The VSTO Programming Model

In Windows Forms programming, a form is a window that contains controls, such as buttons, combo boxes, and so on. To implement a form, you can drag and drop controls from the Visual Studio toolbox onto the form’s designer. The form designer then generates a customized subclass of the Form class. Because each form is implemented by its own class, you can then further customize the form code by adding properties and methods of your own to the class. And because all the controls are added as properties on the form class, you can use IntelliSense to more rapidly program those custom methods.

VSTO’s system of host items and host controls is directly analogous to Windows Forms. By “host” we mean the application—Word or Excel—which hosts the customization. Host items are like forms: programmable objects that contain user interface elements called host controls. The Workbook, Worksheet, and ChartSheet objects are host items in Excel; the Document object is the sole host item in Word. In Outlook, the Outlook Application object is exposed as a host item.

As we saw back in Chapter 2, “Introduction to Office Solutions,” the Visual Studio Excel and Word designers create custom classes which extend the Worksheet and Document base classes. As you place host controls such as lists, named ranges, charts and buttons onto the worksheet they are exposed as fields on the customized subclass.
Separation of Data and View

Some people use spreadsheet software solely for its original purpose: to lay out financial data on a grid of cells that automatically recalculates sums, averages and other formulas as they update the data. For example, you might have a simple Excel spreadsheet that calculates the total expenses for a wedding given all the costs involved. Similarly, some people use word-processing software solely for its original purpose: to automatically typeset letters, memos, essays, books and other written material.

However, in a business setting spreadsheets and documents have evolved to have both high internal complexity and external dependencies. Unlike a wedding budget, a spreadsheet containing an expense report or a document containing an invoice is likely to be just one small part of a much larger business process. This fact has implications on the design of a programming model. Consider this VBA code that might be found in a spreadsheet that is part of a larger business process:

```vba
SendUpdateEmail ThisWorkbook.Sheets(1).Cells(12,15).Value2
```

Clearly, the unreadable snippet is sending an e-mail to someone, but because the Excel object model emphasizes how the spreadsheet represents the data, not what the data represent, it is hard to say what exactly this is doing. The code is not only hard to read, it is brittle; redesigning the spreadsheet layout could break the code. We could improve this code by using a named range rather than a hard-coded direct reference to a particular cell:

```vba
SendUpdateEmail ThisWorkbook.Names("ApproverEmail").RefersToRange.Value2
```

Better, but it would be even nicer if the particular range showed up in IntelliSense. VSTO builds a convenient custom object model for each worksheet, workbook, or document so that you can more easily access the named items contained therein:

```vba
SendUpdateEmail(ExpenseReportSheet.ApproverEmail.Value2);
```

A more readable, maintainable, and discoverable object model is a welcome addition. However, even in the preceding snippet, the VSTO programming model still does not address the more fundamental problem: We are manipulating the data via an object model that treats them as part of a spreadsheet. The spreadsheet is still the...
lens through which we see the data; instead of writing a program that manipulates ice cream sales records, we wrote a program that manipulates a list and a chart.

The crux of the matter is that Word and Excel are *editors*; they are for *designing documents that display data*. Therefore, their object models thoroughly conflate the data themselves with the “view,” the information about how to display them. To mitigate this conflation, the VSTO programming model was designed to enable developers to logically separate view code from data code. Host items and host controls represent the “view” elements; host items and host controls can be data bound to classes that represent the business data.

**Model-View-Controller**

If you’re familiar with design patterns, you will have already recognized this as based on the Model-View-Controller (MVC) design pattern. In the MVC pattern, the data model code represents the business data and the processes that manipulate it. The view code reads the data, listens to Change events from the data, and figures out how to display it. The controller code mediates between the view and the data code, updating the data based upon the gestures the user makes in the view (mouse clicks, key presses, and so on).

![Figure 13-1 Model-View-Controller architecture.](image)

**Benefits of Separation**

Logically separating the data code from the view code leads to a number of benefits when building more complex business documents on top of Word and Excel:
• Business data and rules can be encapsulated in ADO.NET datasets and reused in different applications.
• Changes to view code are less likely to unexpectedly break data code (and vice versa).
• Data code can cache local copies of database state for offline processing.
• Server-side code can manipulate cached data inside the document without starting up Word/Excel.

Now that you know some of the design philosophy behind VSTO, let’s take a look at how the host items and host controls actually extend the Word and Excel object models. (The data side is covered in Chapter 17, “VSTO Data Programming,” and server-side data manipulation is covered in Chapter 18, “Server Data Scenarios.”)

**VSTO Extensions to Word and Excel Objects**

VSTO extends the Word and Excel object models in several ways. Although it is possible to use these features without understanding what is actually happening “behind the scenes,” it is helpful to take a look back there. This section explains by what mechanisms host items and host controls extend the Word and Excel programming models. Then the discussion focuses on exactly which new features are available.

**Aggregation, Inheritance, and Implementation**

If you create a Word project in Visual Studio and open the Object Browser window, you will see several assemblies listed. Two are of particular interest. You already know that the Microsoft.Office.Interop.Word assembly is the primary interop assembly (PIA), containing the definitions for the interfaces that allow managed code to call the unmanaged Word object model. Similarly, the Microsoft.Office.Interop.Excel assembly is the PIA for the unmanaged Excel object model.

You can find the VSTO extensions to the Word and Excel object models in the Microsoft.Office.Tools.Word and Microsoft.Office.Tools.Excel assemblies; each contains a namespace of the same name.

From a VSTO Word document project, open the Object Browser and take a look at the Document host item class in the Tools namespace, as shown in Figure 13-2.
Notice that the host item class implements the properties, methods, and events defined by the Document interface from the PIA, and extends the BindableComponent base class. Chapter 17 gets into the details of how data-bindable components work; for now, the fact that this class implements the properties, methods, and events from the PIA interface rather than extends a base class is important. It is important to notice that even though the Document host item class has all the methods, properties, and events of the Document interface from the PIA, the type definition does not actually say that it implements the Document interface itself. This is a subtle distinction that we will discuss in more detail later.

Conceptually, the difference between extending a base class and implementing the properties, methods, and events from an interface is that the former describes an “is a” relationship, whereas the latter describes a “can act like” relationship. A Microsoft.Office.Tools.Word.Document object really is a bindable component; it actually shares functionality—code—with its base class. But it merely looks like and acts like a Word Document object; it is not a Word document object as far as Word is concerned.

For example, the Sheet1 class in Excel has your event handlers and host controls. It extends the Microsoft.Office.Tools.Excel.Worksheet base class and implements the
Hooking Up the Aggregates

VSTO’s host item and host control objects aggregate some of the underlying Word and Excel document objects (such as the Document and Bookmark objects in Word, or the Worksheet and NamedRange objects in Excel). You have already seen how you can call methods on the document object in a VSTO customization. Suppose, for instance, that you call the CheckGrammar method on the document. If this is not really a Word Document object but merely looks like one, how does it work?

The aggregating object’s implementation of that method checks to see whether it has obtained the aggregated Document object already. If it has not, it makes a call into Word to obtain it (and caches away the object so that it will be available immediately when you make a second method call). After it has the reference to the aggregated object, the aggregating object calls CheckGrammar on the aggregated object.

The great majority of the properties and methods on the aggregating objects do nothing more than just pass the arguments along to the PIA code, which then passes them along to the unmanaged object model.

Events work in the analogous way; if your code listens to an event exposed by an aggregating object, the aggregating object listens to the event on the aggregated object on your behalf. When the event is raised by the aggregated object, the aggregating object’s delegate is called, which then raises the aggregating object’s event and calls your event handling delegate.

All the host controls are hooked up in a similar manner as the host items. For instance, if you have a NamedRange host control member of a worksheet, the aggregating Worksheet object itself creates an aggregating NamedRange object. The first time you call a method on the host control, the aggregating class obtains the underlying “real” object from Excel and passes the call along.

This might seem like a whole lot of rigmarole to go through just to add new functionality to the Word and Excel object models. The key benefit that this system of aggregates affords is that each host item class in each project can be customized. One spreadsheet can have an InvoiceSheet class with a CustomerNameRange property, another can have a MedicalHistorySheet class with a CholesterolLevelChart property, and so on.
In short, VSTO extends the Word and Excel object models by aggregating the unmanaged object models with managed objects. VSTO enables developers to further customize and extend some of those objects—those representing the workbook, worksheet, chart sheet, and document—through subclassing.

**Obtaining the Aggregated Object**

Much of the time, the foregoing details about how the aggregation model works are just that: implementation details. Whether the host item “is a” worksheet or merely “looks like” one seems to be an academic point. However, in some rare scenarios, it does matter.

Word’s and Excel’s object models were not written with the expectation that managed aggregates would implement their interfaces; when you call a method that takes a range, Excel expects that you are passing it a real range, not an aggregated range that acts like a range.

For instance, suppose you have a customized worksheet with two host controls: a NamedRange member called InvoiceTotals and a Chart object called InvoiceChart. You might want to write code something like this snippet:

```csharp
this.InvoiceChart.SetSourceData(this.InvoiceTotals, Excel.XlRowCol.xlColumns);
```

This code will not compile because the SetSourceData method on the chart aggregate must be passed an object that implements the Range interface. It looks like at runtime the InvoiceChart aggregate will pass InvoiceTotals, an aggregated range, to the “real” aggregated chart. But Excel will expect that the object passed to SetSourceData is a range, whereas in fact it is the VSTO aggregate; it merely looks like an Excel range.

When just calling methods, reading or writing properties, and listening to events, the aggregate is more or less transparent; you can just use the object as though it really were the thing it is aggregating. If for any reason you need to pass the aggregate to an Excel object model method that requires the real Excel object, you can obtain the real Excel object via the InnerObject property. The code above will compile and work properly if you rewrite it to look like this:

```csharp
this.InvoiceChart.SetSourceData(this.InvoiceTotals.InnerObject, Excel.XlRowCol.xlColumns);
```
Aggregation and Windows Forms Controls

If you drag and drop a Windows Forms button onto a worksheet or document, the button control is also aggregated. However, Windows Forms controls are aggregated slightly differently than the NamedRange, Bookmark, ListObject, and other controls built in to Word and Excel. There are two relevant differences between Windows Forms controls and Office’s controls. First, Windows Forms controls are implemented by extensible managed classes, unlike the unmanaged Office controls, which only expose interfaces in their PIAs. Second, Word and Excel controls inherently know how they are situated in relation to their containing document or worksheet; non-Office controls on a worksheet do not know that they are in a worksheet.

Word and Excel overcome the second difference by aggregating an extender onto a control sited on a document or worksheet. Word’s extender implements the properties, methods, and events of the _OLEControl interface that can be found in the Word PIA (but as with other aggregated VSTO controls, the type definition does not actually claim to implement the _OLEControl interface). It has five methods, all of which take no arguments and return no result: Activate, Copy, Cut, Delete, and Select. It also exposes floating-point read-write properties Top, Left, Height, and Width, string properties Name and AltHTML, and an Automation object. Excel’s extender implements the properties, methods, and events of the _OLEObject interface that can be found in the Excel PIA.

When you drop a button onto a document or worksheet, the project system adds a new field to the host item class, but types it as Microsoft.Office.Tools.Word.Controls.Button or Microsoft.Office.Tools.Excel.Controls.Button, respectively. Because the underlying System.Windows.Forms.Button class is extensible, this time the aggregate actually is a subclass of the Windows Forms control. However, it still must aggregate the unmanaged extender interface provided by Word or Excel.

As a further convenience, the managed objects representing embedded Windows Forms controls also have read-only Right and Bottom properties aggregated onto them.

Improving C# Interoperability

The Word and Excel object models were originally designed with VBA in mind. Unfortunately, there are some language features which VBA and VB.NET support but C# does not, such as parameterized properties. In VBA, you could do something like this:
Set Up = ThisWorkbook.Names.Item("MyRange").RefersToRange.End(xlUp)

End is a read-only property that takes an argument, but C# does not support passing arguments to property getters; arguments can only be passed to methods and indexers in C#. Therefore, the PIA exposes the property getter as a function. You could talk to the PIA like this in C#:


Note that the PIA interface calls out that this is a “getter” function; for writable properties there would be a corresponding set_ function that took the parameters and new value as arguments.

C# does, however, support something similar to parameterized property accessor: parameterized indexers. In a VSTO project with a host item or host item control that has been extended, you can accomplish the same task like this:

Up = MyRange.End[Excel.XlDirection.xlUp];

The get_End accessor function is implemented by the aggregate, so you can still use it if you want to. However, because it is no longer necessary and there is a more elegant solution, it is not displayed in the IntelliSense drop-down.

In several places in the VSTO object model, parameterized indexers have replaced parameterized properties; you will find a list of them all along with the rest of the changes to the object model at the end of this chapter.

The “Tag” Field

Every host item and host control now has a field called Tag, which can be set to any value. This field is entirely for you to use as you see fit; it is neither read nor written by any code other than your customization code. It is included because it is very common for developers to have auxiliary data associated with a particular control, but no field on the control itself in which to store the data. Having the object keep track of its own auxiliary data is, in many cases, more straightforward than building an external table mapping controls onto data.
**Event Model Improvements**

Like VBA, VSTO encourages an event-driven programming style. In traditional VBA programming, relatively few of the objects source events, which can make writing event-driven code cumbersome. For instance, in Word, the only way to detect when the user double-clicks a bookmark using the standard VBA object model is to declare an “events” class module with a member referring to the application:

```vba
Public WithEvents WordApp As Word.Application
```

Then sink the event and detect whether the clicked range overlaps the bookmark:

```vba
Private Sub App_WindowBeforeDoubleClick(ByVal Sel As Selection, Cancel As Boolean)
    If Sel.Range.InRange(ThisDocument.Bookmarks(1).Range) Then
        MsgBox "Customer Clicked"
    End If
End Sub
```

And initialize the event module:

```vba
Dim WordEvents As New WordEventsModule
Sub InitializeEventHandlers
    Set WordEvents.WordApp = Word.Application
End Sub
```

And then add code that calls the initialization method. In short, this process requires a fair amount of work to detect when an application-level event refers to a specific document or control. The VSTO extensions to the Word and Excel object models were designed to mitigate difficulties in some tasks, such as sinking events on specific controls. In VSTO, the bookmark object itself sources events, so you can start listening to it as you would sink any other event:

```vba
MyBookmark.BeforeDoubleClick += new ClickEventHandler(OnDoubleClick);
```

In Chapter 2, you saw some of the new VSTO extensions to the view object model in action. You also read about events added by VSTO in Chapters 4, “Working with Excel Events,” and 7, “Working with Word Events.” At the end of this chapter, we describe all the additions to the event model in detail.
Dynamic Controls

In Chapter 2, you saw that VSTO allows developers to build customized document solutions by using Word and Excel as designers inside Visual Studio. The host item classes expose the host controls present at design time as custom properties on a class that aggregates the underlying unmanaged object.

But what about host controls not present at design time? What if you want to create new named ranges, bookmarks, buttons, or other controls at runtime? It would be nice to be able to use the new events and other extensions to the programming model on dynamically generated controls. As you will see, VSTO supports dynamically adding both host items and host controls, although the former is a little bit trickier to pull off.

Chapter 14 shows how to dynamically add Windows Forms controls to Word and Excel documents.

The Controls Collection

In a Windows Forms application, every form class has a property called Controls that refers to a collection of all the controls hosted by the form. In VSTO, each worksheet and document class contains a similarly named property; in Word, the document class contains an instance of Microsoft.Office.Tools.Word-ControlCollection, in Excel each worksheet class contains an instance of Microsoft.Office.Tools.Excel.ControlCollection. They are quite similar; the following sections discuss their differences.

Enumerating and Searching the Collection

You can use the Controls collection to enumerate the set of aggregated controls and perform actions upon all of them. For instance, you could disable all the button controls on a sheet or document:

```csharp
foreach (object control in this.Controls)
{
    Button button = control as Button;
    if (button != null)
        button.Enabled = false;
}
```
The Controls collection also has some of the indexing and searching methods you would expect. Both the Excel and Word flavors have methods with these signatures:

```csharp
bool Contains(string name)
bool Contains(object control)
int IndexOf(string name)
int IndexOf(object control)
```

If the collection does not contain the searched-for control, then IndexOf returns -1. Both collections can be enumerated via the `foreach` loop; should you want to enumerate the collection yourself, you can call GetEnumerator. This method returns a `ControlCollectionEnumerator` object from the Microsoft.Office.Tools.Excel or Microsoft.Office.Tools.Word namespace, as appropriate. They are essentially identical functionally. Both classes have only three public methods:

- object get Current
- bool MoveNext()
- void Reset()

Current returns `null` when moved past the final element in the collection, MoveNext moves the enumerator to the next element, and Reset starts the enumerator over at the beginning of the collection.

Both collections also expose three index operators, which take a name string, int index, and object respectively. The indexers throw an `ArgumentOutOfRangeException` if there is no such control in the collection.

**Adding New Word and Excel Host Controls Dynamically**

The worksheet and document Controls collections provide methods to dynamically create host controls. In Word, you can dynamically create aggregated bookmarks:

```csharp
    Microsoft.Office.Interop.Word.Range range, string name)
```

This method creates a new bookmark on the given range and aggregates it with the VSTO host control class.
XMLNode and XMLNodes host controls cannot be created dynamically in Word. The XMLMappedRange host control cannot be created dynamically in Excel.

In Excel, you can dynamically create aggregated NamedRanges, ListObjects, and Chart controls. Of those, only Chart controls can be positioned at arbitrary coordinates; the rest must all be positioned with a range object:

```csharp
    Microsoft.Office.Interop.Excel.Range range, string name)
    double left, double top, double width, double height, string name)
    Microsoft.Office.Interop.Excel.Range range, string name)
    Microsoft.Office.Interop.Excel.Range range, string name)
```

**Removing Controls**

The host controls added to a worksheet or document host item class at design time are exposed as properties on the host item class. If at runtime the user were to accidentally delete one, save the document, and then reload it, the customization code would be unable to find the aggregated control. This would likely result in an exception because eventually the customization would try to listen to an event or call a method on the missing aggregated control. If the customization detects this condition, it will throw a ControlNotFoundException.

Although it is difficult to prevent end users from accidentally or deliberately deleting controls without locking the document, the Controls collection can at least try to prevent programmatic destruction of controls added at design time. There are four equivalent ways to remove controls from the Controls collection; all will throw a CannotRemoveControlException if you attempt to remove a control that was not added dynamically.

The four ways to remove a dynamic control are to call Delete() on the control itself, or to call Remove(object control), Remove(string name), or RemoveAt(int
index) on the Controls collection itself. All four of these remove the control from the collection, remove the control from the document or worksheet, and destroy the extender object.

Most collections have a Clear() method that removes every member from the collection. Because completely clearing a Controls collection would almost always result in an exception when a design-time control was removed, this method always throws a NotSupportedException, and is hidden from IntelliSense.

Dynamic Controls Information Is Not Persisted

What happens when you add one or more dynamic controls to a document, save it, and reload it later?

Dynamically created Windows Forms controls such as buttons and check boxes do not survive being saved and then loaded. They just disappear; your customization code can create them again afresh the next time the document is loaded.

Because “host” controls such as ranges and bookmarks are themselves part of the document, they will be persisted along with the rest of the document. However, the controls do not save any information about any aggregating objects you may have created around them. When the document is reloaded, the controls will still be there, but there will be no aggregates wrapping them. You will have to re-add the controls to the Controls collection to create new aggregates for the controls. The Controls collection provides Add methods that can reconnect an aggregate to an existing control in the document without creating a new control in the document.

Advanced Topic: Dynamic Host Items

As you have just seen, adding new aggregated host controls onto a host item is relatively straightforward: just call the appropriate method on the controls collection for the containing host item and the control is created, aggregated, and placed on the host item automatically.

But what if you should want to use some of the features of an aggregated host item class on a dynamically created worksheet? To do that, you need only three lines of code. Understanding those three lines will require us to delve somewhat deeper
into how the VSTO runtime, the hosting application, and the aggregating class all work together.

Start by creating a helper method on an existing worksheet class that takes in the worksheet you want to be aggregated and returns an aggregated worksheet:

```csharp
{

Recall that the aggregating object obtains the aggregated object “on demand.” That is, it obtains the underlying object only when the first method is called that must be passed along to the underlying object. That means that the aggregating object must not require the aggregated object when the aggregating object is constructed, but it does need to be able to obtain that object at any time. Somehow the aggregating object must talk to the host and obtain the unique object is aggregating.

It does so by passing a string called “the cookie,” which identifies the aggregated object to a special service object provided by the host. In the event that an error occurs when attempting to fetch the worksheet, the runtime will need to raise an error. It is possible that the cookie that uniquely identifies the aggregated object might contain control characters or be otherwise unsuitable for display. Therefore, the aggregate constructor also takes a “human-readable” name used in the event that the host is unable to find the object to be aggregated. In the case of Excel worksheets, we will use a cookie that is already created for each worksheet by VBA called the CodeName. To initialize that cookie, we must make a call into the VBA engine to force the cookie to be created.

How do we obtain a reference to the service that maps cookies onto unmanaged host objects? The already aggregated host item has a member variable called Run-timeCallback that contains a reference to the VSTO runtime library’s service provider. Service provider is actually a bit of a misnomer; a service provider is an object that knows how to obtain objects that provide services, not necessarily one that provides those services itself. We identify services by the interface they implement.

Finally, to make data binding work properly, the aggregating class needs to know what object contains this worksheet; Chapter 17 covers data binding in more detail.
Let’s put all this together. We need to obtain five things to create an aggregating worksheet:

- A host-provided service that can obtain the aggregated object
- The cookie that the host application uses to identify the worksheet
- A human-readable name for the worksheet
- The container of the worksheet
- The VSTO runtime service provider

We obtain the service that maps the name and container to the aggregated object by passing the appropriate interface type to the VSTO runtime service provider:

```csharp
IHostItemProvider hostItemProvider = (IHostItemProvider) this.RuntimeCallBack.GetService(typeof(IHostItemProvider));
```

We next have to make a call into VBA to initialize the CodeName for the new worksheet. This line of code does nothing except force VBA to initialize. It does not add a VBA project to the workbook or anything else of that nature. However, it does access the VBProject object. For a solution that dynamically creates host items in Excel, you must make sure that users of your solution have Trust access to Visual Basic Project checked in the VBA Security dialog (Tools > Macro > Security). Otherwise, this line of code will fail:

```csharp
this.VBProject.VBComponents.Item(1);
```

We will use the name of the new Worksheet object for the human-readable name and the CodeName as the host cookie. The container of the new worksheet is the same as the container of the current worksheet:

```csharp
```

Just as dynamic host controls are not re-created when a document containing them is saved and then reloaded, dynamic host items are also not re-created.
**Advanced Topic: Inspecting the Generated Code**

Let’s take a deeper look behind the scenes at what is going on when you customize a worksheet or document. Create a new Excel C# project, create a named range, and take a look at the code for Sheet1.cs.

**Listing 13-1 The Developer’s Customized Worksheet Class**
```csharp
namespace ExcelWorkbook1
{
    public partial class Sheet1
    {
        private void Sheet1_Startup(object sender, System.EventArgs e)
        {
            this.MyRange.Value2 = "Hello";
        }
        private void Sheet1_Shutdown(object sender, System.EventArgs e)
        {
        }
        #region VSTO Designer generated code
        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InternalStartup()
        {
            this.Startup += new System.EventHandler(Sheet1_Startup);
            this.Shutdown += new System.EventHandler(Sheet1_Shutdown);
        }
        #endregion
    }
}
```

Upon closer inspection, a few questions might come to mind. What does that `partial` mean in the class declaration? Where is the MyRange property declared and initialized? Didn’t we say earlier that the customized worksheet class extends a base class? Where is the base class declaration?

It’s the `partial` that is the key. C# and Visual Basic support a new syntax that allows a class declaration to be split up among several files. The portion that you see before you is the home of all your developer-customized code; the automatically generated code is hidden in another portion of the class not displayed by default.
Click the Show All Files button in the Solution Explorer and you will see that a number of normally hidden files make up the class, as shown in Figure 13-3.

![Figure 13-3 Using the Show All Files button to examine hidden code.](image)

First, notice that behind every worksheet there is an XML file for the worksheet. If you look at the first few lines of the XML, you will see that it contains a description of the contents of the worksheet and how to represent it as a class. This “blueprint” contains information about what namespace the class should live in, what the name of the class should be, what controls are exposed on the class, how Excel identifies those controls, and so on.

Behind this language-independent representation of the class there is another C# file that contains the other half of the partial class, generated from the XML blueprint. It begins something like this:

```csharp
namespace ExcelWorkbook1 {
    [System.Security.Permissions.PermissionSetAttribute(
        Name="FullTrust")]
    public sealed partial class Sheet1 : Microsoft.Office.Tools.Excel.Worksheet,
```
As you can see, here is where the base classes are specified and the member variables declared. The class also specifies that it is one of the startup classes in your customization assembly, and that code that calls members of this class must be fully trusted.

There is plenty more code in the hidden portion of the partial class, most of which is devoted to initializing controls, starting up data binding, and handling data caching; Chapter 17 discusses data binding in more detail. The constructor, in particular, should look familiar:

```csharp
public Sheet1(IRuntimeServiceProvider RuntimeCallback) :
    base(((IHostItemProvider)(RuntimeCallback.GetService(
        typeof(IHostItemProvider)))), RuntimeCallback, "Sheet1",
        null, "Sheet1")
{
    this.RuntimeCallback = RuntimeCallback;
}
```

This is functionally the same code as just discussed in the previous section on creating custom host items by calling the aggregate base class constructor.

If you ever want to debug through this code, ensure that Just My Code Debugging is turned off (via the Tools > Options > Debugging > General dialog); you can then put breakpoints on any portion of the hidden code, just like any other code.

Do not attempt to edit the hidden code. Every time you make a change in the designer that would result in a new control being added, or even change a control property, the hidden half of the partial class is completely regenerated. Any changes you have made to the hidden half will be lost; that is why it is hidden by default!

### The Startup and Shutdown Sequences

You have probably noticed by now that we have been putting custom initialization code in an event handler:

```csharp
private void Sheet1_Startup(object sender, System.EventArgs e) {
    this.MyRange.Value2 = "Hello";
}
```
But exactly what happens, in what order, as the startup classes are created and initialized? Excel customizations typically have many startup classes, one for each sheet and one for the workbook itself; which ones load first?

You already saw a clue that answers the latter question. In the hidden half of the partial class, each class declaration has an attribute:

```
 StartupObjectAttribute(1)]
```

The Workbook class has 0 for the argument, Sheet1 has 1, Sheet2 has 2, and so on. The workbook aggregate always has ordinal 0, and each worksheet is given its ordinal based on what order Excel enumerates its sheets. The startup sequence happens in four phases, and each phase is executed on each startup class in order of the given ordinal before the next phase begins.

In the first phase, each class is constructed using the constructor mentioned above. This simply constructs the classes and stores away the information that will be needed later to fetch the unmanaged aggregated objects from Excel or Word.

In the second phase, the Initialize method of each startup class is called—again, in multiclass customizations, starting with the workbook and then each worksheet by ordinal. If you look at the hidden half of the partial class, you will see the Initialize method:

```
[global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
[global::System.ComponentModel.EditorBrowsableAttribute(
 System.ComponentModel.EditorBrowsableState.Never)]
public void Initialize() {
    this.HostItemHost = ((IHostItemProvider)
        (this.RuntimeCallback.GetService(typeof(IHostItemProvider))));
    this.DataHost = ((ICachedDataProvider)
        (this.RuntimeCallback.GetService(typeof(ICachedDataProvider))));
    Globals.Sheet1 = this;
    this.InitializeCachedData();
    this.InitializeControls();
    this.InitializeComponents();
    this.InitializeData();
    this.BeginInitialization();
}
```
The attributes prevent the Initialize method from showing up in IntelliSense drop-downs and mark the method as being “not my code” for the Debug Just My Code feature. The initializer then fetches services from the host needed to initialize the view and data elements, sets up the global class (discussed in more detail later in this chapter), loads cached data, and initializes all the controls.

In the third phase, data binding code is activated. Data bindings must be activated after all the classes are initialized because a control on Sheet2 might be bound to a dataset on Sheet1.

Finally, in the fourth phase, after everything is constructed, initialized, and data bound, each startup class raises its Startup event, and the code in the developer’s half of the partial class runs.

This multiphase startup sequence ensures that you can write handlers for the Startup event that can assume not just that the class itself is ready to go, but that every startup class in the customization is ready to go.

Ideally, it would be a good idea to write Startup event handlers for each class that do not depend on the order in which they are executed. If you must, however, you can always look at the startup attributes to see in what order the events will be executed.

The shutdown sequence is similar but simpler. As the host application, Word or Excel, shuts down, each host item class raises the Shutdown event. Shutdown events are raised in the same order as each phase in the startup sequence.

**The Globals Class in Excel**

Suppose you’re writing code in the Sheet1 class that needs to set a property on a control hosted by Sheet2. You are probably going to need to obtain the instance of the aggregated Sheet2 class somehow. Instead of aggregating properties representing all the other sheets and the workbook aggregates onto each startup class, VSTO exposes all the sheets and the workbook as static members of the Globals class:

```csharp
private void Sheet1_Startup(object sender, System.EventArgs e)
{
}
```
Because at least the first three phases of the startup sequence have finished at this point, you know that the Globals class and Sheet2 have been initialized, although Sheet2’s Startup event has probably not fired yet.

Notice that by default, controls aggregated onto the worksheet classes are given the internal visibility modifier. You can change the visibility modifier generated for a control by selecting the control in the designer and then selecting the Modifiers property in the Properties window. However, if you change the visibility of the control to private, you will be unable to access the control’s field from the Globals class.

The Globals class is also constructed using partial classes, although by default there is no visible portion. Rather, each generated code file defines a portion of theGlobals class. You can see this code at the bottom of the hidden file for each class. Should you for some reason want to add your own custom members to the Globals class, you can always create your own portion of the partial class.

VSTO Extensions to the Word and Excel Object Models

This chapter finishes up with a detailed list of every new property, event, and method aggregated onto the Word and Excel objects by the VSTO aggregates, with the exception of the new data binding features (which Chapter 17 covers). For Outlook, only the Application object is aggregated, and no new events, methods, or properties are added to that object.

As mentioned previously, every aggregated object now has a Tag property that you can use for any purpose you choose and an InnerObject property that you can use to access the aggregated object. In addition, each host control now has a Delete method that removes it (if it can be added dynamically at runtime) from its document or worksheet. Because every aggregating object has these properties and methods now, they are not mentioned again in the following topics.

The Word Document Class

Document objects in VSTO source the following new events shown in Table 13-1, all of which are raised by the Document object when the Application object raises the identically named event.

### Table 13-1 New Events on VSTO’s Aggregated Document Object

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Delegate</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActivateEvent</td>
<td>WindowEventHandler</td>
<td>From Application, renamed from WindowActivate</td>
</tr>
<tr>
<td>BeforeClose</td>
<td>CancelEventHandler</td>
<td>From Application</td>
</tr>
<tr>
<td>BeforeDoubleClick</td>
<td>ClickEventHandler</td>
<td>From Application</td>
</tr>
<tr>
<td>BeforePrint</td>
<td>CancelEventHandler</td>
<td>From Application</td>
</tr>
<tr>
<td>BeforeRightClick</td>
<td>ClickEventHandler</td>
<td>From Application</td>
</tr>
<tr>
<td>BeforeSave</td>
<td>SaveEventHandler</td>
<td>From Application</td>
</tr>
<tr>
<td>CloseEvent</td>
<td>DocumentEvents2_CloseEventHandler</td>
<td>From Document, renamed</td>
</tr>
<tr>
<td>Deactivate</td>
<td>WindowEventHandler</td>
<td>From Application</td>
</tr>
<tr>
<td>EPostageInsert</td>
<td>EventHandler</td>
<td>From Application</td>
</tr>
<tr>
<td>EPostagePropertyDialog</td>
<td>EventHandler</td>
<td>From Application</td>
</tr>
<tr>
<td>MailMergeAfterMerge</td>
<td>MailMergeAfterMergeEventHandler</td>
<td>From Application</td>
</tr>
<tr>
<td>MailMergeAfterRecordMerge</td>
<td>EventHandler</td>
<td>From Application</td>
</tr>
<tr>
<td>MailMergeBeforeMerge</td>
<td>EventHandler</td>
<td>From Application</td>
</tr>
<tr>
<td>MailMergeBeforeRecordMerge</td>
<td>CancelEventHandler</td>
<td>From Application</td>
</tr>
<tr>
<td>MailMergeDataSourceLoad</td>
<td>EventHandler</td>
<td>From Application</td>
</tr>
<tr>
<td>MailMergeDataSourceValidate</td>
<td>HandledEventHandler</td>
<td>From Application</td>
</tr>
<tr>
<td>MailMergeWindowSendToCustom</td>
<td>EventHandler</td>
<td>From Application</td>
</tr>
<tr>
<td>MailMergeWizardStateChange</td>
<td>MailMergeWizardStateChangeEventHandler</td>
<td>From Application</td>
</tr>
<tr>
<td>New</td>
<td>DocumentEvents2_NewEventHandler</td>
<td>From Document, delayed</td>
</tr>
</tbody>
</table>

*continues*
Notice that the Sync and Close events have been renamed to avoid a naming conflict; C# does not allow a class to have an event and a method with the same name.

The Document class now has OnStartup and OnShutdown methods that force the Document object to source the Startup and Shutdown events.

The New and Open events are delayed so that they are not raised until the aggregate class is fully initialized. These events would normally be raised before any user-authored code could run. If user code does not run until after the event has been raised, however, how would you add an event handling delegate to listen to the event? Therefore, the events are delayed until after the customization’s event binding code can run.

The event delegate types could use some additional explanation. All the event delegate types that begin with DocumentEvents2_ are from the Word PIA. The System.EventHandler, System.ComponentModel.CancelEventHandler and System.ComponentModel.HandledEventHandler delegates are straightforward. The remaining delegate types are all defined in the Microsoft.Office.Tools.Word namespace and have signatures as follows:

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Delegate</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open</td>
<td>DocumentEvents2_OpenEventHandler</td>
<td>From Document, delayed</td>
</tr>
<tr>
<td>SelectionChange</td>
<td>SelectionEventHandler</td>
<td>From Application</td>
</tr>
<tr>
<td>Shutdown</td>
<td>EventHandler</td>
<td></td>
</tr>
<tr>
<td>Startup</td>
<td>EventHandler</td>
<td></td>
</tr>
<tr>
<td>SyncEvent</td>
<td>DocumentEvents2_SyncEventHandler</td>
<td>From Application, renamed</td>
</tr>
<tr>
<td>WindowSize</td>
<td>WindowEventHandler</td>
<td>From Application</td>
</tr>
<tr>
<td>XMLAfterInsert</td>
<td>DocumentEvents2_XMLAfterInsertEventHandler</td>
<td>From Document</td>
</tr>
<tr>
<td>XMLBeforeDelete</td>
<td>DocumentEvents2_XMLBeforeDeleteEventHandler</td>
<td>From Document</td>
</tr>
</tbody>
</table>
delegate void ClickEventHandler(object sender, ClickEventArgs e);
delegate void MailMergeAfterMergeEventHandler(object sender, MailMergeAfterMergeEventArgs e);
delegate void MailMergeWizardStateChangeEventHandler(object sender, MailMergeWizardStateChangeEventArgs e);
delegate void SaveEventHandler(object sender, SaveEventArgs e);
delegate void SelectionEventHandler(object sender, SelectionEventArgs e);
delegate void WindowEventHandler(object sender, WindowEventArgs e);

The arguments classes of each are as follows:

- The `ClickEventArgs` class inherits from `System.ComponentModel.CancelEventArgs` and therefore has a `Cancel` property. It also exposes the selection that was clicked:

```csharp
class ClickEventArgs : CancelEventArgs {
    ClickEventArgs (Interop.Word.Selection selection, bool cancel)
    Interop.Word.Selection Selection { get; }
}
```

- The `MailMergeAfterMergeEventArgs` class exposes the new document created:

```csharp
class MailMergeAfterMergeEventArgs : EventArgs {
    Interop.Word.Document NewDocument { get; }
}
```

- The `MailMergeWizardStateChangeEventArgs` class exposes the previous, current, and handled states:

```csharp
class MailMergeWizardStateChangeEventArgs : EventArgs {
    MailMergeWizardStateChangeEventArgs (int fromState,
    int toState, bool handled)
    int FromState { get; }
    int ToState { get; }
    bool Handled { get; }
}
```
• The `SaveEventArgs` class allows the handler to instruct the event source whether the Save As dialog should display. This is also a cancelable event:

```csharp
class SaveEventArgs : CancelEventArgs {
    SaveEventArgs (bool showSaveAsUI, bool cancel)
    bool ShowSaveAsDialog { get; set; }
}
```

• The `SelectionEventArgs` class provides the selection that was changed:

```csharp
class SelectionEventArgs : EventArgs {
    SelectionEventArgs (Interop.Word.Selection selection)
    Interop.Word.Selection Selection { get; }
}
```

• The `WindowEventArgs` class provides the window that was activated, deactivated, or resized:

```csharp
class WindowEventArgs : EventArgs {
    WindowEventArgs (Interop.Word.Window window)
    Interop.Word.Window Window { get; }
}
```

In addition to the new events, the Document object also contains two new collections. First, as discussed earlier in this chapter, the Document object aggregate contains a collection of controls. Second, the Document object now contains a VSTOSmartTags collection (discussed further in Chapter 16, “Working with Smart Tags in VSTO”).

C# does not support parameterized properties, but two methods in the Document interface use parameterized properties. To make it easier to call these methods from C#, both properties now return instances of helper classes that allow you to use parameterized indexers. They are as follows:

```csharp
_ActiveWritingStyleType ActiveWritingStyle { get; }
_CompatibilityType Compatibility { get; }
```
The helper classes are scoped to within the customized host item’s base class itself, not to the Microsoft.Office.Tools.Word namespace.

The helper classes are as follows:

```csharp
class _ActiveWritingStyleType : System.MarshalByRefObject {
    public string this[object languageID] { get; set; }
}

class _CompatibilityType : System.MarshalByRefObject {
    public string this[Interop.Word.WdCompatibilityType] { get; set; }
}
```

This means that you can access these properties by passing the parameter to the index to fetch or set the property:

```csharp
style = this.ActiveWritingStyle[id];
```

The derived class can be further customized to add new events, methods, and properties. As you edit the document in the Word designer, any bookmarks or other host controls (such as buttons, check boxes, and so on) that you drop onto the design surface will be added as members of the document class. Similarly, any XML mappings added to the document will be added to the document class as either an XMLNode member (if the mapping is to a single node) or an XMLNodes member (if the mapping is to a repeatable node).

The document class has one additional new method, RemoveCustomization, which takes no arguments and has no return value. Calling this method on the aggregated document object removes the customization information from the document, so that after it is saved and reloaded, the customization code will no longer run.

Finally, the document class has a new property, ThisApplication, which refers to the Application object. This property exists to help migrate VSTO 2003 code that referred to a ThisApplication object. The document class also has an ActionsPane property, which is covered in detail in Chapter 15, “Working with Actions Pane.”
The Word Bookmark Host Control

Bookmark objects in the Word object model do not source any events. The aggregated host control Bookmark in VSTO sources the following new events shown in Table 13-2:

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Delegate</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeforeDoubleClick</td>
<td>ClickEventHandler</td>
</tr>
<tr>
<td>BeforeRightClick</td>
<td>ClickEventHandler</td>
</tr>
<tr>
<td>Deselected</td>
<td>SelectionEventHandler</td>
</tr>
<tr>
<td>Selected</td>
<td>SelectionEventHandler</td>
</tr>
<tr>
<td>SelectionChange</td>
<td>SelectionEventHandler</td>
</tr>
</tbody>
</table>

The delegate types and their corresponding argument classes are documented in the document class topic above.

As a convenience for both view programming and data binding, bookmark host controls also aggregate more than 150 methods and properties of the Range object that they represent. For example, these two lines of code are functionally identical:

```csharp
columns = this.bookmark1.range.columns;
columns = this.bookmark1.columns;
```

The methods and properties of the Range object aggregated onto the Bookmark object are for the most part straightforward proxies that just call the method or property accessor on the aggregated range, so almost all of the methods will be functionally identical whether you call them from the Range or the Bookmark.

Three exceptions apply. First, setting the Text property on the Range object directly can sometimes result in the bookmark itself being deleted by Word. If you set the Text property by calling the new property added to the Bookmark aggregate, it ensures that the bookmark is not deleted.

Second and third, the Information and XML properties from the PIA interface are parameterized properties. Because C# does not support calling parameterized properties, the bookmark host control uses helper classes that enable you to use parameterized indexers from C#. The properties are now defined as follows:
The helper classes are scoped inside the Bookmark class itself:

```csharp
class _InformationType : System.MarshalByRefObject {
    object this[Interop.Word.WdInformation Type] { get; }
}

class _XMLType : System.MarshalByRefObject {
    public string this[bool DataOnly] { get; }
}
```

You can then use the properties like this:

```csharp
info = this.myBookmark.Information[WdInformation.wdCapsLock];
```

**The Word XMLNode and XMLNodes Host Control Classes**

When you map a schema into a Word document, element declarations that have a `maxOccurs` attribute in the schema equal to 1 are represented in the host item class as XMLNode objects. All others are represented as XMLNodes objects, because there could be more than one of them.

Table 13-3 shows the new events in VSTO that the XMLNode and XMLNodes objects source.

**Table 13-3  New Events on VSTO’s Aggregated XMLNode and XMLNodes Objects**

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Delegate</th>
</tr>
</thead>
<tbody>
<tr>
<td>AfterInsert</td>
<td>NodeInsertAndDeleteEventHandler</td>
</tr>
<tr>
<td>BeforeDelete</td>
<td>NodeInsertAndDeleteEventHandler</td>
</tr>
<tr>
<td>ContextEnter</td>
<td>ContextChangeEvent Handler</td>
</tr>
<tr>
<td>ContextLeave</td>
<td>ContextChangeEvent Handler</td>
</tr>
<tr>
<td>Deselect</td>
<td>ContextChangeEvent Handler</td>
</tr>
<tr>
<td>Select</td>
<td>ContextChangeEvent Handler</td>
</tr>
<tr>
<td>ValidationError</td>
<td>EventHandler</td>
</tr>
</tbody>
</table>
As you can see, we have two new delegate classes, and therefore two new event argument classes. These events are normally sourced by the application object.

The delegates and event argument classes are all in the Microsoft.Office.Tools.Word namespace. The delegate classes are as follows:

```csharp
delegate void ContextChangeEventHandler(object sender, ContextChangeEventArgs e);
delegate void NodeInsertAndDeleteEventHandler(object sender, NodeInsertAndDeleteEventArgs e);
```

- When a node is inserted or deleted, it is often interesting to know whether the change is a result of the user inserting or deleting the element directly, or whether this is part of an undo or redo operation. This flag is therefore exposed on the event arguments class:

```csharp
class NodeInsertAndDeleteEventArgs : EventArgs {
    NodeInsertAndDeleteEventArgs (bool inUndoRedo)
    bool InUndoRedo { get; }
}
```

- When a node is selected or deselected, the appropriate event is raised. A “context change” is a special kind of selection change in which the insertion point of the document moves from one XML node to another. Therefore, the event arguments for the ContextEnter and ContextLeave events specify the node that was until recently the home of the insertion point, and the new home.

```csharp
class ContextChangeEventArgs : NodeSelectionEventArgs {
    Interop.Word.XMLNode OldXMLNode { get; }
    Interop.Word.XMLNode NewXMLNode { get; }
}
```

The XMLNode interface in the PIA has two parameterized properties, which are not supported in C#. Therefore, these properties have been redefined to return helper classes that implement parameterized indexers instead. The two methods are as follows:
Their helper classes are scoped to the XMLNode class itself. They are defined as follows:

```csharp
class _ValidationErrorTextType : System.MarshalByRefObject {
    string this[bool Advanced] { get; }
}

class _XMLType : System.MarshalByRefObject {
    string this[bool DataOnly] { get; }
}
```

XMLNode objects also implement several convenient new methods for manipulating the XML bound to the document:

```csharp
void LoadXml(string xml)
void LoadXml(System.Xml.XmlDocument document)
void LoadXml(System.Xml.XmlElement element)
void Load(string filename)
```

All of these take the contents of the XML in the argument and insert it into the given node and its children. However, the onus is on the caller to ensure both that the XML inserted into the node corresponds to the schematized type of the node, and that any child nodes exist and are mapped into the document appropriately. These methods will neither create nor delete child nodes.

As a further convenience for both view and data programming, the XMLNode object also provides a property that aggregates the Text property of the node's range:

```csharp
string NodeText { get; set; }
```

Chapters 15, “Working with ActionsPane,” 17, “VSTO Data Programming,” and 22, “Working with XML in Word,” cover data binding scenarios and actions pane scenarios for XMLNode and XMLNodes objects in detail. That sums up the VSTO extensions to the Word object model. The extensions to the Excel object models are similar but somewhat more extensive because of the larger number of host controls.
The Excel Workbook Host Item Class

The aggregating workbook class raises the same 29 events as the aggregated workbook class, with the same delegate types. Aside from renaming the Activate event to ActivateEvent, so as to avoid a collision with the method of the same name, there are no changes to the events raised by the Workbook object.

The Workbook object does have two new events raised when the customization starts up and shuts down:

```csharp
event System.EventHandler Startup;
event System.EventHandler Shutdown;
```

The aggregated Workbook object also has two new methods, OnStartup and OnShutdown, which cause the workbook to raise the Startup and Shutdown events.

As with the Word document class, the Excel workbook class gains a ThisApplication property, which refers back to the Excel Application object; an ActionsPane property, which Chapter 15 covers; and a VstoSmartTags collection, which Chapter 16 covers. The Workbook object also has one additional new method, RemoveCustomization, which takes no arguments and has no return value. Calling this method on the aggregated Workbook object removes the customization information from the spreadsheet, so that after it is saved and reloaded, the customization code will no longer run.

There is only one other minor change to the view programming model of the workbook class. Because C# cannot use parameterized properties, the Colors property now returns a helper class (scoped to the host item class itself) that allows you to use a parameterized index:

```csharp
<Colorstype Colors { get; }

class _Colorstype : System.MarshalByRefObject {
    object this[object Index] { get; set; }
```

The Excel Worksheet Host Item Class

Much like the workbook, the aggregating worksheet class does not have any major changes to its view programming model. The aggregating worksheet class raises the same eight events as the aggregated worksheet class, with the same delegate types.
Aside from renaming the Activate event to ActivateEvent, so as to avoid a collision with the method of the same name, there are no changes to the events raised by the Worksheet object.

The Worksheet object does have two new events raised when the customization starts up and shuts down:

```csharp
    event System.EventHandler Startup;
    event System.EventHandler Shutdown;
```

The Worksheet object has two new methods, OnStartup and OnShutdown, which cause the worksheet to raise the Startup and Shutdown events. The worksheet also provides the Controls collection mentioned earlier in this chapter.

Worksheets classes can be customized by subclassing; the derived classes generated by the design have properties representing charts, named ranges, XML-mapped ranges, list objects, and other controls on each sheet.

There is only one other minor change to the view programming model of the worksheet class. Because C# cannot use parameterized properties, the Range property now returns a helper class (scoped to the worksheet class itself) that allows you to use a parameterized index:

```csharp
_class Range { get; }

class _RangeType : System.MarshalByRefObject {
    Interop.Excel.Range this[object Cell1, object Cell2] { get; }
}
```

### The Excel Chart Sheet Host Item Class and Chart Host Control

Chart sheet host items and chart host controls are practically identical; the only difference between them as far as VSTO is concerned is that chart sheets are host items classes with their own designer and code-behind file. Charts, by contrast, are treated as controls embedded in a worksheet.

Both rename the Activate and Select events (to ActivateEvent and SelectEvent respectively) to avoid the name conflicts with the methods of the same name. The chart sheet host item class raises Startup and Shutdown events and has OnStartup and OnShutdown methods just as the worksheet class does.
Both the chart and the chart sheet have a parameterized HasAxis property that cannot be called from C#. The property therefore now returns an instance of a helper class that allows you to use a parameterized indexer instead:

```
(delegate_type HasAxis ( get; )

class delegate_type : System.MarshalByRefObject {
    object this[object Index1, object Index2] (get; set;)
}
```

The Excel NamedRange, XmlMappedRange, and ListObject Host Controls

All three of these are special kinds of Range objects. They raise the following new events shown in Table 13-4.

**Table 13-4** New Events on VSTO’s Aggregated NamedRange, XmlMappedRange, and ListObject Objects

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Delegate</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeforeDoubleClick</td>
<td>DocEvents_BeforeDoubleClickEventHandler</td>
</tr>
<tr>
<td>BeforeRightClick</td>
<td>DocEvents_BeforeRightClickEventHandler</td>
</tr>
<tr>
<td>Change</td>
<td>DocEvents_ChangeEventHandler</td>
</tr>
<tr>
<td>Deselected</td>
<td>DocEvents_SelectionChangeEventHandler</td>
</tr>
<tr>
<td>Selected</td>
<td>DocEvents_SelectionChangeEventHandler</td>
</tr>
<tr>
<td>SelectionChange</td>
<td>DocEvents_SelectionChangeEventHandler</td>
</tr>
</tbody>
</table>

All the event delegates are from the Microsoft.Office.Tools.Interop.Excel namespace in the Excel PIA.

The list object raises several more events in addition to those above, but because they all are primarily used to implement data binding functionality, Chapter 17 covers them.

There are many parameterized properties in both the NamedRange and XmlMappedRange interfaces that are not supported by C#. To make this functionality usable more easily from C#, these properties now return helper functions (scoped to the NamedRange or XmlMappedRange classes themselves) that expose parameterized indexers.
The NamedRange object only has one redefined property:

```csharp
_EndType End { get; }
```

The _EndType helper class is defined as follows:

```csharp
class _EndType : System.MarshalByRefObject {
    Interop.Excel.Range this[Interop.Excel XlDirection Direction] { get; }
}
```

The NamedRange aggregate also implements a parameterized indexer:

```csharp
object this[object RowIndex, object ColumnIndex] {
    get; set; }
```

The following properties are redefined on both NamedRange and XmlMappedRange aggregates:

```csharp
_AddressLocalType AddressLocal { get; }
_AddressType Address { get; }
_CharactersType Characters { get; }
_ItemType Item { get; }
_OffsetType Offset { get; }
_ResizeType Resize { get; }
```

The corresponding helper classes are defined as follows:

```csharp
class _AddressLocalType : System.MarshalByRefObject {
    string this[bool RowAbsolute, bool ColumnAbsolute,
    Interop.Excel.XlReferenceStyle ReferenceStyle, bool External,
    object RelativeTo] { get; }
}
class _AddressType : System.MarshalByRefObject {
    string this[bool RowAbsolute, bool ColumnAbsolute,
    Interop.Excel.XlReferenceStyle ReferenceStyle, bool External,
    object RelativeTo] { get; }
}
class _CharactersType : System.MarshalByRefObject {
    Interop.Excel.Characters this[int Start, int Length] { get; }
}
class _ItemType : System.MarshalByRefObject {
    object this[int RowIndex] { get; set; }
    object this[int RowIndex, int ColumnIndex] { get; set; }
}
```
As a convenience for both view and data programming, NamedRange host controls also expose directly all the methods of the associated Name object:

- `string RefersTo { get; set; }`
- `string RefersToLocal { get; set; }`
- `string RefersToR1C1 { get; set; }`
- `string RefersToR1C1Local { get; set; }`
- `Interop.Excel.Range RefersToRange { get; }`

If somehow the NamedRange object has been bound to a non-named range, these will throw `NotSupportedException`.

The NamedRange object also has a Name property that is somewhat confusing. The property getter returns the Name object associated with this named range. If you pass a Name object to the setter, it will set the Name property, just as you would expect. If you pass a string, however, it will attempt to set the Name property of the underlying Name object.

The NamedRange host control also slightly changes the exception semantics of the Name property in two ways. First, in the standard Excel object model, setting the Name property of the name object of a named range to the name of another named range deletes the range, oddly enough; doing the same to a VSTO NamedRange host control raises a `ArgumentException` and does not delete the offending range.

Second, in the standard Excel object model, setting the Name property to an invalid string fails silently. The VSTO NamedRange object throws an `ArgumentException` if the supplied name is invalid.

The `XMLMappedRange` and `ListObject` host controls do not aggregate the methods of the Name object or change the error handling semantics of the name setter. The changes to the Name property semantics only apply to the NamedRange object.
XML mapped ranges and list objects are the Excel equivalent of the XMLNode and XMLNodes controls in Word. The XML mapped range represents a mapped singleton element, and the list object represents a set of rows. We cover data binding scenarios in Chapter 17, “VSTO Data Programming,” and other XML scenarios in Excel in Chapter 21, “Working with XML in Excel.” In this chapter, we just discuss their use as host controls.

The list object host control has one new property:

```csharp
bool IsSelected { get; }
```

This property is most useful for determining whether there is an “insert row.” Excel does not display an insert row if the list object’s range is not selected.

The list object host control also slightly changes the error handling semantics of these properties:

```csharp
Interop.Excel.Range DataBodyRange { get; }
Interop.Excel.Range HeaderRowRange { get; }
Interop.Excel.Range InsertRowRange { get; }
Interop.Excel.Range TotalsRowRange { get; }
```

The only difference is that these properties now all return `null` rather than throwing an exception if you attempt to access the property on a list object that lacks a body, header, insert row, or totals row, respectively.

Chapter 17 discusses other new properties and methods added to the list object used for data binding.

**Conclusion**

VSTO brings the Word and Excel object models into the managed code world by aggregating key unmanaged objects onto managed base classes. Developers can then extend these base classes by using Word and Excel as designers in Visual Studio.

The next chapter takes a more detailed look at how to use Windows Forms controls in VSTO.