* parameter is a tag, meaning that it can be any arbitrary string, since its only function is to act as an identifier for this particular data request. The value in *PropertyName* is used to identify the particular asynchronous read to cancel in the **CancelAsyncRead** method, and the value in *PropertyName* is also used to identify the particular asynchronous read that has completed in the AsyncReadComplete event. Only one AsyncRead event with a given *PropertyName* can be active at one time.

The **AsyncRead** method initiates an asynchronous download. The **AsyncRead** events fire synchronously (before this method returns) if the data is already available on the client machine. **AsyncRead** can raise some errors synchronously (such as "bad parameter", "unknown protocol", "UrlMon.dll missing", and so forth), so it's a good idea to have appropriate error handling code before calling **AsyncRead**. If the data is not available locally, then **AsyncRead** returns immediately and the **AsyncRead** events occur later.