

Understanding the Word Object Model from a .NET Developer's Perspective

Office 2003

Mary Chipman
MCW Technologies, LLC

April 2003

Applies to:

Microsoft® Visual Studio® Tools for the Microsoft Office System
Microsoft Office Word 2003
Microsoft Visual Studio® .NET 2003

Summary: Provides information on how to use Microsoft Visual Studio Tools for the Microsoft Office System to take advantage of the objects available in Microsoft Office Word 2003. It introduces several important Word objects and provides examples of how to use them. You will learn how to work with Word 2003 applications and documents, as well as with some of the more important properties and methods. (105 printed pages)

Download the WordObject.exe from the [Microsoft Download Center](#).

Contents

[Introduction](#)
[Getting Started with a Word Project](#)
[The Underpinnings: Documents and Templates](#)
[Bird's-Eye View of the Word Object Model](#)
[The Application Object](#)
[The Document Object](#)
[The Selection Object](#)
[The Range Object](#)
[The Bookmark Object](#)
[Searching and Replacing Text](#)
[Printing](#)
[Creating Word Tables](#)
[Summary](#)

Introduction

Microsoft® Word is probably one of the most commonly used software products in the world today. Most people using Word get by just fine without writing any code at all, although Word has a rich and powerful object model making it eminently programmable. Microsoft Visual Studio Tools for the Microsoft Office System enables developers to interact with the objects provided by the Microsoft Office Word 2003 object model by using a .NET language, such as Microsoft Visual Basic® .NET or Microsoft Visual C#®. Word possesses a rich feature set, all of which are accessible through code. There are quite a few objects to learn about, which can be confusing when you're first getting started. Conveniently, Word objects are arranged in a hierarchical fashion, and you can get a good start on the object model by focusing on the two main classes at the top of the hierarchy, the **Application** and **Document** classes. Focus on these two classes makes sense when you consider that most of the time you'll either be working with the Word application itself, or manipulating Word documents in some way.

As you dig into the Word object model, you'll find that it emulates the Word user interface, making it easy to guess that the **Application** object provides a wrapper around the entire application, each **Document** object represents a single Word document, the **Paragraph** object corresponds to a single paragraph, and so forth. Each of these objects has many methods and properties that allow you to manipulate and interact with it. The behaviors of the members of these objects are generally easy to guess—how about the **PrintOut** method? Others can be more obscure and sometimes tricky to use correctly. Once you learn the basics, you'll find that there isn't anything you can do in the Word user interface that you can't do just as easily in code. Programming in Word allows you to automate repetitive tasks, and to extend and customize the functionality built into Word.

In this document, you'll learn how to take advantage of many of the objects in Word 2003, and you will also be introduced to some of the properties, methods, and events of each object. You'll learn how to work with Word applications and documents, as well as with some of their more important methods and properties.

Note Programming Word and its objects from Visual Basic .NET feels much like programming in VBA. Visual Basic .NET handles optional parameters, and allows late binding, just like VBA. C#, on the other hand, provides unique challenges when programming against the Word object model. Because C# doesn't support optional parameters, parameterized properties, or late binding, you'll need to handle many of the Word methods and properties specially when programming in C#. This document points out the differences between Visual Basic .NET and C# programming, as they come up.

Getting Started with a Word Project

When you create a new Office project in Visual Studio .NET, you are given the option of creating a new Word Document or Word Template project, as shown in **Figure 1**.

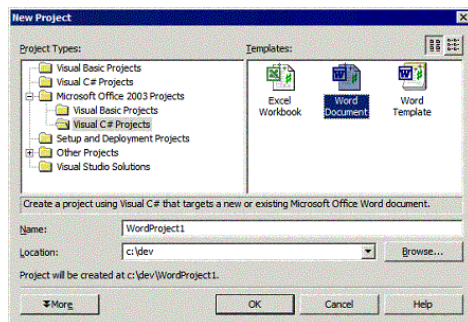


Figure 1. You can create either a Word Document or a Word Template project in Visual Studio .NET.

Visual Studio .NET automatically creates a code file named `ThisDocument.vb` or `ThisDocument.cs` in your new Word project for both Document and Template projects. Open `ThisDocument` in your new project. You'll see that a public class named **OfficeCodeBehind** has already been generated for you. Expand the hidden **Generated initialization code** region. The **OfficeCodeBehind** class includes code that wraps the **Word.Document** and **Word.Application** objects:

```
' Visual Basic
Friend WithEvents ThisDocument As Word.Document
Friend WithEvents ThisApplication As Word.Application

// C#
private Word.Application thisApplication = null;
private Word.Document thisDocument = null;
```

These two variables are declared for you:

- **ThisDocument:** Represents the Word **Document** object, and allows access to all of the built-in **Document** members in Word, including methods, properties and events.

- **ThisApplication:** Represents the Word **Application** object, and allows access to all of the **Application** object's members, including events.

The availability of these two predefined variables allows easy access to both Word objects in your code without having to declare separate **Word.Document** or **Word.Application** objects—just use **ThisDocument** and **ThisApplication**.

Each of the following sections digs into the **Document** and **Application** objects, picking specific members of each object for demonstration. Word has a very rich object model, and it would be impossible to dig into all of the members here: You'll get enough of the flavor of the object model to be able to get started, and you'll learn enough to use the Word online help for more details.

Tip Throughout this article, you'll see many uses of the **CType** and **DirectCast** methods in the Visual Basic .NET code. The reason for this is that the sample project has its **Option Strict** setting on—this means that Visual Basic .NET requires strict type conversions. Many Word methods and properties return **Object** types: For example, the `_Startup` procedure is passed variables named *application* and *document* as **Object** types, and the **CType** function is used to explicitly convert each to **Word.Application** and **Word.Document** objects, respectively. Therefore, to be as rigorous about conversions as possible, the sample has enabled **Option Strict**, and handles each type conversion explicitly. If you're a C# developer reading this document, you'll likely appreciate this decision. However, as you'll see later on, there are certain Word features that don't translate well into the object-oriented paradigm—it can sometimes be more convenient to work with **Option Strict** off. For the most part, you'll want to work with **Option Strict** on; you'll learn about the few exceptions as they arise.

The Underpinnings: Documents and Templates

Before you can effectively program Word, you need to understand how Word works. Most of the actions you'll perform in code have equivalents in the user interface on the menus and toolbars. There is also an underlying architecture that provides structure to those UI choices. One of the most important concepts is the idea of templates. You probably are already familiar with the concept of a template—a Word template can contain boilerplate text and styles as well as code, toolbars, keyboard shortcuts and AutoText entries. Whenever you create a new Word document, it is based on a template, which is distinguished by the .dot file name extension—Word documents have a .doc filename extension. The new document is linked to the template, and has access to all template items. If you do not specify a custom template, any new documents you create will be based on the Normal.dot default template, which is installed when you install Word.

About Normal.dot

The Normal.dot template is global in scope, and is available to every document you create. You could, if you wanted to, put all of your code in the Normal.dot and base all of the documents in your environment on your Normal template. But the file could become quite large, so for many developers, a better solution is to create customized templates for specific applications. Documents created using your custom template still have access to the code in the default Normal template. In fact, you can attach a document to more than one custom template in addition to Normal if you so desire.

Templates and Your Code

You are not limited to templates as containers for styles and code; you can also customize and write code in individual documents without affecting the content of the template the document is based on. When Word runs your code, it uses the fully qualified reference of the source (which can be a template or the document), the module name, and the procedure name. This operates in a similar fashion to namespaces, keeping procedures separated. **Figure 2** shows the Customize dialog box for toolbars, illustrating this concept. Each procedure is fully qualified with the name of the project, the module, and the procedure name. In this case, the item selected in the right pane refers to a procedure named **TileVertical** that is contained in the **Tile** module in **Normal.dot**. The **SaveDocument** procedure listed immediately below it is contained in the active document's code behind project.

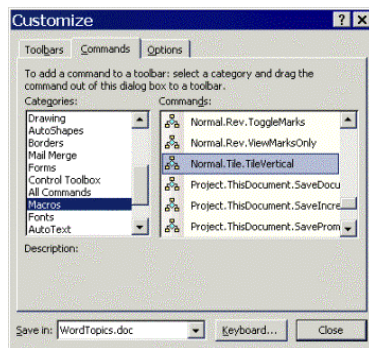


Figure 2. Procedure references are fully qualified when you assign them to a Toolbar.

Tip One thing to remember is that Word always uses the "most local" rule when it comes to templates and documents. If duplicate styles, macros, or any other items exist in all three locations—Normal.dot, a custom template, and the current document—the one in the document is used first, then the one in the custom template, and then the one in Normal.dot.

Styles and Formatting

Word allows you to format a document in two different ways:

- By direct formatting. You can select text and apply formatting options such as font, bold, italic, size, and so forth.
- By applying a style. Word comes with built-in styles that you can modify to customize your documents. When you apply a style to a paragraph or a selection, multiple attributes are applied all at once. Styles are stored in templates and in documents.

Styles are normally applied to an entire paragraph; you can also define character styles that you can apply to a character, word, or range within a paragraph. You can examine the available styles by selecting **Format | Styles and Formatting** from the menu to bring up the Styles and Formatting pane shown in **Figure 3**, where the Normal paragraph style is selected.

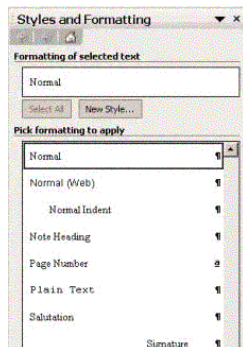
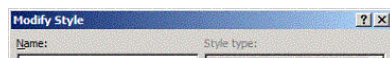


Figure 3. The default paragraph style is Normal.

Modifying Styles

When you click on the style name, it turns into a drop-down list box where you can select **Modify** from the available options. You can modify any of the built-in styles, and optionally save your changes in the template the document is based on, as shown by the **Add to template** check box setting in **Figure 4**. If you omit this check box, your changes will be saved in the document, and will not propagate back to the template the document was based on.



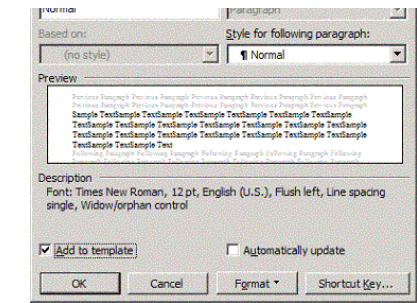


Figure 4. Modifying a style

When working with Word, you'll want to use styles as much as possible. Styles give you a way to control the formatting of complex documents in a way that would be very difficult to achieve with direct formatting. It's important to understand how they work so that you can take advantage of them to make your code more efficient.

Understanding Paragraph Marks

When you look at a document in the Word user interface, you see the document broken up into words, paragraphs, sections, and so on. Under the covers, the Word document is nothing more than a vast stream of characters. Some of these characters are meant to be read, such as the letters and numbers, and others are not, such as spaces, tabs and carriage returns. Each of these characters has a task to perform.

In addition to separating one paragraph from another, a paragraph mark plays a very important role in a Word document: it contains all of the information about how the paragraph is formatted. When you copy a paragraph and include the paragraph mark, all of the formatting in the paragraph travels along with it. If you copy a paragraph and omit the paragraph mark, the formatting of the original paragraph will be lost when it is pasted into a new location.

When you are editing a Word document and you press the ENTER key on your keyboard, a new paragraph is created that is a clone of the previous paragraph in terms of formatting, and whatever paragraph formatting or style that was in effect is propagated in the new paragraph. You can apply different formatting to the second paragraph. On the other hand, a line break, which you create by pressing SHIFT + ENTER, simply puts in a line feed character in the existing paragraph and does not hold any formatting. If you apply paragraph formatting, it will apply to all text both before and after the line break characters.

Figure 5 shows the difference between a line break and a paragraph mark when your options are set to display paragraph marks.

This is a line break... ↵
This is a paragraph mark ¶

Figure 5. The line break does not take paragraph formatting with it and a paragraph mark does.

Displaying Paragraph Marks
If you inadvertently delete a paragraph mark, it's possible you may lose your paragraph formatting. The best course of action is to ensure that they are always displayed by choosing **Tools | Options | View** from the menu and selecting the **Paragraph marks** check box in the **Formatting marks** section, as shown in Figure 6.

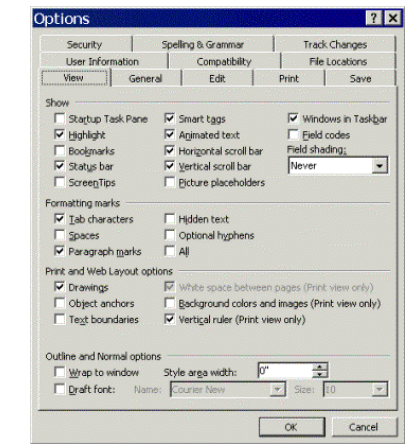


Figure 6. Displaying Paragraph marks in the Formatting marks section of the Options dialog box

Bird's-Eye View of the Word Object Model

At first glance, the Word object model is rather confusing because there appears to be a lot of overlap. For example, the **Document** and **Selection** objects are both members of the **Application** object, but the **Document** object is also a member of the **Selection** object. Both the **Document** and **Selection** objects contain **Bookmarks** and **Range** objects, as you can see in Figure 7. The following sections briefly describe the top-level objects and how they interact with each other.

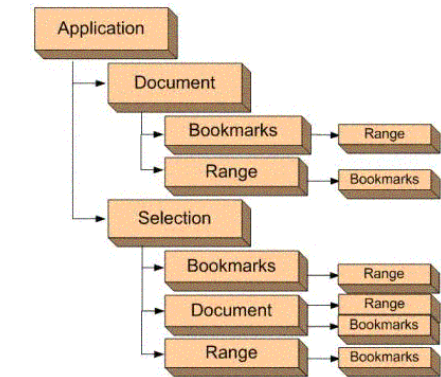


Figure 7. The Application object contains the Document, Selection, Bookmark, and Range objects.

The Application Object

The **Application** object represents the Word application, and is the parent of all of the other objects. Its members usually apply to Word as a whole. You can use its properties and methods to control the Word environment.

The Document Object

The **Document** object is central to programming Word. When you open an existing document or create a new document, you create a new **Document** object, which is added to the Word Documents collection. The document that has the focus is called the active document and is represented by the **Application** object's **ActiveDocument** property.

The Selection Object

The **Selection** object represents the area that is currently selected. When you perform an operation in the Word user interface, such as bolding text, you select the text and then apply the formatting. The **Selection** object is always present in a document; if nothing is selected, the **Selection** object represents the insertion point. The **Selection** object can also be multiple noncontiguous blocks of text.

The Range Object

The **Range** object represents a contiguous area in a document, and is defined by a starting character position and an ending character position. You are not limited to a single **Range** object; you can define multiple **Range** objects in the same document. A **Range** object has the following characteristics:

- It can consist of only an insertion point, a range of text, or the entire document.
- It includes non-printing characters such as spaces, tab characters, and paragraph marks.
- It can be the area represented by the current selection, or it can represent a different area than the current selection.
- It is dynamic; it exists only so long as the code that creates it is running.

When you insert text at the end of a range, Word automatically expands the range to include the inserted text.

The Bookmark Object

The **Bookmark** object is similar to the **Range** object in that it represents a contiguous area in a document, with both a starting position and an ending position. You use bookmarks to mark a location in a document, or as a container for text in a document. A **Bookmark** object can consist of the insertion point alone or be as large as the entire document. You can also define multiple bookmarks in a document. A **Bookmark** has the following characteristics, setting it apart from the **Range** object:

- You can give the **Bookmark** object a name.
- It is saved with the document, and does not go away when the code stops running or your document is closed.
- It is hidden by default, but can be made visible by setting the **View** object's **ShowBookmarks** property to True. (The **View** object is a member of the **Window** and **Pane** objects, which exist for both the **Application** and **Document** objects.)

Tying it all Together

Here are some scenarios for using the **Selection**, **Range**, and **Bookmark** objects:

- Bookmarks are useful in templates. For example, a business letter template can contain bookmarks where data is to be inserted from a database. At run time, your code can create a new document based on the template, obtain the data from a database, locate the named bookmark, and insert the text in the correct location.
- If you need to modify the text inside of a Bookmark, you can use the Bookmark's **Range** property to create a **Range** object, and then use one of the **Range** object's methods to modify the text.
- You can define boilerplate text in a document by using a **Bookmark** object. You specify its contents using a **Range** or a **Selection** object as the source. You can then conditionally navigate to various Bookmarks at some future time to copy and paste the boilerplate text into other documents.

These are just a few of the ways in which you can make use of these objects to build powerful customized applications.

The Application Object

The Word **Application** object represents the Word application itself. Every time you write code, you begin with the **Application** object. From the **Application** object, you can access all the other objects and collections exposed by Word, as well as properties and methods of the **Application** object itself.

Using ThisApplication

If you are working in Word, the **Application** object is automatically created for you, and you can use the **Application** property to return a reference to the Word **Application** object. When you're creating Visual Studio .NET solutions, you can use the **ThisApplication** variable defined for you within the **OfficeCodeBehind** class.

If you are automating Word from outside this class, you must create a Word **Application** object variable and then create an instance of Word:

```
' Visual Basic
Dim appWord As Word.Application = _
    New Word.Application

// C#
Word.Application appWord = new Word.Application();
```

Tip Declaring a **Word.Application** variable works in your **OfficeCodeBehind** class the same way that **ThisApplication** does. However, it's an unnecessary extra step to explicitly create a new **Word.Application** variable because **ThisApplication** has already been created for you.

When you're referring to objects and collections beneath the **Application** object, you don't need to explicitly refer to the **Application** object. For example, you can refer to the active document without the **Application** object by using the built-in **ThisDocument** property. **ThisDocument** refers to the active document, and allows you to work with members of the Document object. The **Document** object will be covered more fully in later sections of this document.

Tip The **ThisApplication.ActiveDocument** property, which refers to the active **Document** object, is likely the one you'll use most often. You'll generally want to use **ThisDocument** instead of **ThisApplication.ActiveDocument** syntax. **ActiveDocument** is available, but you need to fully qualify it with an **Application** object in order to use it in your code. Using **ThisDocument** is more efficient as the variable is already created for you, and amounts to the same thing.

Application Properties

Once you have a reference to an **Application** object, you can work with its methods and properties. The **Application** object provides a large set of methods and properties that you can use in your code to control Word. Most of the members of the **Application** object apply to global settings or the environment rather than to the contents of individual documents. Setting or retrieving some properties often requires only a single line of code; other retrievals are more complex.

- **ActiveWindow**: Returns a **Window** object that represents the window that has the focus. This property allows you to work with whatever window has the focus. The sample code below creates a new window based on the current document and then uses the **Arrange** method of a **Window** object to tile the two windows. Note that the **Arrange** method uses the **WdArrangeStyle.wdTiled** enumerated value.

```
' Visual Basic
Friend Sub CreateNewWindowAndTile()
    ' Create a new window from the active document.
    Dim wnd As Word.Window = _
        ThisApplication.ActiveWindow.NewWindow
    ' Tile the two windows.
    ThisApplication.Windows.Arrange( _
        Word.WdArrangeStyle.wdTiled)
End Sub
```

```
// C#
public void CreateNewWindowAndTile()
{
    // Create a new window from the active document.
    Word.Window wnd = ThisApplication.ActiveWindow.NewWindow();

    // Tile the two windows.
    Object value = Word.WdArrangeStyle.wdTiled;
    ThisApplication.Windows.Arrange(ref value);
}
```

Tip The **Arrange** method, like many methods in Word, requires C# developers to pass one or more parameters using the "ref" keyword. This means that the parameter you pass must be stored in a variable before you can pass it to the method. In every case, you'll need to create an Object variable, assign the variable the value you'd like to pass to the method, and pass the variable using the ref keyword. You'll find many examples of this technique throughout this document.

- **ActiveDocument:** Returns a **Document** object that represents the active document or the document that has the focus
- **ActivePrinter:** Returns or sets the name of the active printer
- **ActiveWindow:** Returns the window that has the focus
- **AutoCorrect:** Returns the current AutoCorrect options, entries, and exceptions. This property is read-only.
- **Caption:** Returns or sets the caption text for the specified document or application window. You can use the **Caption** property to display "My New Caption" in the document window or application title bar:

```
' Visual Basic
Friend Sub SetApplicationCaption()
    ' Change caption in title bar.
    ThisApplication.Caption = "My New Caption"
End Sub

// C#
public void SetApplicationCaption()
{
    // Change caption in title bar.
    ThisApplication.Caption = "My New Caption";
}
```

- **CapsLock:** Determines whether CapsLock is turned on, returning a Boolean value. The following procedure displays the state of CapsLock:

```
' Visual Basic
Friend Sub CapsLockOn()
    MessageBox.Show("CapsLock is " & _
        ThisApplication.CapsLock.ToString())
End Sub

// C#
public void CapsLockOn()
{
    MessageBox.Show(ThisApplication.CapsLock.ToString());
}
```

- **DisplayAlerts:** Lets you specify how alerts are handled when code is running, using the **WdAlertLevel** enumeration. **WdAlertLevel** contains three values: **wdAlertsAll**, which displays all messages and alerts (the default); **wdAlertsMessageBox**, which displays only message boxes; and **wdAlertsNone**, which does not display any alerts or message boxes. When you set **DisplayAlerts** to **wdAlertsNone**, your code can execute without the user seeing any messages and alerts. When you're done, you'll want to ensure that **DisplayAlerts** gets set back to **wdAlertsAll** (generally, you'll reset this in a Finally block):

```
' Visual Basic
Friend Sub DisplayAlerts()
    Try
        ' Turn off display of messages and alerts.
        ThisApplication.DisplayAlerts = Word.WdAlertLevel.wdAlertsNone
        ' Your code runs here without any alerts.
        ' . . .code doing something here.
    Finally
        ' Turn alerts on again when done.
        ThisApplication.DisplayAlerts = Word.WdAlertLevel.wdAlertsAll
    End Try
End Sub

// C#
public void DisplayAlerts()
{
    // Turn off display of messages and alerts.
    try
    {
        ThisApplication.DisplayAlerts = Word.WdAlertLevel.wdAlertsNone;
        // Your code runs here without any alerts.
        // . . .code doing something here.
    }
    catch (Exception ex)
    {
        // Do something with your exception.
    }
    finally
    {
        // Turn alerts on again when done.
        ThisApplication.DisplayAlerts = Word.WdAlertLevel.wdAlertsAll;
    }
}
```

```

        ThisApplication.DisplayStatusBar = Not bln
    }
}

```

- **DisplayStatusBar**: Read/write, returns a Boolean indicating whether or not the status bar is displayed. Returns True if it is displayed and False if it is not. The following procedure toggles the display of the status bar.

```

' Visual Basic
Friend Sub ToggleStatusBar()
    Dim bln As Boolean = (ThisApplication.DisplayStatusBar)
    ThisApplication.DisplayStatusBar = Not bln
End Sub

// C#
public void ToggleStatusBar()
{
    // Toggle display of the status bar.
    bool bln = ThisApplication.DisplayStatusBar;
    ThisApplication.DisplayStatusBar = !bln;
}

```

- **FileSearch**: Searches for files using either an absolute or a relative path. You supply the search criteria, and **FileSearch** returns the name of the files found in the **FoundFiles** collection.

```

' Visual Basic
Public Sub ListAllDocFilesOnC()
    Dim str As String
    Dim sw As New StringWriter
    Try
        ThisApplication.System.Cursor =
            Word.WdCursorType.wdCursorWait
        With ThisApplication.FileSearch
            .FileName = "*.doc"
            .LookIn = "C:\\"
            .SearchSubFolders = True
            .Execute()
            For Each str In .FoundFiles
                sw.WriteLine(str)
            Next
        End With
        MessageBox.Show(sw.ToString())
    Finally
        ThisApplication.System.Cursor =
            Word.WdCursorType.wdCursorNormal
    End Try
End Sub

// C#
public void ListAllDocFilesOnC()
{
    try
    {
        ThisApplication.System.Cursor = Word.WdCursorType.wdCursorWait;
        StringWriter sw = new StringWriter();
        Office.FileSearch fs = ThisApplication.FileSearch;
        fs.FileName = "*.doc";
        fs.LookIn = "C:\\\\";
        fs.SearchSubFolders = true;
        // Select the defaults, optional in VBA:
        fs.Execute(Office.MsoSortBy.msoSortByFileName,
            Office.MsoSortOrder.msoSortOrderAscending, true);
        foreach (String str in fs.FoundFiles)
        {
            sw.WriteLine(str);
        }
        MessageBox.Show(sw.ToString());
    }
    finally
    {
        ThisApplication.System.Cursor =
            Word.WdCursorType.wdCursorNormal;
    }
}

```

Tip In this example, you'll note that Visual Basic .NET developers needn't pass all the parameters to the **Execute** method, as they're all optional. C# developers, however, must pass every parameter. In this case, the code supplies values that match the default values. For reference-type parameters, you can pass the `Type.Missing` value, indicating to Word that it should treat the parameters as if you hadn't passed a value at all. For value-type parameters, you must specify an actual value. You can use the Word VBA help file to determine the default values for the value-type parameters.

- **Path**: When used with the **Application** object, returns the path of the current application:

```

' Visual Basic
MessageBox.Show(ThisApplication.Path)

// C#
MessageBox.Show(ThisApplication.Path);

```

Tip To return the path of the active document, use `ThisDocument.Path`.

- **Options:** Returns an **Options** object that represents application settings for Word, allowing you to set a variety of options in your application. Many, but not all, of these options are available in the Tools | Options dialog box. The following code fragment will turn on the **BackgroundSave** and **Overtyp** properties, among others. If the file is printed, any fields will be automatically updated, and hidden text and field codes will be printed.

```
' Visual Basic
' Set various application options
With ThisApplication.Options
    .BackgroundSave = True
    .Overtyp = True
    .UpdateFieldsAtPrint = True
    .PrintHiddenText = True
    .PrintFieldCodes = True
End With

// C#
// Set various application options.
Word.Options options = ThisApplication.Options;

options.BackgroundSave = true;
options.Overtyp = true;
options.UpdateFieldsAtPrint = true;
options.PrintHiddenText = true;
options.PrintFieldCodes = true;
```

- **Selection:** A read-only object that represents a selected range (or the insertion point). The **Selection** object is covered in detail later in this document.
- **UserName:** Gets or sets the user name. The following procedure displays the current user's name, sets the **UserName** property to "Dudley" and displays the new **UserName**. The code then restores the original **UserName**.

```
' Visual Basic
Friend Sub ChangeUserName()
    Dim str As String = ThisApplication.UserName
    MessageBox.Show(str)
    ' Change UserName.
    ThisApplication.UserName = "Dudley"
    MessageBox.Show(ThisApplication.UserName)
    ' Restore original UserName.
    ThisApplication.UserName = str
End Sub

// C#
public void ChangeUserName()
{
    string str = ThisApplication.UserName;
    MessageBox.Show(str);

    // Change UserName.
    ThisApplication.UserName = "Dudley";
    MessageBox.Show(ThisApplication.UserName);
    // Restore original UserName.
    ThisApplication.UserName = str;
}
```

- **Visible:** A read/write property that turns the display of the Word application itself on or off. While the **Visible** property is False, all open Word windows will be hidden, and it will appear to the user that Word has quit and all document are closed (they are still running in the background). Therefore, if you set the **Visible** property to False in your code, make sure to set it to True before your procedure ends. The following code accomplishes this in the **Finally** block of a **Try/Catch** exception handler:

```
' Visual Basic
Try
    ThisApplication.Visible = False
    ' Do work here, invisibly.

Catch ex As Exception
    ' Your exception handler here.

Finally
    ThisApplication.Visible = True
End Try

// C#
try
{
    ThisApplication.Visible = false;
    // Do whatever it is, invisibly.
}
catch (Exception ex)
{
    // Your exception handler here.
}
finally
{
    ThisApplication.Visible = true;
}
```

There are several methods of the **Application** object that you may find useful for performing actions involving the Word application. Writing code to make use of **Application** object methods is similar to working with properties. Use the following methods to perform actions on the application itself:

- **CheckSpelling:** Checks a string for spelling errors. Returns True if errors are found, and False if no errors. This is handy if you just want to spell check some text to obtain a yes/no answer to the question "Are there any spelling errors?"—It doesn't display the errors or allow you to correct them. The following code checks the string "Speling errors here" and displays False in a **MessageBox**.

```
' Visual Basic
Friend Sub SpellCheckString()
    ' Checks a specified string for spelling errors.
    Dim str As String = "Speling errors here."
    If ThisApplication.CheckSpelling(str) Then
        MsgBox.Show(String.Format("No errors in \"{0}\"", str))
    Else
        MsgBox.Show(String.Format(" \"{0}\" is spelled incorrectly", str))
    End If
End Sub

// C#
public void SpellCheckString()
{
    // Checks a specified string for spelling errors.
    string str = "Speling errors here.";

    Object CustomDictionary = Type.Missing;
    Object IgnoreUppercase = Type.Missing;
    Object MainDictionary = Type.Missing;
    Object CustomDictionary2 = Type.Missing;
    Object CustomDictionary3 = Type.Missing;
    Object CustomDictionary4 = Type.Missing;
    Object CustomDictionary5 = Type.Missing;
    Object CustomDictionary6 = Type.Missing;
    Object CustomDictionary7 = Type.Missing;
    Object CustomDictionary8 = Type.Missing;
    Object CustomDictionary9 = Type.Missing;
    Object CustomDictionary10 = Type.Missing;

    // The CheckSpelling method takes a lot of optional
    // parameters, in VBA:
    if ( ThisApplication.CheckSpelling(str, ref CustomDictionary,
        ref IgnoreUppercase, ref MainDictionary, ref CustomDictionary2,
        ref CustomDictionary3, ref CustomDictionary4,
        ref CustomDictionary5, ref CustomDictionary6,
        ref CustomDictionary7, ref CustomDictionary8,
        ref CustomDictionary9, ref CustomDictionary10) )
    {
        MsgBox.Show(String.Format("No errors in \"{0}\"", str));
    }
    else
    {
        MsgBox.Show(
            String.Format("\"{0}\" is spelled incorrectly", str));
    }
}
```

Tip This example demonstrates another example in which Visual Basic .NET developers have it much easier than those developing in C#. The **CheckSpelling** method accepts a single required string parameter, followed a large number of optional parameters. C# developers must pass a series of values by reference—in this case, a group of variables all containing the Type.Missing value. You'll find it useful, if you must call methods like **CheckSpelling** multiple times, to create a "helper" class that wraps up the method call. This class could include methods that expose only the most useful parameters for the Word method calls.

- **Help:** Displays Help dialog boxes. Specify a member of the **WdHelpType** enumeration to choose the particular dialog box, selecting from the following list:

- **WdHelp:** Displays the Microsoft Word main Help dialog box
- **WdHelpAbout:** Displays the dialog box available from the **Help | About Microsoft Word** menu item
- **WdHelpSearch:** Displays the main Help dialog box with the Answer Wizard displayed

The following line of code will display the Help About Microsoft Word dialog box:

```
' Visual Basic
Friend Sub DisplayHelpAbout()
    ThisApplication.Help(Word.WdHelpType.wdHelpAbout)
End Sub

// C#
public void DisplayHelpAbout()
{
    Object value = Word.WdHelpType.wdHelpAbout;
    ThisApplication.Help(ref value);
}
```

- **Move:** Moves the application's main window based on the required **Left** and **Top** arguments, which are both Integer values
- **Resize:** Resizes the application's main window based on the required arguments **Width** and **Height** (in points). This example moves the application to the uppermost left corner of the screen and sizes it too:

```
' Visual Basic
Friend Sub MoveAndResizeWindow()
    ' None of this will work if the window is maximized
    ' or minimized.
    ThisApplication.ActiveWindow.WindowState = _
        Word.WdWindowState.wdWindowStateNormal
```



```

Word.WdWindowState.wdWindowStateNormal

' Position at upper left corner.
ThisApplication.Move(0, 0)

' Size to 300 x 600 points.
ThisApplication.Resize(300, 600)
End Sub

// C#
public void MoveAndResizeWindow()
{
    // None of this will work if the window is
    // maximized or minimized.
    ThisApplication.ActiveWindow.WindowState =
        Word.WdWindowState.wdWindowStateNormal;

    // Position at upper left corner.
    ThisApplication.Move(0, 0);

    // Size to 300 x 600 points.
    ThisApplication.Resize(300, 600);
}

```

- **Quit:** Quits Word. You can optionally save any open documents, passing a value from the **WdSaveOptions** enumeration: **wdSaveChanges**, **wdPromptToSaveChanges**, and **wdDoNotSaveChanges**. The following fragment shows all three different ways to quit Word:

```

' Visual Basic
' Automatically save changes.
ThisApplication.Quit(Word.WdSaveOptions.wdSaveChanges)

' Prompt to save changes.
ThisApplication.Quit(Word.WdSaveOptions.wdPromptToSaveChanges)

' Quit without saving changes.
ThisApplication.Quit(Word.WdSaveOptions.wdDoNotSaveChanges)

// C#
// Automatically save changes.
Object saveChanges = Word.WdSaveOptions.wdSaveChanges;
Object originalFormat = Type.Missing;
Object routeDocument = Type.Missing;
ThisApplication.Quit(ref saveChanges,
    ref originalFormat, ref routeDocument);

// Prompt to save changes.
saveChanges = Word.WdSaveOptions.wdPromptToSaveChanges;
originalFormat = Type.Missing;
routeDocument = Type.Missing;
ThisApplication.Quit(ref saveChanges,
    ref originalFormat, ref routeDocument);

// Quit without saving changes.
saveChanges = Word.WdSaveOptions.wdDoNotSaveChanges;
originalFormat = Type.Missing;
routeDocument = Type.Missing;
ThisApplication.Quit(ref saveChanges,
    ref originalFormat, ref routeDocument);

```

Tip The **Application.Quit** method adds to the list of methods that require special handling in C#. In this case, the method requires three parameters, each passed by reference.

- **SendFax:** Launches the Fax Wizard, as shown in **Figure 8**. The user can then step through the Wizard to complete the operation.

```

' Visual Basic
Friend Sub LaunchFaxWizard()
    ThisApplication.SendFax()
End Sub

// C#
public void LaunchFaxWizard()
{
    ThisApplication.SendFax();
}

```

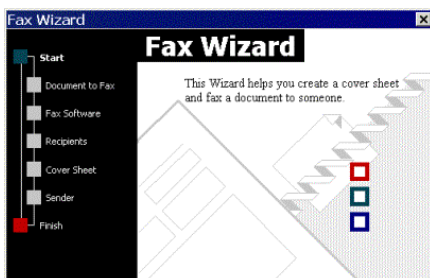




Figure 8. The Fax Wizard can be launched by the SendFax method.

Using the Built-In Dialog Boxes in Word

When working with Word, there are times when you need to display dialog boxes for user input. Although you can create your own, you might also want to take the approach of using the built-in dialog boxes in Word, which are exposed in the **Application** object's **Dialogs** collection. This allows you to access over 200 of the built-in dialog boxes in Word, represented as values in the **WdWordDialog** enumeration. To use a **Dialog** object in your code, declare it as a **Word.Dialog**:

```
' Visual Basic
Dim dlg As Word.Dialog

// C#
Word.Dialog dlg;
```

To specify the Word dialog you're interested in, assign the variable to one of the values returned from the array of available dialogs:

```
' Visual Basic
dlg = ThisApplication.Dialogs(Word.WdWordDialog.wdDialogFileNew)

// C#
Dlg = ThisApplication.Dialogs[Word.WdWordDialog.wdDialogFileNew];
```

Once you've created the **Dialog** variable, you can make use of its methods. The **Show** method displays the dialog box as though the user had selected it manually from the Word menus. The following procedure displays the File New dialog box:

```
' Visual Basic
Friend Sub DisplayFileNewDialog()
    Dim dlg As Word.Dialog
    dlg = ThisApplication.Dialogs( _
        Word.WdWordDialog.wdDialogFileNew)
    dlg.Show()
End Sub

// C#
public void DisplayFileNewDialog()
{
    Object timeOut = Type.Missing;

    Word.Dialog dlg;
    dlg = ThisApplication.Dialogs[Word.WdWordDialog.wdDialogFileNew];
    dlg.Show(ref timeOut);
}
```

Tip The **Show** method allows you to specify a TimeOut parameter, indicating the number of milliseconds to wait before automatically closing the dialog box. In Visual Basic .NET, you can ignore the parameter. In C#, pass Type.Missing by reference to indicate the default value—no timeout at all.

Another good use for the Word dialogs is to spell check a document. The following procedure launches the spell checker for the active document with the **wdDialogToolsSpellingAndGrammar** enumeration:

```
' Visual Basic
Friend Sub DisplaySpellCheckDialog()
    Dim dlg As Word.Dialog
    dlg = ThisApplication.Dialogs( _
        Word.WdWordDialog.wdDialogToolsSpellingAndGrammar)
    dlg.Show()
End Sub

// C#
public void DisplaySpellCheckDialog()
{
    Object timeOut = Type.Missing;
    Word.Dialog dlg;
    dlg = ThisApplication.Dialogs
        [Word.WdWordDialog.wdDialogToolsSpellingAndGrammar];
    dlg.Show(ref timeOut);
}
```

Word.Dialog Methods

In addition to the **Show** method, there are three additional methods that you can use with Word dialog boxes: **Display**, **Update**, and **Execute**:

- **Display**: Displays the specified built-in Word dialog box until either the user closes it or the specified amount of time has passed. It does not execute any of the actions that the dialog box normally would. You can also specify an optional timeout value. The code in the following procedure uses the Display method, supplying an optional Timeout value that will display the UserInfo dialog box for approximately three seconds. If the user does not dismiss the dialog, it will automatically close:

```
' Visual Basic
Friend Sub DisplayUserInfoDialog()
    Dim dlg As Word.Dialog
    dlg = ThisApplication.Dialogs( _
        Word.WdWordDialog.wdDialogToolsSpellingAndGrammar)
```

```
word.wdworddialog.wdDialogToolsOptionsUserInfo)
dlg.Display(3000)
End Sub

// C#
public void DisplayUserInfoDialog()
{
    Word.Dialog dlg;
    Object timeout = 3000;
    dlg = ThisApplication.Dialogs[
        Word.WdWordDialog.wdDialogToolsOptionsUserInfo];
    dlg.Display(ref timeout);
}
```

Tip Although it's tempting for C# developers to attempt to pass literal parameters as simple parameters, doing so will make your code not compile. Instead, C# developers must create an **Object** variable, place the literal value into the variable, and pass the variable by reference.

If you care about which button the user chooses in dismissing a dialog box, you can return the result of the **Display** method in an **Integer** variable so that you can branch in your code depending on the button selected. For example, a user might have edited the name of the user in the UserInfo dialog box. If the user clicks the OK button, they expect their changes to be saved, and if they click the Cancel button, they expect any edits to be abandoned.

```
' Visual Basic
Dim returnValue As Integer = dlg.Display()

// C#
Integer returnValue = dlg.Display();
```

The possible return values are displayed in **Table 1**:

Table 1. Command button return values

Value	Button Clicked
-2	The Close button
-1	The OK button
0 (zero)	The Cancel button
> 0 (zero)	A command button: 1 is the first button, 2 is the second button, and so on.

Unless you take explicit action to save changes made in this dialog box, it doesn't matter which button the user selects; all changes will be thrown away. You need to use the **Execute** method to explicitly apply changes within the dialog box.

- **Execute:** If you simply call the **Display** method and the user changes values in the dialog box, those changes won't be applied. You need to use the **Execute** method after the **Display** method to explicitly apply any changes the user made. Unlike the **Save** method that saves user changes, all changes will be discarded even if the user clicks OK. The following code calls the UserInfo dialog box using **Display**, and then the code checks the return value of the **Integer** variable. If the user clicked the OK button (returning a value of -1), the code uses the **Execute** method to apply the changes:

```
' Visual Basic
Friend Sub DisplayExecuteDialog()
    Dim dlg As Word.Dialog
    dlg = ThisApplication.Dialogs( _
        Word.WdWordDialog.wdDialogToolsOptionsUserInfo)
    ' Wait 10 seconds for results.
    Dim int As Integer = dlg.Display(10000)
    ' Did the user press OK?
    If int = -1 Then
        dlg.Execute()
    End If
End Sub

// C#
public void DisplayExecuteDialog()
{
    Word.Dialog dlg;
    dlg = ThisApplication.Dialogs[
        Word.WdWordDialog.wdDialogToolsOptionsUserInfo];

    // Wait 10 seconds for results.
    Object timeout = 10000;
    int value = dlg.Display(ref timeout);

    // Did the user press OK?
    if (value == -1)
    {
        dlg.Execute();
    }
}
```

Tip You can retrieve the Name value entered in the UserInfo dialog box using the **ThisApplication.UserName** property.

- **Update:** Use the **Update** method when you want to ensure that the dialog box is displaying the correct values. Because you can modify the contents of the dialog box using code, even after you've retrieved a reference to the dialog box, you may need to update the contents of the dialog box before displaying it. (See the next section for information on using the Word dialog boxes in hidden mode—that's when you would need this method.)

Modifying Dialog Values

Because of the way the Word dialog boxes have been designed, all the properties of the various dialogs that correspond to values of controls on the forms are available only at run time. That is, when Word loads the dialog box, it creates the various properties and adds them at run time to the appropriate objects. This type of scenario makes it difficult for developers working in a strongly typed world (as in C#, and in Visual Basic .NET with **Option Strict** set to **On**) to write code that compiles.

For example, the Page Setup dialog box (represented by the **WdWordDialog.wdDialogFilePageSetup** enumeration) provides a number of properties dealing with page setup, including **PageWidth**, **PageHeight**, and so on. You'd like to be able to write code like the following to access these properties:

```
' Visual Basic
Dim dlg As Word.Dialog
dlg = ThisApplication.Dialogs( _
    Word.WdWordDialog.wdDialogFilePageSetup)
dlg.PageWidth = 3.3
dlg.PageHeight = 6.6
```

Unfortunately, this code simply won't compile in Visual Basic .NET with **Option Strict** set **On**, or in C# at all—the **PageWidth** and **PageHeight** properties aren't defined for the **Dialog** object until run time.

You have two options for working with these properties: you can either create a Visual Basic file that includes the **Option Strict Off** setting at the top, or you can find a way to perform late binding. (You could, of course, work through the intricacies of using the System.Reflection namespace to perform the late binding yourself—that's what Visual Basic .NET does when you turn **Option Strict** off, under the covers.) The **CallByName** method, provided by the Microsoft.VisualBasic assembly, allows you to specify the name of the property with which you want to interact, as a string, along with the value for the property and the action you'd like to take. You can use this method whether you're a Visual Basic .NET or a C# developer. C# developers will, of course, need to add a reference to the Microsoft.VisualBasic assembly in order to take advantage of this technique.

Once you've referenced the assembly, you can add code like the following:

```
' Visual Basic
dlg = ThisApplication.Dialogs( _
    Word.WdWordDialog.wdDialogFilePageSetup)

CallByName(dlg, "PageWidth", CallType.Let, 3.3)
CallByName(dlg, "PageHeight", CallType.Let, 6)

// C#
using VB = Microsoft.VisualBasic;

// Then, within some procedure:
dlg = ThisApplication.Dialogs
    [Word.WdWordDialog.wdDialogFilePageSetup];

VB.Interaction.CallByName(dlg, "PageWidth",
    VB.CallType.Let, 3.3);
VB.Interaction.CallByName(dlg, "PageHeight",
    VB.CallType.Let, 6);
```

It's not a perfect solution—the code is tricky to read and write, and requires C# developers to use methods from Visual Basic (which may seem too draconian a solution, in any case), but it does provide a simple way to work with an otherwise unavailable set of properties.

The sample project includes the following procedure, which modifies the page settings for the current document, using the Page Setup dialog box. Note that this code uses the **Execute** method of the **Dialog** class, and never actually displays the dialog box—this is perfectly valid behavior, and this technique provides a simple way to set a large number of properties in one place without requiring you to dig around into multiple objects:

```
' Visual Basic
using VB = Microsoft.VisualBasic;

Public Sub HiddenPageSetupDialog()
    Dim dlg As Word.Dialog
    dlg = ThisApplication.Dialogs( _
        Word.WdWordDialog.wdDialogFilePageSetup)

    CallByName(dlg, "PageWidth", CallType.Let, ConvertToInches(3.3))
    CallByName(dlg, "PageHeight", CallType.Let, ConvertToInches(6))
    CallByName(dlg, "TopMargin", CallType.Let, ConvertToInches(0.72))
    CallByName(dlg, "BottomMargin", CallType.Let, _
        ConvertToInches(0.72))
    CallByName(dlg, "LeftMargin", CallType.Let, _
        ConvertToInches(0.66))
    CallByName(dlg, "RightMargin", CallType.Let, _
        ConvertToInches(0.66))
    CallByName(dlg, "Orientation", CallType.Let, _
        Word.WdOrientation.wdOrientPortrait)
    CallByName(dlg, "DifferentFirstPage", CallType.Let, False)
    CallByName(dlg, "HeaderDistance", CallType.Let, _
        ConvertToInches(0.28))
    ' Use the ApplyPropsTo property to determine where
    ' the property settings are applied:
    ' 0=This Section
    ' 1=This Point Forward
    ' 2=Selected Sections
    ' 3=Selected Text
    ' 4=Whole Document
    CallByName(dlg, "ApplyPropsTo", CallType.Let, 0)
    dlg.Execute()
End Sub

Private Function ConvertToInches(ByVal value As Double) As String
    Return String.Format("{0}""", value)
End Function

// C#
public void HiddenPageSetupDialog()
```

```

{
    Word.Dialog dlg;
    dlg = ThisApplication.Dialogs[
        Word.WdWordDialog.wdDialogFilePageSetup];

    VB.Interaction.CallByName(dlg, "PageWidth", VB.CallType.Let,
        ConvertToInches(3.3));
    VB.Interaction.CallByName(dlg, "PageHeight", VB.CallType.Let,
        ConvertToInches(6));
    VB.Interaction.CallByName(dlg, "TopMargin", VB.CallType.Let,
        ConvertToInches(0.72));
    VB.Interaction.CallByName(dlg, "BottomMargin", VB.CallType.Let,
        ConvertToInches(0.72));
    VB.Interaction.CallByName(dlg, "LeftMargin", VB.CallType.Let,
        ConvertToInches(0.66));
    VB.Interaction.CallByName(dlg, "RightMargin", VB.CallType.Let,
        ConvertToInches(0.66));
    VB.Interaction.CallByName(dlg, "Orientation", VB.CallType.Let,
        Word.WdOrientation.wdOrientPortrait);
    VB.Interaction.CallByName(dlg, "DifferentFirstPage",
        VB.CallType.Let, false);
    VB.Interaction.CallByName(dlg, "HeaderDistance", VB.CallType.Let,
        ConvertToInches(0.28));
    // Use the ApplyPropsTo property to determine where
    // the property settings are applied:
    // 0=This Section
    // 1=This Point Forward
    // 2=Selected Sections
    // 3=Selected Text
    // 4=Whole Document
    VB.Interaction.CallByName(dlg, "ApplyPropsTo", VB.CallType.Let,
        0);
    dlg.Execute();
}
private String ConvertToInches(double value)
{
    return String.Format("{0}\"", value);
}

```

Tip The use of measurements in Word can be confusing at times. Most measurements in Word contain values in points (1/72"), but you can always override the units by passing a string, like "3"" to indicate three inches. The sample includes the **ConvertToInches** method, which handles tacking on the necessary quote mark. The odd thing is that the **PageWidth** and **PageHeight** properties use the default units set by the user, whereas the other properties in this example require either points or a string containing a value with the unit indicator. The call to **ConvertToInches** is therefore unnecessary, in countries that use inches by default, for the first two properties in the example. Calling the method can't hurt, however.

When deciding whether or not to use the built-in dialog boxes, consider the amount of work that you are doing. If you are only setting a couple of properties in a single object, you're probably better off simply working with that object. If you need to display an interface for your users to interact with, your best bet is to use the corresponding dialog box. Consult the Word Help file for information on using the other Word built-in dialog boxes.

Tip If you want to make full use of the Word **Dialog** object, you'll need to look deeper than the included Word help file. This file barely touches on the huge set of dialog boxes provided by Word. To find more information, you'll need the WordBasic help file, from Word 95. You can find this help file on www.microsoft.com. Search for "Word 95 WordBasic Help File" to locate the correct page.

The Document Object

The bulk of your programming activity in Word will involve the **Document** object or its contents. When you work with a particular document in Word, it is known as the active document, and can be referenced by the **Application** object's **ActiveDocument** property. All Word Document objects are also members of the **Application** object's **Documents** collection, which consists of all open documents. Using the **Document** object allows you to work with a single document, and the **Documents** collection allows you to work with all open documents. The **Application** and **Document** classes share many members as it is possible to perform document operations at both the **Application** and **Document** levels.

Some common tasks you can perform involving documents include:

- Creating and opening documents
- Adding, searching and replacing text
- Printing

Document Object Collections

A document consists of characters arranged into words, with words structured into sentences. Sentences are arranged into paragraphs, which can, in turn, be arranged inside of sections. Each section contains its own headers and footers. The **Document** object has collections that map to these constructs:

- Characters
- Words
- Sentences
- Paragraphs
- Sections
- Headers/Footers

Referencing Documents

You can refer to a **Document** object as a member of the **Documents** collection by using its index value. The index value is the **Document** object's location in the **Documents** collection, which is a 1-based collection (like all the collections within Word). The following code fragment sets an object variable to refer to the first **Document** object in the **Documents** collection:

```

' Visual Basic
Dim doc As Word.Document = _
    DirectCast(ThisApplication.Documents(1), Word.Document)

// C#
Word.Document doc = (Word.Document) ThisApplication.Documents[1];

```

You can also reference a document by its name, which is usually a better choice if you want to work with a specific document. You will rarely refer to a document by using its index value in the **Documents** collection because this value can change for a given document as other documents are opened and closed. The following code fragment sets an object variable to point to the named document, "MyDoc.doc":

```
' Visual Basic
Dim doc As Word.Document = _
    DirectCast(ThisApplication.Documents("MyDoc.doc"), Word.Document)

// C#
Word.Document doc =
    (Word.Document) ThisApplication.Documents["MyDoc.doc"];
```

If you want to refer to the active document (the document that has the focus), you can use the **ActiveDocument** property of the **Application** object. You already have the property created for you in the Visual Studio .NET project, so your code will be more efficient if you use the **ThisDocument** reference when you need to refer to the document that has the focus. The following code fragment retrieves the name of the active document:

```
' Visual Basic
Dim str As String = ThisDocument.Name

// C#
String str = ThisDocument.Name;
```

Opening, Closing, and Creating New Documents

The reference to the **ThisDocument** object in your Word project gives you access to all members of the **Document** object, allowing you to work with its methods and properties, as applied to the active document. The first step in working with the **Document** object is to open an existing Word document or to create a new one.

Creating a New Word Document

When you create a new Word document, you add it to the Application's **Documents** collection of open Word documents. Consequently, the **Add** method creates a new Word document. This is the same as clicking on the **New Blank Document** button on the toolbar.

```
' Visual Basic
' Create a new document based on Normal.dot.
ThisApplication.Documents.Add()

// C#
// Create a new document based on Normal.dot.
Object template = Type.Missing;
Object newTemplate = Type.Missing;
Object documentType = Type.Missing;
Object visible = Type.Missing;

ThisApplication.Documents.Add(
    ref template, ref newTemplate, ref documentType, ref visible);
```

Tip The **Documents.Add** method accepts up to four optional parameters, indicating the template name, a new template name, the document type, and the visibility of the new document. In C#, you must pass **Type.Missing** by reference for each of these optional parameters in order to take advantage of the default value for each parameter.

The **Add** method has an optional **Template** argument to create a new document based on a template other than Normal.dot. You need to supply the fully qualified path and file name where the template can be found.

```
' Visual Basic
' Create a new document based on a custom template.
ThisApplication.Documents.Add( _
    "C:\Test\MyTemplate.Dot")

// C#
// Create a new document based on a custom template.
Object template = @"C:\Test\MyTemplate.Dot";
Object newTemplate = Type.Missing;
Object documentType = Type.Missing;
Object visible = Type.Missing;

ThisApplication.Documents.Add(
    ref template, ref newTemplate, ref documentType, ref visible);
```

This code achieves the same result as a user choosing **File | New** from the menu and choosing a template from the New Document toolbar. When you write code specifying the **Template** argument, you can ensure that all documents will be created using the specified template. Coding your own new file routine might be easier in the long run for your users, who might be confused when it comes to choosing the correct template.

Opening an Existing Document

The **Open** method opens an existing document. The basic syntax is very simple—you use the **Open** method and supply the fully qualified path and file name. There are other optional arguments that you can supply, such as a password or whether to open the document read-only, which you can find by using IntelliSense in the code window. The following code opens a document, passing only one of several optional parameters. The C# code, of course, must pass all the parameters, only supplying a real value for the **FileName** parameter:

```
' Visual Basic
ThisApplication.Documents.Open("C:\Test\MyNewDocument")

// C#
Object filename = @"C:\Test\MyNewDocument";
Object confirmConversions = Type.Missing;
Object readOnly = Type.Missing;
Object addToRecentFiles = Type.Missing;
Object passwordDocument = Type.Missing;
Object passwordTemplate = Type.Missing;
Object revert = Type.Missing;
```

```

Object writePasswordDocument = Type.Missing;
Object writePasswordTemplate = Type.Missing;
Object format = Type.Missing;
Object encoding = Type.Missing;
Object visible = Type.Missing;
Object openConflictDocument = Type.Missing;
Object openAndRepair = Type.Missing;
Object documentDirection = Type.Missing;
Object noEncodingDialog = Type.Missing;

ThisApplication.Documents.Open(ref filename,
    ref confirmConversions, ref readOnly, ref addToRecentFiles,
    ref passwordDocument, ref passwordTemplate, ref revert,
    ref writePasswordDocument, ref writePasswordTemplate,
    ref format, ref encoding, ref visible, ref openConflictDocument,
    ref openAndRepair, ref documentDirection, ref noEncodingDialog);

```

Saving Documents

There are several ways to save and close documents, depending on what you want the end result to be. The two methods you use to save and close documents are **Save** and **Close**, respectively. They have different results depending on how they are used. If applied to a **Document** object, only that document is affected. If applied to the **Documents** collection, all open documents are affected.

- **Save all Documents:** The **Save** method saves changes to all open documents when applied to the **Documents** object. There are two different ways to use it, depending on whether you want the user to be prompted to save changes or whether you want the save operation to proceed without user intervention. If you simply call the **Save** method on the **Documents** collection, the user will be prompted to save all files.

```

' Visual Basic
ThisApplication.Documents.Save()

// C#
Object noPrompt = Type.Missing;
Object originalFormat = Type.Missing;

ThisApplication.Documents.Save(ref noPrompt, ref originalFormat);

```

- The following code sets the *NoPrompt* parameter to True, and saves all open documents without user intervention.

```

' Visual Basic
ThisApplication.Documents.Save(NoPrompt:=True)

// C#
Object noPrompt = true;
Object originalFormat = Type.Missing;
ThisApplication.Documents.Save(ref noPrompt, ref originalFormat);

```

Tip The default value for *NoPrompt* is False, so if you call **Save** without specifying a *NoPrompt* value in Visual Basic .NET, or by specifying **Type.Missing** in C#, the user will be prompted to save.

- **Save a Single Document:** The **Save** method saves changes to a specified **Document** object. The following code fragment shows two ways to save the active document:

```

' Visual Basic
' Save the active document.
ThisDocument.Save()

' or
ThisApplication.ActiveDocument.Save()

// C#
// Save the active document.
ThisDocument.Save();
// or
ThisApplication.ActiveDocument.Save();

```

- If you're not sure if the document you want to save is the active document, you can refer to it by its name. This code uses the **Save** method on a named document.

```

' Visual Basic
ThisApplication.Documents("MyNewDocument.doc").Save()

// C#
Object file = "MyNewDocument.doc";
ThisApplication.Documents.get_Item(ref file).Save();

```

Tip Although Visual Basic .NET developers can retrieve items from the various collections using the standard Visual Basic syntax (calling the **Item** property, or leaving out this optional call, and supplying an index or name), C# developers generally cannot. Instead, C# developers generally must call the hidden **get_Item** method, passing an index or name by reference, as in the previous example. C# developers can directly access elements of arrays (as in the **Dialogs** array, shown previously), but for collections, you'll need to use the **get_Item** method.

- An alternate syntax would be to use the document's index number, although that isn't as reliable for two reasons. The first is that you can't be sure which document is being referred to since the index number can change, and the second is that if the referenced document hasn't been saved yet the save dialog box will appear. The following code saves the first document in the **Documents** collection:

```

' Visual Basic
ThisApplication.Documents(1).Save()

// C#
Object file = 1;

```

```
ThisApplication.Documents.get_Item(ref file).Save();
```

- **SaveAs:** The **SaveAs** method allows you to save a document under another file name. It requires that you specify the new file name, but other arguments are optional. The following procedure saves a document with a hard coded path and file name. If a file by that name already exists in that folder, it will be silently overwritten. (Note that the **SaveAs** method accepts several optional parameters, all of which must be satisfied in C#.)

```
' Visual Basic
' Save the document. In a real application,
' you'd want to test to see if the file
' already exists. This will overwrite any previously
' existing document with the specified name.
ThisDocument.SaveAs("c:\test\MyNewDocument.doc")

// C#
// Save the document. In a real application,
// you'd want to test to see if the file
// already exists. This will overwrite any previously
// existing documents.
Object fileName = @"C:\Test\MyNewDocument.doc";
Object fileFormat = Type.Missing;
Object lockComments = Type.Missing;
Object password = Type.Missing;
Object addToRecentFiles = Type.Missing;
Object writePassword = Type.Missing;
Object readOnlyRecommended = Type.Missing;
Object embedTrueTypeFonts = Type.Missing;
Object saveNativePictureFormat = Type.Missing;
Object saveFormsData = Type.Missing;
Object saveAsAOCELetter = Type.Missing;
Object encoding = Type.Missing;
Object insertLineBreaks = Type.Missing;
Object allowSubstitutions = Type.Missing;
Object lineEnding = Type.Missing;
Object addBidiMarks = Type.Missing;

ThisDocument.SaveAs(ref fileName, ref fileFormat, ref lockComments,
    ref password, ref addToRecentFiles, ref writePassword,
    ref readOnlyRecommended, ref embedTrueTypeFonts,
    ref saveNativePictureFormat, ref saveFormsData,
    ref saveAsAOCELetter, ref encoding, ref insertLineBreaks,
    ref allowSubstitutions, ref lineEnding, ref addBidiMarks);
```

Closing Documents

The **Close** method can be used to save documents as well as close them. You can close documents individually or close all at once.

- **Closing All Documents:** The **Close** method works similarly to the **Save** method when applied to the **Documents** collection. When called with no arguments, it prompts the user to save changes to any unsaved documents.

```
' Visual Basic
ThisApplication.Documents.Close()

// C#
Object saveChanges = Type.Missing;
Object originalFormat = Type.Missing;
Object routeDocument = Type.Missing;
ThisApplication.Documents.Close(ref saveChanges,
    ref originalFormat, ref routeDocument);
```

Like the **Save** method, the **Close** method accepts an optional **SaveChanges** argument that has three **WdSaveOptions** enumerations you can use: **wdDoNotSaveChanges**, **wdPromptToSaveChanges**, or **wdSaveChanges**. The following lines of code close all open documents, silently saving and discarding changes respectively:

```
' Visual Basic
' Closes all documents: saves with no prompt.
ThisApplication.Documents.Close( _
    Word.WdSaveOptions.wdSaveChanges)

' Closes all documents: does not save any changes.
ThisApplication.Documents.Close( _
    Word.WdSaveOptions.wdDoNotSaveChanges)

// C#
// Closes all documents: saves with no prompt.
Object saveChanges = Word.WdSaveOptions.wdSaveChanges;
Object originalFormat = Type.Missing;
Object routeDocument = Type.Missing;
ThisApplication.Documents.Close(ref saveChanges,
    ref originalFormat, ref routeDocument);

// Closes all documents: does not save any changes.
Object saveChanges = Word.WdSaveOptions.wdDoNotSaveChanges;
Object originalFormat = Type.Missing;
Object routeDocument = Type.Missing;
ThisApplication.Documents.Close(ref saveChanges,
    ref originalFormat, ref routeDocument);
```


Note When you call the **Application.Quit** method, Word shuts down. Closing all open documents doesn't cause Word to quit—if you close all open documents, Word will still be running and you'll still need to shut it down.

- **Close a Single Document:** The code fragments listed here close the active document without saving changes, and close MyNewDocument silently saving changes:

```
' Visual Basic
' Close the active document without saving changes.
ThisDocument.Close( _
    Word.WdSaveOptions.wdDoNotSaveChanges)

'Close MyNewDocument and save changes without prompting.
ThisApplication.Documents("MyNewDocument.doc").Close( _
    Word.WdSaveOptions.wdSaveChanges)

// C#
// Close the active document without saving changes.
Object saveChanges = Word.WdSaveOptions.wdDoNotSaveChanges;
Object originalFormat = Type.Missing;
Object routeDocument = Type.Missing;
ThisDocument.Close(ref saveChanges,
    ref originalFormat, ref routeDocument);

// Close MyNewDocument and save changes without prompting.
Object name = "MyNewDocument.doc";
saveChanges = Word.WdSaveOptions.wdSaveChanges;
originalFormat = Type.Missing;
routeDocument = Type.Missing;

Word.Document doc = ThisApplication.Documents.get_Item(ref name);
ThisDocument.Close(ref saveChanges,
    ref originalFormat, ref routeDocument);
```

Looping Through the Documents Collection

Most of the time you probably aren't going to be interested in iterating through the entire **Documents** collection: you'll want to work with an individual document. There are occasions when you want to visit each open document and conditionally perform some operation. You can refer to a Word document in the **Documents** collection by its name, by its index in the collection, or you can use a **For Each** (in Visual Basic .NET) or **foreach** (in C#) loop to iterate through the documents. Inside the loop, you can conditionally perform operations on selected files. In this example, the code walks through all open documents, and if a document hasn't been saved, saves it.

The code in the sample procedure takes the following actions:

- Loops through the collection of open documents
- Checks the **Saved** property of each document and saves the document if it hasn't been saved
- Collects the name of each saved document
- Displays the name of each saved document in a **MessageBox**, or a message indicating that no documents need saving based on the length of the string

```
' Visual Basic
Public Sub SaveUnsavedDocuments()
    ' Iterate through the Documents collection.
    Dim str As String
    Dim doc As Word.Document
    Dim sw As New StringWriter

    For Each doc In ThisApplication.Documents
        If Not doc.Saved Then
            ' Save the document.
            doc.Save()
            sw.WriteLine(doc.Name)
        End If
    Next

    str = sw.ToString()
    If str = String.Empty Then
        str = "No documents need saving."
    End If
    MessageBox.Show(str, "SaveUnsavedDocuments")
End Sub

// C#
public void SaveUnsavedDocuments()
{
    // Iterate through the Documents collection.
    string str;
    StringWriter sw = new StringWriter();

    foreach (Word.Document doc in ThisApplication.Documents)
    {
        if (!doc.Saved)
        {
            // Save the document.
            doc.Save();
            sw.WriteLine(doc.Name);
        }
    }

    str = sw.ToString();
    if (str == string.Empty)
    {
        str = "No documents need saving.";
    }
}
```

```

    str = "No documents need saving.";
}
MessageBox.Show(str, "SaveUnsavedDocuments");
}

```

The Selection Object

The **Selection** object represents the area in a Word document that is currently selected. When you perform an operation in the Word user interface, such as bolding text, you select the text and then apply the formatting. You use the Selection object the same way in your code: define the Selection and then perform the operation. You can use the **Selection** object to select, format, manipulate, and print text in your document.

The **Selection** object is always present in a document. If nothing is selected, it represents the insertion point. Therefore, it's important to know what a **Selection** object consists of before you attempt to do anything with it.

Note The **Selection** and **Range** objects have many members in common. The difference is that a **Selection** object refers to what is displayed in the user interface, whereas a **Range** object is not displayed (although it can be, by calling its **Select** method).

Tip Be wary of modifying the user's selection, using the **Selection** object. If you need to work with a portion of the document but don't want to modify the user's selection, use a specific range, paragraph, sentence, and so on.

Using the Type Property

There are various types of selections and it's important to know what, if anything, is selected. For example, if you're performing an operation on a column in a table, you'd like to ensure that the column is selected to avoid triggering a run-time error. This is easily achieved with the **Type** property of the **Selection** object. The **Type** property contains the following **WdSelectionType** enumerated values that you can use in your code to determine what is selected:

- wdSelectionBlock
- wdSelectionColumn
- wdSelectionFrame
- wdSelectionInlineShape
- wdSelectionIP
- wdSelectionNormal
- wdNoSelection
- wdSelectionRow
- wdSelectionShape

The intended purpose of most of the enumerations are obvious given their names, but some are a little more obscure. For example, **wdSelectionIP** represents the insertion point. The **wdInlineShape** value represents an image or a picture. The value **wdSelectionNormal** represents selected text, or a combination of text and other selected objects.

The code in the following procedure works with the Selection's **Type** property in order to determine the type of selection. To test it, enter and select some text within the sample document, then run the demo form. The code doesn't do much after determining the current **Selection** object's **Type** property. It simply uses a **Case** structure to store the value in a string variable to display in a **MessageBox**:

```

' Visual Basic
Friend Sub ShowSelectionType()
    Dim str As String
    Select Case ThisApplication.Selection.Type
        Case Word.WdSelectionType.wdSelectionBlock
            str = "block"
        Case Word.WdSelectionType.wdSelectionColumn
            str = "column"
        Case Word.WdSelectionType.wdSelectionFrame
            str = "frame"
        Case Word.WdSelectionType.wdSelectionInlineShape
            str = "inline shape"
        Case Word.WdSelectionType.wdSelectionIP
            str = "insertion point"
        Case Word.WdSelectionType.wdSelectionNormal
            str = "normal text"
        Case Word.WdSelectionType.wdNoSelection
            str = "no selection"
        Case Word.WdSelectionType.wdSelectionRow
            str = "row"
        Case Else
            str = "(unknown)"
    End Select
    MessageBox.Show(str, "ShowSelectionType")
End Sub

// C#
public void ShowSelectionType()
{
    string str;
    switch (ThisApplication.Selection.Type)
    {
        case Word.WdSelectionType.wdSelectionBlock:
            str = "block";
            break;
        case Word.WdSelectionType.wdSelectionColumn:
            str = "column";
            break;
        case Word.WdSelectionType.wdSelectionFrame:
            str = "frame";
            break;
        case Word.WdSelectionType.wdSelectionInlineShape:
            str = "inline shape";
            break;
        case Word.WdSelectionType.wdSelectionIP:
            str = "insertion point";
            break;
        case Word.WdSelectionType.wdSelectionNormal:
            str = "normal text";
            break;
        case Word.WdSelectionType.wdNoSelection:
            str = "no selection";
    }
}

```

```

        break;
    case Word.WdSelectionType.wdSelectionRow:
        str = "row";
        break;
    default:
        str = "(unknown)";
        break;
    }
    MessageBox.Show(str, "ShowSelectionType");
}

```

Navigating and Selecting Text

In addition to determining what has been selected, you can also use the following methods of the **Selection** object to navigate and select different ranges of text in a document. These methods mimic the actions of keys on your keyboard.

Home and End Key Methods

Using these methods also changes the selection.

HomeKey([Unit], [Extend]): Acts as if you'd pressed the HOME key on your keyboard.

EndKey([Unit], [Extend]): Acts as if you'd pressed the END key on the keyboard.

You use one of the following **wdUnits** enumerations for the **Unit** argument, which determines the range of the move:

- **WdLine**: Move to the beginning or the end of a line. This is the default value.
- **WdStory**: Move to the beginning or the end of the document.
- **WdColumn**: Move to the beginning or end of a column. Valid for tables only.
- **WdRow**: Move to the beginning or the end of a row. Valid for tables only.

You use one of the following **WdMovementType** enumerations for the **Extend** argument, which determines whether the Selection object will be an extended range or the insertion point:

- **WdMove**: Moves the selection. The end result is that the new **Selection** object consists of the insertion point. When used with **wdLine**, it moves the insertion point to the beginning or end of the line. When used with **WdStory**, it moves the insertion point to the beginning or end of the document.
- **WdExtend**: Extends the selection. The end result is that the new **Selection** object consists of a range that extends from the insertion point to the end point. If the starting point is not the insertion point, the behavior varies depending on the method used. For example, if a line is currently selected and the **HomeKey** method is called with the **wdStory** and **wdExtend** enumerations, the line will not be included in the new selection. If the **EndKey** method is called with the **wdStory** and **wdExtend** enumerations, the line will be included in the selection. This behavior mirrors the keyboard shortcuts CTRL+SHIFT+HOME and CTRL+SHIFT+END, respectively.

The following code moves the insertion point to the beginning of the document using the **HomeKey** method with the **wdStory** and the **WdMovementType wdMove** enumerations. The code then extends the selection to the end of the document using the **EndKey** with the **wdExtend** enumeration:

```

' Visual Basic
' Position the insertion point at the beginning of the document.
ThisApplication.Selection.HomeKey( _
    Word.WdUnits.wdStory, Word.WdMovementType.wdMove)

' Select from the insertion point to the end of the document.
ThisApplication.Selection.EndKey( _
    Word.WdUnits.wdStory, Word.WdMovementType.wdExtend)

// C#
// Position the insertion point at the beginning
// of the document.
Object unit = Word.WdUnits.wdStory;
Object extend = Word.WdMovementType.wdMove;
ThisApplication.Selection.HomeKey(ref unit, ref extend);

// Select from the insertion point to the end of the document.
unit = Word.WdUnits.wdStory;
extend = Word.WdMovementType.wdExtend;
ThisApplication.Selection.EndKey(ref unit, ref extend);

```

Arrow Key Methods

You can also move a selection with the following methods, each of which have a *Count* argument that determines the number of units to move for a given direction. These methods correspond to using the cursor arrow keys on your keyboard:

- **MoveLeft**([Unit], [Count], [Extend])
- **MoveRight**([Unit], [Count], [Extend])
- **MoveUp**([Unit], [Count], [Extend])
- **MoveDown**([Unit], [Count], [Extend])

The *Extend* argument takes the same two enumerations, **wdMove** and **wdExtend**. You have a different selection of **WdUnits** enumerations for the *Unit* argument for **MoveLeft** and **MoveRight**:

- **wdCharacter**: Move in character increments. This is the default value.
- **wdWord**: Move in word increments.
- **wdCell**: Move in cell increments. Valid for tables only.
- **wdSentence**: Move in sentence increments.

The following code fragment moves the insertion point to the left three characters, and then selects the three words to the right of the insertion point.

```

' Visual Basic
' Move the insertion point left 3 characters.
ThisApplication.Selection.MoveLeft( _
    Word.WdUnits.wdCharacter, 3, _
    Word.WdMovementType.wdMove)

' Select the 3 words to the right of the insertion point.
ThisApplication.Selection.MoveRight(

```

```

' Visual Basic
Word.WdUnits.wdWord, 3, _
Word.WdMovementType.wdExtend)

// C#
// Move the insertion point left 3 characters.
Object unit = Word.WdUnits.wdCharacter;
Object count = 3;
Object extend = Word.WdMovementType.wdMove;
ThisApplication.Selection.MoveLeft(ref unit, ref count,
    ref extend);

// Select the 3 words to the right of the insertion point.
unit = Word.WdUnits.wdWord;
count = 3;
extend = Word.WdMovementType.wdExtend;
ThisApplication.Selection.MoveRight(ref unit, ref count,
    ref extend);

```

The **MoveUp** and **MoveDown** methods take the following enumerations for **WdUnits**:

- **wdLine**: Moves in line increments. This is the default value.
- **wdParagraph**: Moves in paragraph increments
- **wdWindow**: Moves in window increments
- **wdScreen**: Moves in screen increments

The following code fragment moves the insertion point up one line, and then selects the three following paragraphs. What actually ends up being selected depends on where the insertion point is or whether a range of text is selected at the beginning of the operation.

```

' Visual Basic
' Move the insertion point up one line.
ThisApplication.Selection.MoveUp( _
    Word.WdUnits.wdLine, 1, Word.WdMovementType.wdMove)

' Select the following 3 paragraphs.
ThisApplication.Selection.MoveDown( _
    Word.WdUnits.wdParagraph, 3, Word.WdMovementType.wdMove)

// C#
// Move the insertion point up one line.
Object unit = Word.WdUnits.wdLine;
Object count = 1;
Object extend = Word.WdMovementType.wdMove;
ThisApplication.Selection.MoveUp(ref unit, ref count, ref extend);

// Select the following 3 paragraphs.
unit = Word.WdUnits.wdParagraph;
count = 3;
extend = Word.WdMovementType.wdMove;
ThisApplication.Selection.MoveDown(ref unit, ref count,
    ref extend);

```

The Move Method

The **Move** method collapses the specified range or selection and then moves the collapsed object by the specified number of units. The following code fragment collapses the original **Selection** object and moves three words over. The end result is an insertion point at the beginning of the third word, not the third word itself:

```

' Visual Basic
ThisApplication.Selection.Move(Word.WdUnits.wdWord, 3)

// C#
// Use the Move method to move 3 words.
Object unit = Word.WdUnits.wdWord;
Object count = 3;
ThisApplication.Selection.Move(ref unit, ref count);

```

Inserting Text

The simplest way to insert text in your document is to use the **TypeText** method of the **Selection** object. **TypeText** behaves differently depending on the user's options. The code in the following procedure declares a **Selection** object variable and turns off the overtype option if it is turned on. If the overtype option is activated, any text next to the insertion point will be overwritten:

```

' Visual Basic
Friend Sub InsertTextAtSelection()
    Dim sIn As Word.Selection = ThisApplication.Selection

    ' Make sure overtype is turned off.
    ThisApplication.Options.Overtime = False

// C#
public void InsertTextAtSelection()
{
    Word.Selection sIn = ThisApplication.Selection;

    // Make sure overtype is turned off.
    ThisApplication.Options.Overtime = false;

```

The code then tests to see if the current selection is an insertion point. If it is, the code inserts a sentence using **TypeText**, and then a paragraph mark by using the **TypeParagraph** method:

```
' Visual Basic
With sln
    ' Test to see if selection is an insertion point.
    If .Type = Word.WdSelectionType.wdSelectionIP Then
        .TypeText("Inserting at insertion point. ")
        .TypeParagraph()

// C#
// Test to see if selection an insertion point.
if (sln.Type == Word.WdSelectionType.wdSelectionIP )
{
    sln.TypeText("Inserting at insertion point. ");
    sln.TypeParagraph();
}
```

The code in the **ElseIf/else if** block tests to see if the selection is a normal selection. If it is, another If block tests to see if the **ReplaceSelection** option is turned on. If it is, the code uses the Selection's **Collapse** method to collapse the selection to an insertion point at the start of the selected block of text. The text and a paragraph mark are then inserted:

```
' Visual Basic
ElseIf .Type = Word.WdSelectionType.wdSelectionNormal Then
    ' Move to start of selection.
    If ThisApplication.Options.ReplaceSelection Then
        .Collapse(Word.WdCollapseDirection.wdCollapseStart)
    End If
    .TypeText("Inserting before a text block. ")
    .TypeParagraph()
Else
    ' Do nothing
End If
End With
End Sub

// C#
else if (sln.Type == Word.WdSelectionType.wdSelectionNormal )
{
    // Move to start of selection.
    if ( ThisApplication.Options.ReplaceSelection )
    {
        Object direction = Word.WdCollapseDirection.wdCollapseStart;
        sln.Collapse(ref direction);
    }
    sln.TypeText("Inserting before a text block. ");
    sln.TypeParagraph();
}
else
{
    // Do nothing.
}
}
```

If the selection is not an insertion point or a block of selected text, the code simply does nothing at all.

You can also use the **TypeBackspace** method of the **Selection** object, which mimics the functionality of the BACKSPACE key on your keyboard. But when it comes to inserting and manipulating text, the **Range** object offers you more control.

The Range Object

The **Range** object represents a contiguous area in a document, and you create one by defining by a starting character position and an ending character position. You are not limited to a single **Range** object; you can define multiple **Range** objects in the same document. If you define both the start and end of the range at the same location, the result will be a range that consists of an insertion point. Or you can define a range that encompasses the entire document by starting at the first character and including the last character. Note that the range also includes all non-printing characters, such as spaces, tabs, and paragraph marks.

Note The ranges you create are only in existence as long as your code is running.

The **Range** object shares many members with the **Selection** object. The main difference between the two is that the **Selection** object always returns a reference to the selection in the user interface, and the **Range** object allows you to work with text without displaying the range in the user interface.

The main advantages of using a **Range** object over a **Selection** object are:

- The **Range** object generally requires fewer lines of code to accomplish a given task.
- The **Range** object doesn't incur the overhead associated with Word having to move or change the highlighting in the active document.
- The **Range** object has greater capabilities than the **Selection** object, as you'll see in the following section.

Defining and Selecting a Range


You can define a range in a document by using the **Range** method of a **Document** object to supply a start value and an end value. The following code creates a new Range object that includes the first seven characters in the active document, including non-printing characters. It then uses the **Range** object's **Select** method to highlight the range. If you omit this line of code, the **Range** object will not be selected in the Word user interface, but you'll still be able to manipulate it programmatically.

```
' Visual Basic
Dim rng As Word.Range = ThisDocument.Range(0, 7)
rng.Select()

// C#
Object start = 0;
```

```
Object end = 7;
Word.Range rng = ThisDocument.Range(ref start, ref end);
rng.Select();
```

Figure 9 shows the results, which include the paragraph mark and a space.



em ipsum dolor sit amet, con
dignissim. Fusce auctor mauris id 1
imperdiet nec, aliquet a, leo. Phase

Figure 9. A Range object includes non-printing characters.

Counting Characters

The first character in a document is at character position 0, which represents the insertion point. The last character position is equal to the total number of characters in the document. You can determine the number of characters in a document by using the **Characters** collection's **Count** property. The following code selects the entire document and displays the number of characters in a **MessageBox**:

```
' Visual Basic
Dim rng As Word.Range = _
    ThisDocument.Range(0, ThisDocument.Characters.Count)
'Or use:
' rng = ThisDocument.Range()
rng.Select()
MessageBox.Show( _
    "Characters: " & ThisDocument.Characters.Count.ToString)

// C#
Object start = Type.Missing;
Object end = Type.Missing;

Word.Range rng = ThisDocument.Range(ref start, ref end);
rng.Select();
MessageBox.Show("Characters: " +
    ThisDocument.Characters.Count.ToString());
```

Tip In Visual Basic .NET, calling the **Range** method without any parameters returns the entire range—you needn't specify the start and end values if you simply want to work with the entire contents of the document. This doesn't hold true for C# developers, of course, because you must always pass values for all the optional parameters. In C#, you can pass `Type.Missing` for all optional parameters to assume the default values.

Setting Up Ranges

If you don't care about the number of characters and all you want to do is to select the entire document, you can use the **Document** object's **Select** method on its **Range** property:

```
' Visual Basic
ThisDocument.Range.Select()

// C#
Object start = Type.Missing;
Object end = Type.Missing;

Word.Range rng = ThisDocument.Range(ref start, ref end);
rng.Select();
```

If you want to, you can use the **Document** object's **Content** property to define a range that encompasses the document's main story—that is, the content of the document not including headers, footers, and so on:

```
' Visual Basic
Dim rng As Word.Range = ThisDocument.Content
rng.Select()

// C#
Word.Range rng = ThisDocument.Content;
```

You can also use the methods and properties of other objects to determine a range. The code in the next procedure takes the following actions to select the second sentence in the active document:

- Creates a **Range** variable
- Checks to see if there are at least two sentences in the document
- Sets the **Start** argument of the **Range** method to the start of the second sentence
- Sets the **End** argument of the **Range** method to the end of the second sentence
- Selects the range

```
' Visual Basic
Friend Sub SelectSentence()
    Dim rng As Word.Range
    With ThisDocument
        If .Sentences.Count >= 2 Then
            ' Supply a Start and End value for the Range.
            rng = .Range( _
                CType(.Sentences(2).Start, System.Object), _
                CType(.Sentences(2).End, System.Object))
            ' Select the Range.
```

```

        rng.Select()
    End If
End With
End Sub

//C#
public void SelectSentence()
{
    Word.Range rng;

    if (ThisDocument.Sentences.Count >= 2 )
    {
        // Supply a Start and end value for the Range.
        Object start = ThisDocument.Sentences[2].Start;
        Object end = ThisDocument.Sentences[2].End;
        rng = ThisDocument.Range(ref start, ref end);
        rng.Select();
    }
}
}

```

Note The parameters passed to the **Range** property are declared as System.Object, so the code must convert these values explicitly. If you're working in Visual Basic .NET with **Option Strict** set to **Off**, this won't be necessary. If you're working in C#, you must pass these values by reference, as you've already seen.

Tip Unlike the **Documents** collection, which requires C# developers to use the hidden **get_Item** method to retrieve individual elements, the **Paragraphs**, **Sentences**, and other properties return arrays. Therefore, C# developers can index into these arrays just as they would any other array.

If all you want to do is to select the second sentence, you can do so in fewer lines of code by setting the range directly to the **Sentence** object. The following code fragment is equivalent to the previous procedure listing:

```

' Visual Basic
Dim rng As Word.Range = ThisDocument.Sentences(2)
rng.Select()

// C#
Word.Range rng = ThisDocument.Sentences[2];
rng.Select();

```

Extending a Range

Once you define a **Range** object, you can extend its current range by using its **MoveStart** and **MoveEnd** methods. The **MoveStart** and **MoveEnd** methods each take the same two arguments: *Unit* and *Count*. The *Unit* argument can be one of the following **WdUnits** enumerations:

- wdCharacter
- wdWord
- wdSentence
- wdParagraph
- wdSection
- wdStory
- wdCell
- wdColumn
- wdRow
- wdTable

The *Count* argument specifies the number of the units to move. The following code defines a range consisting of the first seven characters in the document. The code then uses the **Range** object's **MoveStart** method to move the starting point of the range by seven characters. Because the end of the range is also seven characters, the result is a range consisting of the insertion point. The code then moves the ending position by seven characters using the **MoveEnd** method.

```

' Visual Basic
' Define a range of 7 characters.
Dim rng As Word.Range = _
    ThisDocument.Range(0, 7)

' Move the starting position 7 characters.
rng.MoveStart(Word.WdUnits.wdCharacter, 7)

' Move the ending position 7 characters.
rng.MoveEnd(Word.WdUnits.wdCharacter, 7)

// C#
// Define a range of 7 characters.
Object start = 0;
Object end = 7;
Word.Range rng = ThisDocument.Range(ref start, ref end);

// Move the starting position 7 characters.
Object unit = Word.WdUnits.wdCharacter;
Object count = 7;
rng.MoveStart(ref unit, ref count);

// Move the ending position 7 characters.
unit = Word.WdUnits.wdCharacter;
count = 7;
rng.MoveEnd(ref unit, ref count);

```

Figure 10 shows how the code progresses; the top line is the initial range, the second line is after the **MoveStart** method has moved the starting position by seven characters (the range is an insertion point, displayed as an I-bar), and the third line displays the characters selected after the **MoveEnd** statement moves the end of the range by seven characters.

Lorem ipsum dolor

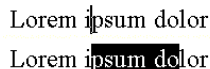


Figure 10. Using the MoveStart and MoveEnd methods to resize a range

Retrieving Start and End Characters in a Range

You can retrieve the character positions of the start and end positions of a range by retrieving the Range object's **Start** and **End** properties, as shown in the following code fragment:

```

' Visual Basic
MessageBox.Show(String.Format( _
    "Start: {0}, End: {1}", rng.Start, rng.End), _
    "Range Start and End")

// C#
MessageBox.Show(String.Format("Start: {0}, End: {1}",
    rng.Start, rng.End), "Range Start and End");

```

Using SetRange to Reset a Range

You can also use **SetRange** to resize an existing range. The following code sets an initial Range starting with the first seven characters in the document. Next, it uses **SetRange** to start the range at the second sentence and end it at the end of the fifth sentence:

```

' Visual Basic
Dim rng As Word.Range
rng = ThisDocument.Range(0, 7)
' Reset the existing Range.
rng.SetRange( _
    ThisDocument.Sentences(2).Start, _
    ThisDocument.Sentences(5).End)

// C#
Word.Range rng;
Object start = 0;
Object end = 7;
rng = ThisDocument.Range(ref start, ref end);

// Reset the existing Range.
rng.SetRange(ThisDocument.Sentences[2].Start,
    ThisDocument.Sentences[5].End);
rng.Select();

```

Formatting Text

You can also use the **Range** object to format text. The steps you need to take in your code are:

- Define the range to format.
- Apply the formatting.
- Optionally select the formatted range to display it.

The code in the sample procedure selects the first paragraph in the document and changes the font size, font name, and the alignment. It then selects the range and displays a **MessageBox** to pause before executing the next section of code, which calls the Document object's **Undo** method three times. The next code block applies the Normal Indent style and displays a **MessageBox** to pause the code. Then the code calls the **Undo** method once, and displays a **MessageBox**.

```

' Visual Basic
Friend Sub FormatRangeAndUndo()
    ' Set the Range to the first paragraph.
    Dim rng As Word.Range = _
        ThisDocument.Paragraphs(1).Range

    ' Change the formatting.
    With rng
        .Font.Size = 14
        .Font.Name = "Arial"
        .ParagraphFormat.Alignment = _
            Word.WdParagraphAlignment.wdAlignParagraphCenter
    End With
    rng.Select()
    MessageBox.Show("Formatted Range", "FormatRangeAndUndo")

    ' Undo the three previous actions.
    ThisDocument.Undo(3)
    rng.Select()
    MessageBox.Show("Undo 3 actions", "FormatRangeAndUndo")

    ' Apply the Normal Indent style.
    rng.Style = "Normal Indent"
    rng.Select()
    MessageBox.Show("Normal Indent style applied",
        "FormatRangeAndUndo")

    ' Undo a single action.
    ThisDocument.Undo()
    rng.Select()
    MessageBox.Show("Undo 1 action", "FormatRangeAndUndo")
End Sub

```



```
// C#
public void FormatRangeAndUndo()
{
    // Set the Range to the first paragraph.
    Word.Range rng = ThisDocument.Paragraphs[1].Range;

    // Change the formatting.
    rng.Font.Size = 14;
    rng.Font.Name = "Arial";
    rng.ParagraphFormat.Alignment =
    Word.WdParagraphAlignment.wdAlignParagraphCenter;
    rng.Select();
    MessageBox.Show("Formatted Range", "FormatRangeAndUndo");

    // Undo the three previous actions.
    Object times = 3;
    ThisDocument.Undo(ref times);
    rng.Select();
    MessageBox.Show("Undo 3 actions", "FormatRangeAndUndo");

    // Apply the Normal Indent style.
    Object style = "Normal Indent";
    rng.set_Style(ref style);
    rng.Select();
    MessageBox.Show("Normal Indent style applied",
        "FormatRangeAndUndo");

    // Undo a single action.
    times = 1;
    ThisDocument.Undo(ref times);
    rng.Select();
    MessageBox.Show("Undo 1 action", "FormatRangeAndUndo");
}
```

Tip Because the **Range.Style** property expects a Variant, C# developers must call the hidden **set_Style** method of the **Range** class in order to set the style. Pass either the name of the style or a Style object by reference to the **set_Style** method in order to apply a style to the range. In cases in which a read/write property has been defined as a Variant in VBA, C# developers must call the appropriate hidden accessor methods, like **set_Style** and **get_Style**. Visual Basic .NET developers can simply set or get the value of the property directly.

Inserting Text

You can use the **Text** property of a **Range** object to insert or replace text in a document. The following code fragment specifies a range that is the insertion point at the beginning of a document and inserts the text " New Text " (note the spaces) at the insertion point. The code then selects the **Range**, which now includes the inserted text. **Figure 11** shows the results after the code has run.

```
' Visual Basic
Dim str As String = "New Text"
Dim rng As Word.Range = ThisDocument.Range(0, 0)
rng.Text = str
rng.Select()

// C#
string str = " new Text ";
Object start = 0;
Object end = 0;
Word.Range rng = ThisDocument.Range(ref start, ref end);
rng.Text = str;
rng.Select();
```

New Text Lorem ipsum do
sem. Fusce dignissim. Fusce auctor
gravida vitae, imperdiet nec, aliquet
vel, egestas vel, quam. ¶

Figure 11. Inserting new text at an insertion point

Replacing Text in a Range

If your range is a selection and not the insertion point, all text in the range is replaced with the inserted text. The following code creates a **Range** object that consists of the first 12 characters in the document. The code then replaces those characters with the string.

```
' Visual Basic
rng = ThisDocument.Range(0, 12)
rng.Text = str
rng.Select()

// C#
start = 0;
end = 12;
rng = ThisDocument.Range(ref start, ref end);
rng.Text = str;
rng.Select();
```

New Text dolor sit amet, c
dignissim. Fusce auctor mauris id n
imperdiet nec, aliquet a, leo. Phasell
vel, quam. ¶

Figure 12. Inserting new text over existing text**Collapsing a Range or Selection**

If you are working with a **Range** or **Selection** object, you may want to change the selection to a prior insertion point to avoid overwriting existing text. Both the **Range** and **Selection** objects have a **Collapse** method that makes use of two **WdCollapseDirection** enumerated values:

- **WdCollapseStart**: Collapses the selection to the beginning of the selection. This is the default if you do not specify an enumeration.
- **WdCollapseEnd**: Collapses the selection to the end of the selection.

The following procedure creates a **Range** object consisting of the first paragraph in the document. It then uses the **wdCollapseStart** enumeration to collapse the range. Next, it inserts the new text and selects the range. **Figure 13** shows the results.

```
' Visual Basic
Dim str As String = " New Text "
Dim rng As Word.Range = _
    ThisDocument.Paragraphs(1).Range
rng.Collapse(Word.WdCollapseDirection.wdCollapseStart)
rng.Text = str
rng.Select()

// C#
string str = " new Text ";
Word.Range rng = ThisDocument.Paragraphs[1].Range;
Object direction = Word.WdCollapseDirection.wdCollapseStart;
rng.Collapse(ref direction);
rng.Text = str;
rng.Select();
```

New Text Lorem ipsum
sem. Fusce dignissim. Fusce auct
gravidam vitae, imperdiet nec, aliquet
vel, egestas vel, quam. ¶

Figure 13. The text you insert after collapsing a paragraph range gets inserted at the beginning of the paragraph.

If you use the **wdCollapseEnd** value, the new text gets inserted at the beginning of the following paragraph:

```
' Visual Basic
rng.Collapse(Word.WdCollapseDirection.wdCollapseEnd)

// C#
Object direction = Word.WdCollapseDirection.wdCollapseEnd;
rng.Collapse(ref direction);
```

Lorem ipsum dolor sit amet
dignissim. Fusce auctor mauris id m
imperdiet nec, aliquet a, leo. Phasell
vel, quam. ¶

New Text In hac habitasse

Figure 14. Collapsing the end of a paragraph inserts text in the next paragraph.

You might have expected that inserting the new sentence would have inserted it before the paragraph marker, but that is not the case as the original range includes the paragraph marker. The next section addresses how to work with paragraph marks to insert text safely.

Inserting Text and Dealing with Paragraph Marks

Whenever you create a **Range** object based on a paragraph, all non-printing characters are included as well. The following example procedure declares two string variables and retrieves the contents of the first and second paragraphs in the active document:

```
' Visual Basic
Friend Sub ManipulateRangeText()
    ' Retrieve contents of first and second paragraphs
    Dim str1 As String = ThisDocument.Paragraphs(1).Range.Text
    Dim str2 As String = ThisDocument.Paragraphs(2).Range.Text

// C#
public void ManipulateRangeText()
{
    // Retrieve contents of first and second paragraphs
    string str1 = ThisDocument.Paragraphs[1].Range.Text;
    string str2 = ThisDocument.Paragraphs[2].Range.Text;
```

The following code creates two **Range** variables for the first and second paragraphs and assigns the **Text** property, swapping the text between the two paragraphs. The code then selects each range in turn, pausing with **MessageBox** statements in between so that the results are displayed. **Figure 15** shows the document after the swap, with rng1 selected.

```
' Visual Basic
' Swap the paragraphs.
Dim rng1 As Word.Range = _
    ThisDocument.Paragraphs(1).Range
rng1.Text = str2

Dim rng2 As Word.Range = _
    ThisDocument.Paragraphs(2).Range
```

```

ThisDocument.Paragraphs(2).Range
rng2.Text = str1

' Pause to display the results.
rng1.Select()
MessageBox.Show(rng1.Text, "ManipulateRangeText")
rng2.Select()
MessageBox.Show(rng2.Text, "ManipulateRangeText")

// C#
// Swap the paragraphs.
Word.Range rng1 = ThisDocument.Paragraphs[1].Range;
rng1.Text = str2;

Word.Range rng2 = ThisDocument.Paragraphs[2].Range;
rng2.Text = str1;

// Pause to display the results.
rng1.Select();
MessageBox.Show(rng1.Text, "ManipulateRangeText");
rng2.Select();
MessageBox.Show(rng2.Text, "ManipulateRangeText");

```

In hac habitasse platea dictumst. Pellentesque
metus et malesuada fames ac turpis egestas. Integer tel
vitae, faucibus a, purus. Aliquam erat volutpat. Nunc
feugiat augue eu risus. ¶

Lorem ipsum dolor sit amet, consectetur adi
dignissim. Fusce auctor mauris id nulla. Sed a libero.
imperdiet nec, aliquet a, leo. Phasellus ligula quam, ul
vel, quam. ¶

Figure 15. The first and second paragraphs have been swapped, and rng1 is selected.

The next section of code adjusts rng1 using the **MoveEnd** method so that the paragraph marker is no longer a part of rng1. The code then replaces the rest of the text in the first paragraph, assigning the Range's **Text** property to a new string:

```

' Visual Basic
rng1.MoveEnd(Word.WdUnits.wdCharacter, -1)
' Write new text for paragraph 1.
rng1.Text = "New content for paragraph 1."

// C#
Object unit = Word.WdUnits.wdCharacter;
Object count = -1;
rng1.MoveEnd(ref unit, ref count);
// Write new text for paragraph 1.
rng1.Text = "new content for paragraph 1.";

```

The following section of code simply replaces the text in rng2, including the paragraph mark:

```

' Visual Basic
rng2.Text = "New content for paragraph 2."

// C#
rng2.Text = "new content for paragraph 2.";

```

The code then selects rng1 and pauses to display the results in a **MessageBox**, and then does the same with rng2. Because rng1 was redefined to exclude the paragraph mark, the original formatting of the paragraph is preserved. A sentence was inserted over the paragraph mark in rng2, which obliterated it as a paragraph. **Figure 16** shows rng2 highlighted; it has been merged into what was formerly the third paragraph and no longer exists as a paragraph on its own.

```

' Visual Basic
' Pause to display the results.
rng1.Select()
MessageBox.Show(rng1.Text, "ManipulateRangeText")
rng2.Select()
MessageBox.Show(rng2.Text, "ManipulateRangeText")

// C#
// Pause to display the results.
rng1.Select();
MessageBox.Show(rng1.Text, "ManipulateRangeText");
rng2.Select();
MessageBox.Show(rng2.Text, "ManipulateRangeText");

```

New content for paragraph 1.¶

New content for paragraph 2.Proin ac erat. Maecenas
ntesque tincidunt. Nunc justo est, eleifend ut, sollicitudin

Figure 16. The new content inserted in rng2 overwrote the paragraph mark.

Because the original contents of both ranges were saved as **String** variables, it's not too hard to restore the document to its original condition with the two paragraphs in their original order. The next line of code redefines

because the original contents of both ranges were saved as string variables, it's not too hard to restore the document to its original condition with the two paragraphs in their original order. The next line of code readjusts rng1 to include the paragraph mark by using the **MoveEnd** method to move the mark by one character position:

```
' Visual Basic
rng1.MoveEnd(Word.WdUnits.wdCharacter, 1)

// C#
unit = Word.WdUnits.wdCharacter;
count = 1;
rng1.MoveEnd(ref unit, ref count);
```

The code then deletes rng2 entirely. This will restore the third paragraph to its original position.

```
' Visual Basic
rng2.Delete()

// C#
// Note that in C#, you must specify
// both parameters--it's up to you
// to calculate the length of the range.
unit = Word.WdUnits.wdCharacter;
count = rng2.Characters.Count;
rng2.Delete(ref unit, ref count);
```

The code then restores the original paragraph text in rng1:

```
' Visual Basic
rng1.Text = str1

// C#
rng1.Text = str1;
```

The last section of code uses the Range object's **InsertAfter** method to insert the original second paragraph's content after rng1, and then selects rng1.

Figure 17 displays the three paragraphs, with rng1 selected. Note that the second paragraph is now included in rng1, which has been extended to include the inserted text.

```
' Visual Basic
rng1.InsertAfter(str2)
rng1.Select()
End Sub

// C#
rng1.InsertAfter(str2);
rng1.Select();
```

```
}

    Lorem ipsum dolor sit amet, conse-
    dignissim. Fusce auctor mauris id nulla. Se-
    imperdiet nec, aliquet a, leo. Phasellus ligu-
    vel, quam. ¶
    In hac habitasse platea dictumst. P-
    netus et malesuada fames ac turpis egestas.
    vitae, faucibus a, purus. Aliquam erat volut-
    feugiat augue eu risus. ¶
    Proin ac erat. Maecenas eget nunc
```

Figure 17. The InsertAfter method extends the Range object to include the inserted text.

The Bookmark Object

The **Bookmark** object is similar to the **Range** and **Selection** objects in that it represents a contiguous area in a document, with both a starting position and an ending position. You use bookmarks to mark a location in a document, or as a container for text in a document. A **Bookmark** object can consist of the insertion point, or be as large as the entire document. You can also define multiple bookmarks in a document. You can think of a Bookmark as a named location in the document that is saved with the document.

Creating a Bookmark

The **Bookmarks** collection exists as a member of the **Document**, **Range**, and **Selection** objects. The following sample procedure shows how to create bookmarks in a document and use them for inserting text. The code takes the following actions:

- Declares a **Range** and two **Bookmark** variables and sets the **ShowBookmarks** property to True. Setting this property causes a Bookmark set as an insertion point to appear as a gray I-bar, and a **Bookmark** set to a range of text to appear as gray brackets surrounding the bookmarked text.

```
' Visual Basic
Friend Sub CreateBookmarks()
    Dim rng As Word.Range
    Dim bookMk1 As Word.Bookmark
    Dim bookMk2 As Word.Bookmark

    ' Display Bookmarks.
```

```

        ThisDocument.ActiveWindow.View.ShowBookmarks = True
// C#
public void CreateBookmarks()
{
    Word.Range rng;
    Word.Bookmark bookMk1;
    Word.Bookmark bookMk2;

    // Display Bookmarks.
    ThisDocument.ActiveWindow.View.ShowBookmarks = true;

```

- Defines a **Range** object as the first insertion point at the beginning of the document

```

' Visual Basic
rng = ThisDocument.Range(0, 0)

// C#
Object start = 0;
Object end = 0;
rng = ThisDocument.Range(ref start, ref end);

```

- Adds a Bookmark named bookMk1 consisting of the **Range** object and displays a **MessageBox** to halt the execution of the code. At this point, you'll see the Bookmark displayed as a faint I-bar to the left of the start of the paragraph, as shown in **Figure 18**.

```

' Visual Basic
bookMk1 = ThisDocument.Bookmarks.Add( _
    "bookMk1", DirectCast(rng, Word.Range))

' Display the bookmark.
MessageBox.Show("bookMk1 Text: " & bookMk1.Range.Text, _
    "CreateBookmarks")

// C#
Object range = rng;
bookMk1 = ThisDocument.Bookmarks.Add("bookMk1", ref range);

// Display the bookmark.
MessageBox.Show("bookMk1 Text: " + bookMk1.Range.Text,
    "CreateBookmarks");

```

Figure 18. A Bookmark defined as an insertion point is displayed as an I-bar.

- Uses the **Range** method's **InsertBefore** method to insert text before the Bookmark and pauses the code with a **MessageBox**. **Figure 19** displays the Bookmark and the inserted text. Note that the text is inserted after the Bookmark, and in fact is now included in the bookmark.

```

' Visual Basic
rng.InsertBefore("***InsertBefore bookMk1**")

' Show Bookmark code.
MessageBox.Show("bookMk1 Text: " & bookMk1.Range.Text, _
    "CreateBookmarks")

// C#
rng.InsertBefore("***InsertBefore bookMk1**");

// Show Bookmark text.
MessageBox.Show("bookMk1 Text: " + bookMk1.Range.Text,
    "CreateBookmarks");

```

Figure 19. The inserted text is included in the bookmark.

- Uses the **Range** method's **InsertAfter** method to insert text after the Bookmark. Note that the inserted text appears directly after the text that was inserted before the Bookmark, as shown in **Figure 20**.

```

' Visual Basic
rng.InsertAfter("***InsertAfter bookMk1**")
MessageBox.Show("bookMk1 Text: " & bookMk1.Range.Text, _
    "CreateBookmarks")

// C#
rng.InsertAfter("***InsertAfter bookMk1**");
MessageBox.Show("bookMk1 Text: " + bookMk1.Range.Text,
    "CreateBookmarks");

```

consequatur adipiscing elit. Curabitur enim. Fusce augue enim. Fusce augue

Figure 20. Inserting text after the Bookmark.

- Resets the **Range** object to point to the second paragraph in the document and creates a new **Bookmark** object named bookMk2 on the second paragraph. When the code pauses to display the **MessageBox**, you can see the Bookmark brackets surrounding the second paragraph, as shown in **Figure 21**.

```
' Visual Basic
rng = ThisDocument.Paragraphs(2).Range

' Create new Bookmark on the second paragraph.
bookMk2 = ThisDocument.Bookmarks.Add( _
    "bookMk2", DirectCast(rng, Word.Range))
MessageBox.Show("bookMk2 Text: " & bookMk2.Range.Text, _
    "bookMk2 set")

// C#
rng = ThisDocument.Paragraphs[2].Range;

// Create new Bookmark on paragraph 2.
range = rng;
bookMk2 = ThisDocument.Bookmarks.Add("bookMk2", ref range);
MessageBox.Show("bookMk2 Text: " + bookMk2.Range.Text,
    "bookMk2 set");
```

In hac habitasse platea d
t malesuada fames ac turp
aucibus a, purus. Aliquam
augue eu risus. ¶
Proin ac erat. Maecenas

Figure 21. A Bookmark encompassing a range of text is displayed as open and closed square brackets.

- Uses **InsertBefore** to insert text before the Bookmark. **Figure 22** shows that the inserted text is now included in the Bookmark.

```
' Visual Basic
rng.InsertBefore("**InsertBefore bookMk2**")
MessageBox.Show("bookMk2 Text: " & bookMk2.Range.Text, _
    "InsertBefore bookMk2")

// C#
rng.InsertBefore("**InsertBefore bookMk2**");
MessageBox.Show("bookMk2 Text: " + bookMk2.Range.Text,
    "InsertBefore bookMk2");
```

InsertBefore bookMk2In hac habitasse plat
t morbi tristique senectus et netus et malesuada fa
netus, mattis non, lobortis vitae, faucibus a, purus. ¶
ante eu quam. Vivamus feugiat augue eu risus. ¶
Proin ac erat. Maecenas eget nunc in purus pelle

Figure 22. When the Bookmark is a range of text, the InsertBefore method adds it to the Bookmark.

- Uses **InsertAfter** to insert text after the Bookmark. Note that this text is inserted outside of the Bookmark, and is not included in the Bookmark. **Figure 23** shows the document after the Bookmark's **Select** method has run.

```
' Visual Basic
' Insert text after.
rng.InsertAfter("**InsertAfter bookMk2**")
MessageBox.Show("bookMk2 Text: " & bookMk2.Range.Text, _
    "InsertAfter bookMk2")

bookMk2.Select()
MessageBox.Show("bookMk2.Select()", "CreateBookmarks")

// C#
rng.InsertAfter("**InsertAfter bookMk2**");
MessageBox.Show("bookMk2 Text: " + bookMk2.Range.Text,
    "InsertAfter bookMk2");

bookMk2.Select();
MessageBox.Show("bookMk2.Select()", "CreateBookmarks");
```

InsertBefore bookMk2In hac habitasse platea
t morbi tristique senectus et netus et malesuada fam
netus, mattis non, lobortis vitae, faucibus a, purus. ¶
ante eu quam. Vivamus feugiat augue eu risus. ¶
InsertAfter bookMk2Proin ac erat. Maecenas

Figure 23. Text inserted after a Bookmark is not included in the Bookmark.

The Bookmarks Collection

The **Bookmarks** collection contains all of the bookmarks in a document. In addition, bookmarks can exist in other sections of the document, such as headers and footers. You can visit each **Bookmark** object and retrieve its properties. The following procedure iterates through the **Bookmarks** collection and displays the name of each Bookmark in the document and its **Range.Text** property using the **MessageBox.Show** method:

```

' Visual Basic
Friend Sub ListBookmarks()
    Dim sw As New StringWriter
    Dim bmrk As Word.Bookmark

    For Each bmrk In ThisDocument.Bookmarks
        sw.WriteLine("Name: {0}, Contents: {1}", _
            bmrk.Name, bmrk.Range.Text)
    Next
    MessageBox.Show(sw.ToString(), "Bookmarks and Contents")
End Sub

// C#
public void ListBookmarks()
{
    StringWriter sw = new StringWriter();

    foreach (Word.Bookmark bmrk in ThisDocument.Bookmarks)
    {
        sw.WriteLine("Name: {0}, Contents: {1}",
            bmrk.Name, bmrk.Range.Text);
    }
    MessageBox.Show(sw.ToString(), "Bookmarks and Contents");
}

```

Updating the Bookmark Text Property

Updating a bookmark's text is easy—you simply assign a value to the **Range.Text** property of the bookmark. Doing so, however, deletes the entire bookmark. There is no easy way to insert text into a placeholder bookmark so that you can retrieve the text at a later time. One option is to replace the bookmark with the inserted text and delete the bookmark. You then re-create the bookmark around the inserted text. The following procedure demonstrates this technique, with results shown in **Figure 24**.

```

' Visual Basic
Friend Sub BookmarkText()
    ' Create a bookmark on the first paragraph.
    With ThisDocument
        .Bookmarks.Add("bkMark", _
            .Paragraphs(1).Range)

        ' Create Range on Bookmark object.
        Dim rng As Word.Range = _
            .Bookmarks("bkMark").Range

        ' Replace Range text (this deletes the Bookmark).
        rng.Text = "New Bookmark Text."

        ' Re-create the Bookmark.
        .Bookmarks.Add( _
            "bkMark", DirectCast(rng, Word.Range))

        ' Display Bookmark.
        .ActiveWindow.View.ShowBookmarks = True
        MessageBox.Show(.Bookmarks("bkMark").Range.Text, _
            "BookmarkReplaceText")
    End With
End Sub

// C#
public void BookmarkText()
{
    // Create a bookmark on the first paragraph.
    Object range = ThisDocument.Paragraphs[1].Range;
    ThisDocument.Bookmarks.Add("bkMark", ref range);

    // Create Range on Bookmark object.
    Object name = "bkMark";
    Word.Range rng = ThisDocument.Bookmarks.get_Item(ref name).Range;

    // Replace Range text (this deletes the Bookmark).
    rng.Text = "new Bookmark Text.";

    // Recreate the Bookmark.
    range = rng;
    ThisDocument.Bookmarks.Add("bkMark", ref range);

    // Display Bookmark.
    ThisDocument.ActiveWindow.View.ShowBookmarks = true;
    name = "bkMark";
    MessageBox.Show(ThisDocument.Bookmarks.
        get_Item(ref name).Range.Text, "BookmarkReplaceText");
}

```

New Bookmark Text. In h
fames ac turpis egestas. In
ante eu quam. Vivamus fe
Proin ac erat. Maecenas e
sapien. Donec risus. Lore

Figure 24. The original Bookmark is replaced with new text.

```
' Visual Basic
Friend Sub ReplaceBookmarkText( _
    ByVal BookmarkName As String, _
    ByVal NewText As String)
    If ThisDocument.Bookmarks.Exists(BookmarkName) Then
        Dim rng As Word.Range = _
            ThisDocument.Bookmarks(BookmarkName).Range
        rng.Text = NewText
        ThisDocument.Bookmarks.Add( _
            BookmarkName, DirectCast(rng, Word.Range))
    End If
End Sub

// C#
public void ReplaceBookmarkText(string BookmarkName,
    string NewText)
{
    if (ThisDocument.Bookmarks.Exists(BookmarkName))
    {
        Object name = BookmarkName;
        Word.Range rng = ThisDocument.Bookmarks.
            get_Item(ref name).Range;
        rng.Text = NewText;
        Object range = rng;
        ThisDocument.Bookmarks.Add(BookmarkName, ref range);
    }
}
```

```
' Visual Basic
ReplacBookmarkText("FirstNameBookmark", "Joe")

// C#
ReplaceBookmarkText("FirstNameBookmark", "Joe");
```

When you edit a document in the Word user interface, you probably make extensive use of the **Find** and **Replace** commands on the Edit menu. The dialog boxes displayed let you specify search criteria for the text you want to locate. The Replace command is an extension of the Find command, allowing you to replace the searched text.

Finding Text with a Selection Object

```

Visual Basic
Public Sub FindInSelection()
    ' Move selection to beginning of doc.
    ThisApplication.Selection.HomeKey( _
        Word.WdUnits.wdStory, Word.WdMovementType.wdMove)

    Dim strFind As String = "dolor"
    Dim fnd As Word.Find = ThisApplication.Selection.Find
    fnd.ClearFormatting()
    fnd.Text = strFind
    If fnd.Execute() Then
        MessageBox.Show("Text found.")
    Else
        MessageBox.Show("Text not found.")
    End If
End Sub

// C#
public void FindInSelection()
{
    // Move selection to beginning of doc.
    Object unit = Word.WdUnits.wdStory;
    Object extend = Word.WdMovementType.wdMove;
    ThisApplication.Selection.HomeKey(ref unit, ref extend);

    Word.Find fnd = ThisApplication.Selection.Find;
    fnd.ClearFormatting();

    Object findText = "dolor";
    Object matchCase = Type.Missing;
    Object matchWholeWord = Type.Missing;

```



```

Object matchWildcards = Type.Missing;
Object matchSoundsLike = Type.Missing;
Object matchAllWordForms = Type.Missing;
Object forward = Type.Missing;
Object wrap = Type.Missing;
Object format = Type.Missing;
Object replaceWith = Type.Missing;
Object replace = Type.Missing;
Object matchKashida = Type.Missing;
Object matchDiacritics = Type.Missing;
Object matchAleFHamza = Type.Missing;
Object matchControl = Type.Missing;

if (fnd.Execute(ref findText, ref matchCase, ref matchWholeWord,
    ref matchWildcards, ref matchSoundsLike, ref matchAllWordForms,
    ref forward, ref wrap, ref format, ref replaceWith,
    ref replace, ref matchKashida, ref matchDiacritics,
    ref matchAleFHamza, ref matchControl))
{
    MessageBox.Show("Text found.", "FindInSelection");
}
else
{
    MessageBox.Show("Text not found.", "FindInSelection");
}
}

```

Lorem ipsum dolor
 ssim. Fusce auctor mau
 diet nec, aliquet a, leo.
 uam. ¶

Figure 25. Found text is automatically selected when searching with a Selection object.

Tip Find criteria are cumulative, which means that criteria are added to previous search criteria. You should get in the habit of clearing formatting from previous searches by using the **ClearFormatting** method prior to each search.

Setting Find Options

There are two different ways to set search options: by setting individual properties of the **Find** object, or by using arguments of the **Execute** method. The following procedure illustrates both forms. The first code block sets properties of the **Find** object and the second code block uses arguments of the **Execute** object. The code in both code blocks performs the identical search—only the syntax is different. Because you're unlikely to use many of the parameters required by the **Execute** method, and because you can specify many of the values as properties of the **Find** object, this is a perfect place for C# developers to create "wrapper" methods, hiding the intricacies of calling the **Find.Execute** method. This isn't required for Visual Basic developers, but the following example shows off a useful technique for C# developers:

```

' Visual Basic
Public Sub CriteriaSpecify()
    ' Use Find properties to specify search criteria.
    With ThisApplication.Selection.Find
        .ClearFormatting()
        .Forward = True
        .Wrap = Word.WdFindWrap.wdFindContinue
        .Text = "ipsum"
        .Execute()
    End With

    ' Use Execute method arguments to specify search criteria.
    With ThisApplication.Selection.Find
        .ClearFormatting()
        .Execute(FindText:="dolor", _
            Forward:=True, Wrap:=Word.WdFindWrap.wdFindContinue)
    End With
End Sub

// C#
public void CriteriaSpecify()
{
    // Use Find properties to specify search criteria.
    Word.Find fnd = ThisApplication.Selection.Find;

    fnd.ClearFormatting();
    fnd.Forward = true;
    fnd.Wrap = Word.WdFindWrap.wdFindContinue;
    fnd.Text = "ipsum";

    ExecuteFind(fnd);

    // Use Execute method arguments to specify search criteria.
    fnd = ThisApplication.Selection.Find;
    fnd.ClearFormatting();

    Object findText = "dolor";
    Object wrap = Word.WdFindWrap.wdFindContinue;
    Object forward = true;
    ExecuteFind(fnd, wrap, forward);
}

private Boolean ExecuteFind(Word.Find find)
{
    return ExecuteFind(find, Type.Missing, Type.Missing);
}

```

```

private Boolean ExecuteFind(
    Word.Find find, Object wrapFind, Object forwardFind)
{
    // Simple wrapper around Find.Execute:
    Object findText = Type.Missing;
    Object matchCase = Type.Missing;
    Object matchWholeWord = Type.Missing;
    Object matchWildcards = Type.Missing;
    Object matchSoundsLike = Type.Missing;
    Object matchAllWordForms = Type.Missing;
    Object forward = forwardFind;
    Object wrap = wrapFind;
    Object format = Type.Missing;
    Object replaceWith = Type.Missing;
    Object replace = Type.Missing;
    Object matchKashida = Type.Missing;
    Object matchDiacritics = Type.Missing;
    Object matchAlefHamza = Type.Missing;
    Object matchControl = Type.Missing;

    return find.Execute(ref findText, ref matchCase,
        ref matchWholeWord, ref matchWildcards, ref matchSoundsLike,
        ref matchAllWordForms, ref forward, ref wrap, ref format,
        ref replaceWith, ref replace, ref matchKashida,
        ref matchDiacritics, ref matchAlefHamza, ref matchControl);
}

```

Finding Text with a Range Object

Finding text using a **Range** object allows you to search for text without displaying anything in the user interface. The **Find** method returns a **Boolean** value indicating its results. This method also redefines the Range object to match the search criteria if the text is found. That is, if the **Find** method finds a match, its range is moved to the location of the match.

The following procedure defines a **Range** object consisting of the second paragraph in the document. It then uses the **Find** method, first clearing any existing formatting options, and then searches for the string "faucibus". The code displays the results of the search using the **MessageBox.Show** method, and selects the **Range** to make it visible. If the search fails, the second paragraph is selected; if it succeeds, the search criteria is selected and displayed, as shown in **Figure 26**. The C# version of this example uses the **ExecuteFind** wrapper method discussed previously.

```

' Visual Basic
Public Sub FindInRange()
    ' Set the second paragraph as the search range.
    Dim rng As Word.Range = _
        ThisDocument.Paragraphs(2).Range
    Dim fnd As Word.Find = rng.Find

    ' Clear existing formatting.
    fnd.ClearFormatting()
    ' Execute the search
    fnd.Text = "faucibus"
    If fnd.Execute() Then
        MessageBox.Show("Text found.", "FindInRange")
    Else
        MessageBox.Show("Text not found.", "FindInRange")
    End If

    ' The word faucibus will be displayed if the
    ' search succeeds; paragraph 2 will be displayed
    ' if the search fails.
    rng.Select()
End Sub

// C#
public void FindInRange()
{
    // Set the second paragraph as the search range.
    Word.Range rng = ThisDocument.Paragraphs[2].Range;
    Word.Find fnd = rng.Find;

    // Clear existing formatting.
    fnd.ClearFormatting();

    // Execute the search.
    fnd.Text = "faucibus";
    if (ExecuteFind(fnd))
    {
        MessageBox.Show("Text found.", "FindInRange");
    }
    else
    {
        MessageBox.Show("Text not found.", "FindInRange");
    }

    // The word faucibus will be displayed if the
    // search succeeds; paragraph 2 will be displayed
    // if the search fails.
    rng.Select();
}

```

utens et maresuada iames a
vitae, **faucibus** a, purus. Ali
feugiat augue eu risus. ¶

Figure 26. If the Range method's Find succeeds, the range will be redefined to contain the search criteria.

Looping through Found Items

The **Find** method also has a **Found** property, which returns True whenever a searched-for item is found. You can make use of this in your code, as shown in the sample procedure. The code uses a **Range** object to search for all occurrences of the string "lorem" in the active document, changes the font color and bold properties for each match. It uses the **Found** property in a loop, and increments a counter each time the string is found. The code then displays the number of times the string was found in a **MessageBox**. The C# version of this demonstration uses the **ExecuteFind** method discussed previously.

```
' Visual Basic
Public Sub FindInLoopAndFormat()
    Dim intFound As Integer
    Dim rngDoc As Word.Range = ThisDocument.Range
    Dim fnd As Word.Find = rngDoc.Find

    ' Find all instances of the word "lorem" and bold each.
    fnd.ClearFormatting()
    fnd.Forward = True
    fnd.Text = "lorem"
    fnd.Execute()
    Do While fnd.Found
        ' Set the new font weight and color.
        ' Note that each "match" resets
        ' the searching range to be the found text.
        rngDoc.Font.Color = Word.WdColor.wdColorRed
        rngDoc.Font.Bold = 600
        intFound += 1
        fnd.Execute()
    Loop
    MessageBox.Show( _
        String.Format("lorem found {0} times.", intFound), _
        "FindInLoopAndFormat")
End Sub

// C#
public void FindInLoopAndFormat()
{
    int intFound = 0;

    Object start = 0;
    Object end = ThisDocument.Characters.Count;
    Word.Range rngDoc = ThisDocument.Range(ref start, ref end);
    Word.Find fnd = rngDoc.Find;

    // Find all instances of the word "lorem" and bold each.
    fnd.ClearFormatting();
    fnd.Forward = true;
    fnd.Text = "lorem";
    ExecuteFind(fnd);
    while (fnd.Found)
    {
        // Set the new font weight and color.
        // Note that each "match" resets
        // the searching range to be the found text.
        rngDoc.Font.Color = Word.WdColor.wdColorRed;
        rngDoc.Font.Bold = 600;
        intFound++;
        ExecuteFind(fnd);
    }
    MessageBox.Show(
        String.Format("lorem found {0} times.", intFound),
        "FindInLoopAndFormat");
}
```

Tip It may seem odd that the range, **rngDoc**, started out referring to the entire document's contents, but ends up referring to each match within the search. This is by design—when you call the **Find** method of a **Range** variable, Word always updates the **Range** variable to refer to the found text. If you must retain a reference to the original range, you'll need to keep a second variable that maintains the original information. In this case, because the original range was the entire document, it's easy to get that range reference back later, and there's no need to retain the information in a variable.

Replacing Text

There are several ways to search and replace text in code. A typical scenario uses the **Find** object to loop through a document looking for specific text, formatting, or style. If you want to replace any of the items found, you use the **Find** object's **Replacement** property. Both the **Find** object and the **Replacement** object provide a **ClearFormatting** method. When you are performing a find and replace operation, you must use the **ClearFormatting** method of both objects. If you only use it on the Find part of the replace operation, it's likely that you may end up replacing the text with unanticipated options.

You then use the **Execute** method to replace each found item. The **Execute** method has a **WdReplace** enumeration that consists of three additional values:

- **wdReplaceAll**: replaces all found items
- **wdReplaceNone**: replaces none of the found items
- **wdReplaceOne**: replaces the first found item

The code in the following procedure searches and replaces all of the occurrences of the string "Lorum" with the string "Forum" in the selection. The C# version of this example uses the **ExecuteReplace** helper method, modeled after the **ExecuteFind** method discussed previously:

```
' Visual Basic
Friend Sub SearchAndReplace()
    With ThisApplication.Selection.Find
        .ClearFormatting()
        .Text = "Lorum"
```

```

        .Text = "Lorem"
        With .Replacement
            .ClearFormatting()
            .Text = "Forum"
        End With
        .Execute(Replace:=Word.WdReplace.wdReplaceAll)
    End With
End Sub

// C#
public void SearchAndReplace()
{
    // Move selection to beginning of document.
    Object unit = Word.WdUnits.wdStory;
    Object extend = Word.WdMovementType.wdMove;
    ThisApplication.Selection.HomeKey(ref unit, ref extend);

    Word.Find fnd = ThisApplication.Selection.Find;
    fnd.ClearFormatting();
    fnd.Text = "Lorem";
    fnd.Replacement.ClearFormatting();
    fnd.Replacement.Text = "Forum";
    ExecuteReplace(fnd);
}

private Boolean ExecuteReplace(Word.Find find)
{
    return ExecuteReplace(find, Word.WdReplace.wdReplaceAll);
}

private Boolean ExecuteReplace(Word.Find find,
    Object replaceOption)
{
    // Simple wrapper around Find.Execute:
    Object findText = Type.Missing;
    Object matchCase = Type.Missing;
    Object matchWholeWord = Type.Missing;
    Object matchWildcards = Type.Missing;
    Object matchSoundsLike = Type.Missing;
    Object matchAllWordForms = Type.Missing;
    Object forward = Type.Missing;
    Object wrap = Type.Missing;
    Object format = Type.Missing;
    Object replaceWith = Type.Missing;
    Object replace = replaceOption;
    Object matchKashida = Type.Missing;
    Object matchDiacritics = Type.Missing;
    Object matchAlefHamza = Type.Missing;
    Object matchControl = Type.Missing;

    return find.Execute(ref findText, ref matchCase,
        ref matchWholeWord, ref matchWildcards, ref matchSoundsLike,
        ref matchAllWordForms, ref forward, ref wrap, ref format,
        ref replaceWith, ref replace, ref matchKashida,
        ref matchDiacritics, ref matchAlefHamza, ref matchControl);
}

```

Restoring the User's Selection After a Search

If you search and replace text in a document, you may want to restore the user's original selection after the search is completed. The code in the sample procedure makes use of two **Range** objects: one to store the current Selection, and one to set to the entire document to use as a search range. The search and replace operation is then performed and the user's original selection restored. The C# version of this example uses the **ExecuteReplace** helper method shown previously:

```

' Visual Basic
Friend Sub ReplaceAndRestoreSelection()
    ' Save user's original selection.
    Dim rngStart As Word.Range = _
        ThisApplication.Selection.Range

    ' Define search range of entire document.
    Dim rngSearch As Word.Range = ThisDocument.Range

    With rngSearch.Find
        .ClearFormatting()
        .Text = "vel"
        With .Replacement
            .ClearFormatting()
            .Text = "VELLO"
        End With
        .Execute(Replace:=Word.WdReplace.wdReplaceAll)
    End With

    ' Restore user's original selection.
    rngStart.Select()
End Sub

// C#
public void ReplaceAndRestoreSelection()
{
    // Save user's original selection.
    Word.Range rngStart = ThisApplication.Selection.Range;
    Word.Range rngSearch = ThisDocument.Range;

    rngSearch.Find.ClearFormatting();
    rngSearch.Find.Text = "vel";
    rngSearch.Find.Replacement.ClearFormatting();
    rngSearch.Find.Replacement.Text = "VELLO";
    rngSearch.Find.Execute(Replace:=Word.WdReplace.wdReplaceAll);

    rngStart.Select();
}

```

```

word.Range rngStart = thisApplication.Selection.Range;

// Define search range of entire document.
Object start = 0;
Object end = ThisDocument.Characters.Count;
Word.Range rngSearch = ThisDocument.Range(ref start, ref end);

Word.Find fnd = rngSearch.Find;
fnd.ClearFormatting();
fnd.Text = "vel";
fnd.Replacement.ClearFormatting();
fnd.Replacement.Text = "VELLO";
ExecuteReplace(fnd);

// Restore user's original selection.
rngStart.Select();
}

```

Printing

Word possesses a rich set of built-in functionality when it comes to printing. It's very easy to work with the print engine to print out entire documents or sections of documents.

Working with Print Preview

You can display a document in Print Preview mode by setting the active document's **PrintPreview** property to True:

```

' Visual Basic
ThisDocument.PrintPreview = True

// C#
ThisDocument.PrintPreview = true;

```

You can toggle the **PrintPreview** property of the **Application** object to display the current document in Print Preview mode. If the document is already in preview mode, it will be displayed in normal view, if it is in normal view, it will be displayed in Print Preview:

```

' Visual Basic
Friend Sub TogglePrintPreview()
    ThisApplication.PrintPreview = Not ThisApplication.PrintPreview
End Sub

// C#
public void TogglePrintPreview()
{
    ThisApplication.PrintPreview = !ThisApplication.PrintPreview;
}

```

The PrintOut Method

You can use the **PrintOut** method to send a document (or part of a document) to the printer. You can call it from an **Application** or **Document** object. The following code fragment prints out the active document with all of the default options:

```

' Visual Basic
ThisDocument.PrintOut()

// C#
Object background = Type.Missing;
Object append = Type.Missing;
Object range = Type.Missing;
Object outputFileName = Type.Missing;
Object from = Type.Missing;
Object to = Type.Missing;
Object item = Type.Missing;
Object copies = Type.Missing;
Object pages = Type.Missing;
Object pageType = Type.Missing;
Object printToFile = Type.Missing;
Object collate = Type.Missing;
Object fileName = Type.Missing;
Object activePrinterMacGX = Type.Missing;
Object manualDuplexPrint = Type.Missing;
Object printZoomColumn = Type.Missing;
Object printZoomRow = Type.Missing;
Object printZoomPaperWidth = Type.Missing;
Object printZoomPaperHeight = Type.Missing;

ThisDocument.PrintOut(ref background, ref append,
    ref range, ref outputFileName, ref from, ref to,
    ref item, ref copies, ref pages, ref pageType,
    ref printToFile, ref collate, ref fileName, ref activePrinterMacGX,
    ref manualDuplexPrint, ref printZoomColumn, ref printZoomRow,
    ref printZoomPaperWidth, ref printZoomPaperHeight);

```

The **PrintOut** method has multiple optional arguments that allow you to fine tune how to print the document, as summarized in **Table 2**.

Table 2. Commonly used PrintOut arguments

Argument	Description
Background	Set to True to allow processing while Word prints the document
Append	Use this with the <i>OutputFileName</i> argument. Set to True to append the specified document to the file name specified by the <i>OutputFileName</i> argument. Set to False to overwrite the contents of <i>OutputFileName</i> .
Range	The page range. Can be any WdPrintOutRange enumeration: wdPrintAllDocument , wdPrintCurrentPage , wdPrintFromTo , wdPrintRangeOfPages , or wdPrintSelection
OutputFileName	If PrintToFile is True, this argument specifies the path and file name of the output file.
From	The starting page number when Range is set to wdPrintFromTo
To	The ending page number when Range is set to wdPrintFromTo
Item	The item to be printed. Can be any WdPrintOutItem enumeration: wdPrintAutoTextEntries , wdPrintComments , wdPrintDocumentContent , wdPrintKeyAssignments , wdPrintProperties , wdPrintStyles
Copies	The number of copies to be printed
Pages	The page numbers and page ranges to be printed, separated by commas. For example, "2, 6-10" prints page 2 and pages 6 through 10.
PageType	The type of pages to be printed. Can be any WdPrintOutPages constant: wdPrintAllPages , wdPrintEvenPagesOnly , wdPrintOddPagesOnly
PrintToFile	Set to True to send printer instructions to a file. Make sure to specify a file name with OutputFileName .
Collate	Use when printing multiple copies of a document. Set to True to print all pages of the document before printing the next copy.
FileName	Available only with the Application object. The path and file name of the document to be printed. If this argument is omitted, Word prints the active document.
ManualDuplexPrint	Set to True to print a two-sided document on a printer without a duplex printing kit.

The following procedure prints out the first page of the active document:

```
' Visual Basic
Friend Sub PrintOutDoc()
    ThisDocument.PrintOut( _
        Background:=True, _
        Append:=False, _
        Range:=Word.WdPrintOutRange.wdPrintCurrentPage, _
        Item:=Word.WdPrintOutItem.wdPrintDocumentContent, _
        Copies:=2, _
        Pages:=1, _
        PageType:=Word.WdPrintOutPages.wdPrintAllPages, _
        PrintToFile:=False, _
        Collate:=True, _
        ManualDuplexPrint:=False)
End Sub

// C#
public void PrintOutDoc()
{
    Object background = true;
    Object append = false;
    Object range = Word.WdPrintOutRange.wdPrintCurrentPage;
    Object outputFileName = Type.Missing;
    Object from = Type.Missing;
    Object to = Type.Missing;
    Object item = Word.WdPrintOutItem.wdPrintDocumentContent;
    Object copies = 2;
    Object pages = 1;
    Object pageType = Word.WdPrintOutPages.wdPrintAllPages;
    Object printToFile = false;
    Object collate = Type.Missing;
    Object fileName = Type.Missing;
    Object activePrinterMacGX = Type.Missing;
    Object manualDuplexPrint = Type.Missing;
    Object printZoomColumn = Type.Missing;
    Object printZoomRow = Type.Missing;
    Object printZoomPaperWidth = Type.Missing;
    Object printZoomPaperHeight = Type.Missing;

    ThisDocument.PrintOut(ref background, ref append,
        ref range, ref outputFileName, ref from, ref to,
        ref item, ref copies, ref pages, ref pageType,
        ref printToFile, ref collate, ref fileName, ref activePrinterMacGX,
        ref manualDuplexPrint, ref printZoomColumn, ref printZoomRow,
        ref printZoomPaperWidth, ref printZoomPaperHeight);
}
```

Creating Word Tables

The **Tables** collection is a member of the **Document**, **Selection**, and **Range** objects, which means that you can create a table in any of those contexts. You use the **Add** method to add a table at the specified range. The following code adds a table consisting of three rows and four columns at the beginning of the active document:

```
' Visual Basic
Dim rng as Word.Range = _
    ThisDocument.Range(0, 0)
ThisDocument.Tables.Add(rng, 3, 4)

// C#
Object start = 0;
Object end = 0;
Word.Range rng = ThisDocument.Range(ref start, ref end);

Object defaultTableBehavior = Type.Missing;
Object autoFitBehavior = Type.Missing;
ThisDocument.Tables.Add(rng, 3, 4, ref defaultTableBehavior,
    ref autoFitBehavior);
```

Working with a Table Object

Once you've created the table, you automatically add it to the **Document** object's **Tables** collection and you can then refer to the table by its item number, as shown in the following code fragment:

```
' Visual Basic
Dim tbl As Word.Table = ThisDocument.Tables(1)

// C#
Word.Table tbl = ThisDocument.Tables[1];
```

Each **Table** object also has a **Range** property, which allows you to set direct formatting attributes. The **Style** property allows you to apply one of the built-in styles to the table, as shown in the following code fragment:

```
' Visual Basic
With ThisDocument.Tables(1)
    .Range.Font.Size = 8
    .Style = "Table Grid 8"
End With

// C#
Word.Table tbl = ThisDocument.Tables[1];
tbl.Range.Font.Size = 8;
Object style = "Table Grid 8";
tbl.set_Style(ref style);
```

The Cells Collection

Each Table consists of a collection of Cells, with each individual **Cell** object representing one cell in the table. You refer to each cell by its location in the table. The following code refers to the cell located in the first row and the first column of the table, adding text and applying formatting:

```
' Visual Basic
With ThisDocument.Tables (1)
    With .Cell(1, 1).Range
        .Text = "Name"
        .ParagraphFormat.Alignment = _
            Word.WdParagraphAlignment.wdAlignParagraphRight
    End With
End With

// C#
Word.Range rng = ThisDocument.Tables[1].Cell(1, 1).Range;
rng.Text = "Name";
rng.ParagraphFormat.Alignment =
    Word.WdParagraphAlignment.wdAlignParagraphRight;
```

Rows and Columns

The cells in a table are organized into rows and columns. You can add a new row to a table by using the **Add** method:

```
' Visual Basic
Dim tbl As Word.Table = ThisDocument.Tables(1)
tbl.Rows.Add()

// C#
Word.Table tbl = ThisDocument.Tables[1];
Object beforeRow = Type.Missing;
tbl.Rows.Add(ref beforeRow);
```

Although you generally define the number of columns when you create a new table, you can also add columns after the fact with the **Add** method. The following code adds a new column to an existing table, inserting the new column before the existing first column, and then uses the **DistributeWidth** method to make them all the same width:

```
' Visual Basic
Dim tbl As Word.Table = ThisDocument.Tables(1)
tbl.Columns.Add(tbl.Columns(1))
tbl.Columns.DistributeWidth
```

```
// C#
Word.Table tbl = ThisDocument.Tables[1];
Object beforeColumn = tbl.Columns[1];
tbl.Columns.Add(ref beforeColumn);
tbl.Columns.DistributeWidth();
```

Pulling it all Together

The following example creates a Word table at the end of the active document and populates it with document properties:

```
' Visual Basic
Public Sub CreateTable()
    ' Move to start of document.
    Dim rng As Word.Range = _
        ThisDocument.Range(0, 0)

    ' Insert some text and paragraph marks.
    rng.InsertBefore("Document Statistics")
    rng.Font.Name = "Verdana"
    rng.Font.Size = 16
    rng.InsertParagraphAfter()
    rng.InsertParagraphAfter()
    rng.SetRange(rng.End, rng.End)

    ' Add the table.
    Dim tbl As Word.Table = rng.Tables.Add( _
        ThisDocument.Paragraphs(2).Range, 3, 2)

    ' Format the table and apply a style.
    tbl.Range.Font.Size = 12
    tbl.Columns.DistributeWidth()
    tbl.Style = "Table Colorful 2"

    ' Insert text in cells.
    tbl.Cell(1, 1).Range.Text = "Document Property"
    tbl.Cell(1, 2).Range.Text = "Value"
    tbl.Cell(2, 1).Range.Text = "Number of Words"
    tbl.Cell(2, 2).Range.Text = ThisDocument.Words.Count.ToString
    tbl.Cell(3, 1).Range.Text = "Number of Characters"
    tbl.Cell(3, 2).Range.Text = _
        ThisDocument.Characters.Count.ToString
    tbl.Select()
End Sub

// C#
public void CreateTable()
{
    // Move to start of document.
    Object start = 0;
    Object end = 0;
    Word.Range rng = ThisDocument.Range(ref start, ref end);

    // Insert some text and paragraph marks.
    rng.InsertBefore("Document Statistics");
    rng.Font.Name = "Verdana";
    rng.Font.Size = 16;
    rng.InsertParagraphAfter();
    rng.InsertParagraphAfter();
    rng.SetRange(rng.End, rng.End);

    // Add the table.
    Object defaultTableBehavior = Type.Missing;
    Object autoFitBehavior = Type.Missing;
    Word.Table tbl = rng.Tables.Add(
        ThisDocument.Paragraphs[2].Range, 3, 2,
        ref defaultTableBehavior, ref autoFitBehavior);

    // Format the table and apply a style.
    tbl.Range.Font.Size = 12;
    tbl.Columns.DistributeWidth();
    Object style = "Table Colorful 2";
    tbl.set_Style(ref style);

    // Insert text in cells.
    tbl.Cell(1, 1).Range.Text = "Document Property";
    tbl.Cell(1, 2).Range.Text = "value";
    tbl.Cell(2, 1).Range.Text = "Number of Words";
    tbl.Cell(2, 2).Range.Text =
        ThisDocument.Words.Count.ToString();
    tbl.Cell(3, 1).Range.Text = "Number of Characters";
    tbl.Cell(3, 2).Range.Text =
        ThisDocument.Characters.Count.ToString();
    tbl.Select();
}
```

Summary

Word has a rich object model that enables you to programmatically control Word and the creation of documents from managed code. This article has only scratched the surface of what's available. Should you wish to investigate further, see the online Help file for VBA in Word. Armed with the information from this article you should be able to tackle almost any task involving document creation and production.

Investigate further, see the online help for the Word object model. This article is intended to be used as a reference for the Word object model. It is not intended to be used as a guide for creating or modifying documents.