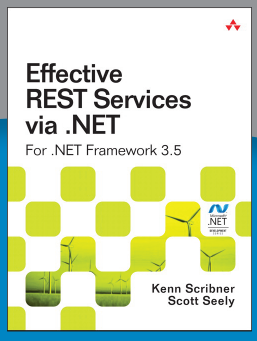
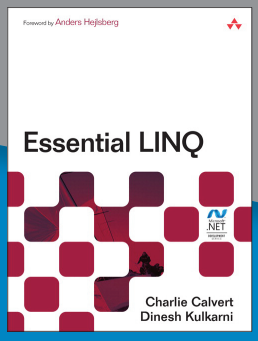
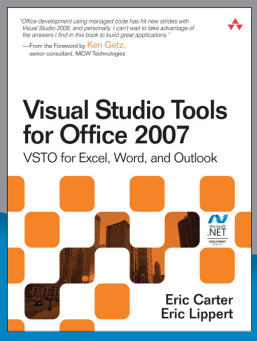
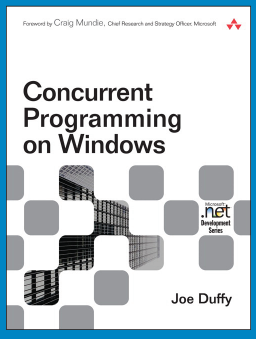
MICROSOFT® .NET DEVELOPER

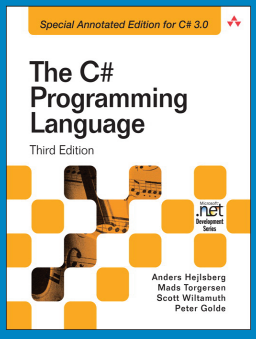
SAMPLE CHAPTERS

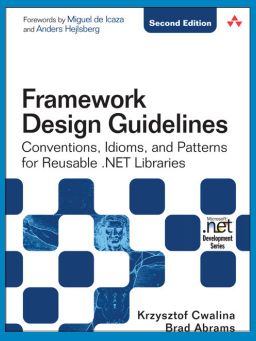


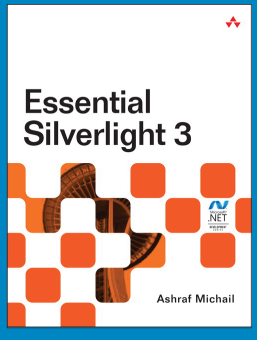


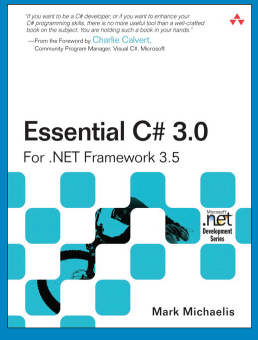


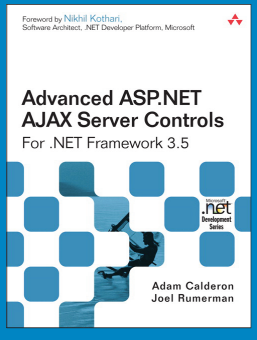








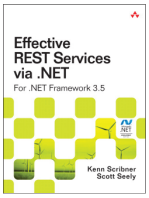




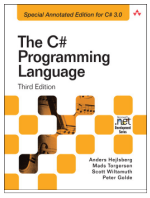
informIT.com/teched09

MICROSOFT® .NET DEVELOPER

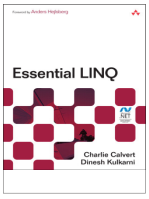
eBOOK TABLE OF CONTENTS



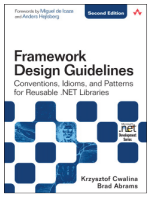
Effective REST Services via .NET
9780321613257
Kenn Scribner, Scott Seely
CHAPTER 3: Desktop Client Operations



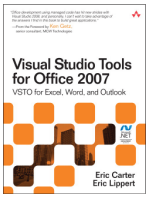
The C# Programming Language, Third Edition
9780321562999
Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth, Peter Golde
CHAPTER 3: Basic Concepts



Essential LINQ
9780321564160
Charlie Calvert, Dinesh Kulkarni
CHAPTER 3: The Essence of LINQ

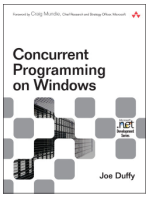


Framework Design Guidelines, Second Edition
97800321545619
Krzysztof Cwalina, Brad Abrams
CHAPTER 3: Naming Guidelines

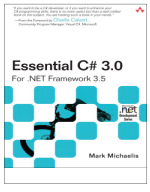


Visual Studio Tools for Office 2007
9780321533210
Eric Carter, Eric Lippert
CHAPTER 3: Programming Excel

Essential Silverlight 3
9780321554161
Ashraf Michail
CHAPTER 3: Graphics



Concurrent Programming on Windows
9780321434821
Joe Duffy
CHAPTER 3: Threads



Essential C# 3.0: For .NET Framework 3.5, Second Edition
9780321533920
Mark Michaelis
CHAPTER 3: Operators and Control Flow

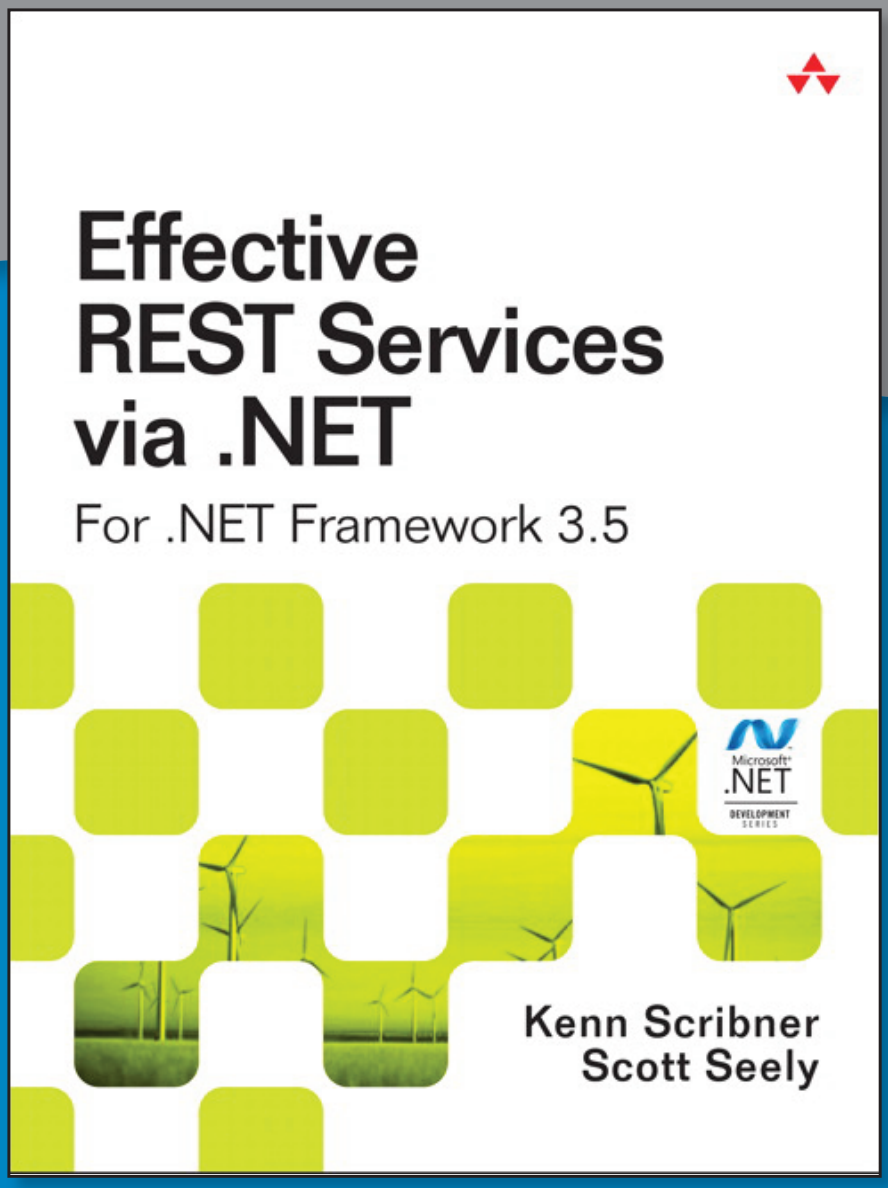
Advanced ASP.NET AJAX Server Controls for .NET Framework 3.5
9780321514448
Adam Calderon, Joel Rumerman
CHAPTER 3: Components

BROUGHT TO YOU BY

informIT.com
THE TRUSTED TECHNOLOGY LEARNING SOURCE

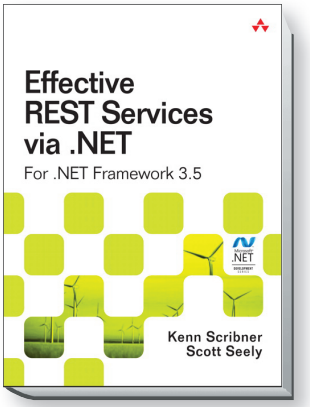
Addison Wesley

UPPER SADDLE RIVER, NJ | BOSTON | INDIANAPOLIS | SAN FRANCISCO | NEW YORK | TORONTO | MONTREAL | LONDON | MUNICH
PARIS | MADRID | CAPETOWN | SYDNEY | TOKYO | SINGAPORE | MEXICO CITY



Ken Scribner
Scott Seely

BUY ME



Effective REST Services via .NET

For .NET Framework 3.5

Developers are rapidly discovering the power of REST to simplify the development of even the most sophisticated Web services–and today's .NET platform is packed with tools for effective REST development. Now, for the first time, there's a complete, practical guide to building REST-based services with .NET development technologies.

Long-time .NET and Web services developers and authors Kenn Scribner and Scott Seely explain why REST fits so smoothly into the Internet ecosystem, why RESTful services are so much easier to build, what it means to be RESTful, and how to identify behaviors that are not RESTful. Next, they review the core Internet standards and .NET technologies used to develop RESTful solutions and show exactly how to apply them on both the client and server side. Using detailed code examples, Scribner and Seely begin with simple ASP.NET techniques, and then introduce increasingly powerful options–including Windows Communication Foundation (WCF) and Microsoft's cloud computing initiative, Azure. Coverage includes

- Accessing RESTful services from desktop applications, using Windows Forms and WPF
- Supporting Web client operations using Silverlight 2.0, JavaScript, and other technologies
- Understanding how IIS 7.0 processes HTTP requests and using that knowledge to build better REST services
- Constructing REST services based on traditional ASP.NET constructs
- Utilizing the ASP.NET MVC Framework to implement RESTful services more effectively
- Taking advantage of WCF 3.5's powerful REST-specific capabilities
- Creating RESTful data views effortlessly with ADO.NET Data Services
- Leveraging Microsoft's Azure cloud-computing platform to build innovative new services
- Choosing the right .NET technology for each REST application or service

AVAILABLE
- BOOK: 9780321613257
- SAFARI ONLINE Safari
- KINDLE: ????????????

About the Authors

Kenn Scribner has been writing cutting-edge, software-based books on Microsoft technologies for more than 10 years. His books include Windows Workflow Foundation Step by Step (Microsoft Press) and Understanding SOAP (SAMS). Kenn is a senior software consultant whose clients have included The Weather Channel, CBS, Burton, and Microsoft.

Scott Seely, an architect at MySpace, works on the OpenSocial API, one of the world's most successful REST-based APIs. Before joining MySpace, he was a developer on the Windows Communication Foundation team at Microsoft. His books include Creating and Consuming Web Services in Visual Basic (Addison-Wesley) and SOAP: Cross Platform Web Service Development Using XML (Prentice Hall).

Addison Wesley

informit.com/aw

3

Desktop Client Operations

EVERY DAY, MORE RESOURCES are made available on private intranets and the Internet at large. .NET has many different tools that allow you to consume these services from any application environment: Web, desktop, and mobile. In this chapter, we will examine the classes and tools you need to use in order to build applications for both of the .NET desktop class libraries: Windows Forms (WinForms) and Windows Presentation Foundation (WPF).

We Still Write Desktop Applications

The desktop application has all sorts of benefits that, today, trump anything you can do on the Web. Desktop applications have access to storage devices, arbitrary network resources, and network hardware. They can make application demands that their Web brethren cannot. For example, you can write a desktop application that will install only to a Windows machine that has .NET 3.5 or greater installed. Finally, these applications can do something useful even when they are disconnected from the network. Outlook will still let you read e-mail and create new messages when you are disconnected from your Exchange server. Outlook's fraternal twin, Outlook Web Access, needs a network connection to work. Because of the capability to do so much more when having access to inexpensive yet

powerful hardware resources (memory, CPU, storage, and so forth), desktop applications are not going away any time soon.

These desktop applications frequently become more useful, however, when they can connect to network-based resources. E-mail can be sent and received. Games get updates, communicate high scores, and, most important, allow for players to meet and play interactively. Individuals at your company update, modify, and delete documents through WebDAV and Windows Explorer. As a desktop application developer, you have one question you have to answer: "How do I get that data?" This chapter shows you how to obtain information from RESTful services.

Everything in this chapter can be applied to any executable you might write. Accessing RESTful data does not change whether you use a command-line application, a Windows service, a Windows Forms (WinForms) application, or even a Windows Presentation Foundation (WPF) application. The chapter concentrates on applications created using WinForms and WPF because those two environments have a requirement you do not necessarily have in other environments. Such as? The user interface (UI) has to stay responsive while the Web request is executing.

An Introduction to our Web Service

This chapter and the next focus on consuming RESTful services. The later parts of this book focus on implementing RESTful architectures. In Chapter 8, "Building REST Services Using WCF," we talk about building RESTful services using Windows Communication Foundation (WCF). This chapter utilizes one of the WCF services from Chapter 8. At this point, you do not need to know how the service is built, but you do need to know what the service does. A copy of the service is provided in this chapter's sample project in order to keep things easier to build and navigate as you work with this chapter's sample client applications.

The service itself demonstrates a few basic capabilities that pretty much every consumer/producer needs to understand:

- Exchanging binary data
- Exchanging simple data types

- Exchanging structured data
- Exchanging arrays of structured data

Understanding these simple building blocks enables you to build or consume any RESTful service. When I was looking at scenarios that demonstrate the previous capabilities without needing to implement an overly sophisticated solution, one scenario popped out as simple to understand and small enough to fit within the chapter of a book: sharing photos. Photos are binary and have extra, interesting attributes, such as an owner and a caption. In our case, photos have these pieces of metadata:

- Is the photo public or private?
- Who is the photo owner?
- Does the photo have a caption, and if so, what is it?
- Does the photo have an extended description?
- What is the photo's unique identifier?

The REST service allows users to do all sorts of things with photos. For photos you own, you can update the caption and description, and state whether the photo is public. Regardless of who you are, you can ask for a list of photos from a particular user. If you are that user, the list contains all photos. If you ask for someone else's list, only public photos are returned. To do all this work, the service supports URLs of the following forms:

- Add an image: `POST` to [base service address]/AddImage
- Update an image: `PUT` to [base service address]/Image/{imageId}
- Delete an image: `DELETE` to [base service address]/Image/{imageId}
- Get images for a user: `GET` to [base service address]/Images/{username}
- Get a single image for a user: `GET` to [base service address]/Image/{imageId}

For this chapter, we will be using the XML-based endpoint for this service. The `POST`, `PUT`, and `GET` verbs all manipulate an `ImageItem`, which in serialized XML form appears as shown in Listing 3.1.

Listing 3.1: `ImageItem` serialized as XML

```
<ImageItem xmlns="http://www.scottseely.com/RESTBook/2008"
    xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
    <Caption>Some caption</Caption>
    <Description>Some description</Description>
    <ImageBytes>AQIDBAUGAQIDBAU[...]</ImageBytes>
    <ImageId>33f741ae-934a-4c77-b7f8-0f316000ab53</ImageId>
    <ImageUrl>[path to ashx that will yield a valid
              image/jpeg]</ImageUrl>
    <PublicImage>true</PublicImage>
    <UserName>restuser</UserName>
</ImageItem>
```

Chapter 4, "Web Client Operations," shows the same object in JavaScript Object Notation (JSON) format. Just to show the difference here, the JSON representation of the object is given in Listing 3.2.

Listing 3.2: `ImageItem` serialized as JSON

```
{
    "Caption":"Some caption",
    "Description":"Some description",
    "ImageBytes":[1,2,3,4,5,6 ...],
    "ImageId":"4bfa5653-be2b-4198-8c21-bdbf5d2f7bc6",
    "ImageUrl":>[path to ashx that will yield a valid image/jpeg],
    "PublicImage":true,
    "UserName":"restuser"
}
```

It's important to note that the only time `ImageBytes` will contain data is for an HTTP `POST` request. The only time that `ImageUrl` will be populated is in response to an HTTP `GET` request. (You'd use the URI contained within `ImageUrl` to request the actual image.) Lastly, the RESTful service validates users based on a username and password.

Reading Data

We have lots of options for dealing with XML markup. Because reading and writing data is a big part, maybe even the biggest part, of consuming RESTful services, the first code examples in this chapter detail how we would read the `ImageItem` as XML. A complete description of every possible mechanism to read and write data is far beyond the scope of this book. Instead,

this section introduces you to the .NET namespaces and tools most often used—those you need to be familiar with. With knowledge of the basics, you should be able to implement special cases and go as deep as you need to go. As a goal, we want to read the XML and transform it into an object that is more useful to a .NET developer, a process known as *deserialization.* Let's start with an object that can hold the data, which I've named `ImageItem`. `ImageItem` needs seven fields: one for each element in the XML representation shown in Listing 3.1. Given the XML from Listing 3.1, the `ImageItem` class should have the structure shown in Listing 3.3.

Listing 3.3: The `ImageItem` class

```
public class ImageItem
{
    public string Caption { get; set; }
    public string Description { get; set; }
    public byte[] ImageBytes { get; set; }
    public Guid ImageId { get; set; }
    public Uri ImageUrl { get; set; }
    public bool PublicImage { get; set; }
    public string UserName { get; set; }
}
```

> **NOTE**
>
> Something to keep in mind is that you do not always need to populate a class to make use of the data. You could keep the data in an XML document, a database, or another storage medium. Populating the `ImageItem` class is simply for the convenience of the application's C#-based code. Populating a form, placing data in a database, or something else are also possible goals that similarly rely on the capability to extract information from an XML stream.

With .NET, we have lots of options for dealing with XML markup. We can parse the XML manually using `System.Xml.XmlDocument` and `System.Xml.Linq.XDocument`. We can also create special classes and use the serialization mechanisms offered by `System.Runtime.Serialization.DataContractSerializer` and `System.Xml.Serialization.XmlSerializer`.

To demonstrate the various techniques for working with XML data, I've created a sample application called `SerializationSampler` and placed it in this chapter's demonstration solution. All the XML deserialization techniques just mentioned are illustrated using a static method that returns a single `ImageItem` object, which is hard-coded as an XML string for simplicity. The code offers access to the XML-based representation via a `System.IO.Stream` named `DataStream`, which is shown in Listing 3.4.

Listing 3.4: The preserialized `ImageItem` resource

```
const string Data =
    "<ImageItem xmlns=\"http://www.scottseely.com/RESTBook/2008\" "
        + "xmlns:i=\"http://www.w3.org/2001/XMLSchema-instance\">"
        + "<Caption>Some caption</Caption>"
        + "<Description>Some description</Description>"
        + "<ImageBytes>AQIDBAUGAQIDBAUGAQIDBAUGAQIDBAUGAQIDBAUG"
        + "</ImageBytes>"
        + "<ImageId>33f741ae-934a-4c77-b7f8-0f316000ab53</ImageId>"
        + "<ImageUrl>"
        +    "http://www.scottseely.com/PhotoWeb/Image.ashx"
        +    "/33f741ae-934a-4c77-b7f8-0f316000ab53</ImageUrl>"
        + "<PublicImage>true</PublicImage>"
        + "<UserName>restuser</UserName>"
  + "</ImageItem> ";

static Stream DataStream
{
    get { return new MemoryStream(Encoding.UTF8.GetBytes(Data)); }
}
```

The sample code demonstrates deserializing XML data using `XmlDocument`, `XDocument`, XML serialization, and data contract serialization, as well as a modified technique I call `XDocumentAlternate`. Each of these techniques is executed using a simple loop that iterates a list of delegates, as shown in Listing 3.5.

Listing 3.5: `SerializationSampler` `Main` method

```
delegate ImageItem DemoXml();

static void Main(string[] args)
{
    List<DemoXml> examples = new List<DemoXml>()
      { UseXmlDocument,
        UseXDocument,
```

```
            UseXDocumentAlternate,
            UseXmlSerialization,
            UseDataContractSerialization};
        foreach (DemoXml demo in examples)
        {
            Console.WriteLine("{0}: {1}{2}",
                demo.Method.Name,
                Environment.NewLine,
                demo());
        }
    }
```

XmlDocument

`XmlDocument` has a long history with the .NET Framework—it has been part of the framework since the beginning. To use `XmlDocument`, you load the document with XML that might come from a number of sources: a file, a URL, a string, or a stream. You can also populate an `XmlDocument` from scratch using code, creating each element by hand.

To load XML data into an `XmlDocument` instance, you would use its `LoadXml` to load an XML string or its `Load` method to use XML data encoded within one of the other sources (file, stream, and so on). After the document is loaded, you can extract the data; a number of mechanisms exist to help extract data to be placed in objects for code manipulation. An obvious approach is to iterate over the contents of the document, looking for nodes with known element names. Unfortunately, this technique can be fairly code heavy. Another technique involves writing code that knows where the element should be located within the document and then using indexers to grab the text of that node. This technique winds up being very fragile because the layout of the document might change over time. Adding or deleting items from the document can cause specific indexes to change and code to malfunction accordingly.

My preferred technique to extract data from an `XmlDocument` involves using XPath expressions and `System.Xml.XmlReader`. `XmlDocument` has a method named `SelectSingleNode` that accepts an XPath expression and an optional `XmlNamespaceManager`. XPath is a rich language for querying XML. The most common queries involve looking for elements or attributes with known names. Each element and attribute in the XML has an XML

qualified name (QName). To choose a specific element or attribute, you need to know the item's QName. The QName is consistent no matter the XML namespace being used.

`ImageItem` has a namespace of http://www.scottseely.com/RESTBook/2008. The `ImageItem` XML shown in Listing 3.4 establishes the default namespace with this markup:

```
xmlns="http://www.scottseely.com/RESTBook/2008"
```

This allows us to identify `ImageItem` content without any prefixes for our convenience. The declaration could have just as easily been

```
xmlns:restbook="http://www.scottseely.com/RESTBook/2008"
```

If this was the case, the `ImageItem` XML would have to be changed to prefix all element tags with `restbook:`. The resulting XML, regardless of how the namespace was declared, would be considered equivalent between the two documents. What does this digression have to do with anything? Well, if we want to ask for the caption, description, or other elements contained within the XML document, we have to present a name that is meaningful via the XPath expression. These prefixes do not have to match what is in the actual XML document because XPath expressions operate against the *infoset* representation of the document, not the string representation.

> **NOTE**
>
> One cannot talk about XML without talking about the infoset. In a nutshell, the infoset representation concentrates on the tree-based structure of the XML document rather than what data the tree structure contains. Everything in an infoset is some type of XML node: attribute, text, element, whitespace, comment, processing instruction, and so on. Elements and attributes have QNames. Using this, specification authors found it easier to write specifications because they were no longer worrying about XML representation. Developers have since used this to introduce new serialization schemes beyond XML 1.0 (text). Today, many binary serializers have been created to reduce the bulkiness issues associated with XML.

> To read more about infosets, consider going to the source specification at www.w3.org/TR/xml-infoset/. O'Reilly also has a great site with articles and information about XML in general at www.xml.com.

Knowing what we do about infosets, we need a crash course in XPath. XPath expressions can get complicated. However, most queries are fairly straightforward: Select all nodes with this name, select the first node with this name, only select nodes at this location relative to the root, and so forth. Here is some of the basic XPath syntax:

- `//`: Used at the start of an expression to select any node with a given name. Example: `//id` selects all nodes named `id`.
- `/`: When used as a single character, denotes that the expression starts at the node being queried. Example: `/ImageItem/Caption` selects all nodes named `Caption` that are children of a node named `ImageItem`.
- `text()`: An element may contain a text node. The expression `text()` allows the query to select the child text nodes. Example: `/ImageItem/Caption/text()` selects all text contents within nodes named `Caption`.
- *`@attribute name`*: Elements may contain attribute tags. The `@` syntax says that the name that follows in the XPath expression is an attribute. Example: If we have XML such as `<foo @bar="some value" />` and the XPath expression `//foo/@bar/text()`, then the result of the expression is `some value`.

XPath allows for other options too. You can perform logical tests such as `=`, `!=`, `<`, `>`, and more. You can also check for string contents and other things to filter the results to a finer degree. The articles at xml.com provide a rich source of information for how to handle most of these deeper issues.

For our example, we will want to parse all nodes with the following QName:

```
http://www.scottseely.com/RESTBook/2008:ImageItem
```

To identify the prefix we will map to the namespace, we use a class called `System.Xml.XmlNamespaceManager`. By setting up the name correctly, we can now ask for all `ImageItem` elements:

```
XmlNamespaceManager nsMgr = new XmlNamespaceManager(doc.NameTable);
nsMgr.AddNamespace("item", "http://www.scottseely.com/RESTBook/2008");
XmlNode node = doc.DocumentElement.SelectSingleNode(
    "/item:ImageItem", nsMgr);
```

At this point, we can use additional XPath expressions to iteratively select child nodes, or we can use a `System.Xml.XmlReader` to read the remaining data directly. The choice is up to you. The `XmlReader` approach has the benefit of being able to spin through the node data fairly quickly, whereas the XPath approach is more deliberate. The `XmlReader` approach is presented only for completeness because both approaches are useful. An `XmlReader` tends to require more testing effort as changes to document structure can introduce bugs fairly easily, which is something to keep in mind.

Using an `XmlNodeReader`, we can then look at each element individually. Each element has two name properties: `Name` and `LocalName`. The `Name` represents the QName for the element. The `LocalName` represents the simple form of the element tag name—essentially a non-namespace version. If your expectation is that the names within a given node will not vary, and that name collisions will not occur, you can safely use `LocalName`. Otherwise, if names in a node might be reused across XML namespaces (for example, *address* could mean *memory address* or *street address*), use the `Name` property instead. The `XmlNodeReader` then has helper methods to read common data types: strings, numbers, and Boolean values. Using this knowledge, we can now read the XML using an `XmlDocument`. The code I've provided introduces a helper method, `ReadToText(XmlReader)`, that advances the reader to the (child) text node. This particular function exists to make sure that we don't skip over other elements by advancing the reader too far using functions like `ReadElementContentAsString`. `ReadElementContentAsString` will advance the reader to the next `Element` node. A call to `XmlReader.Read()`, as in the `while` loop shown in Listing 3.6, would march past too many elements without `ReadToText` to slow it down.

Listing 3.6: Reading and consuming XML using `XMLDocument` and XPath

```
static void ReadToText(XmlReader reader)
{
    while (reader.Read())
    {
        if (reader.NodeType == XmlNodeType.Text)
        {
            // Break when we hit a Text node.
            break;
        }
    }
}

private static ImageItem UseXmlDocument()
{
    ImageItem retval = new ImageItem();

    XmlDocument doc = new XmlDocument();
    doc.Load(DataStream);

    // Set up the Infoset mapping information.
    XmlNamespaceManager nsMgr = new XmlNamespaceManager(doc.NameTable);
    nsMgr.AddNamespace("item",
       "http://www.scottseely.com/RESTBook/2008");

    // Only pick the first node named ImageItem.
    // Use SelectNodes to pick ALL.
    XmlNode node = doc.DocumentElement.SelectSingleNode(
        "/item:ImageItem", nsMgr);

    // Look at each node in the document.
    XmlNodeReader reader = new XmlNodeReader(node);
    while (reader.Read())
    {
        // Ignore anything that isn't a start element.
        if (reader.NodeType == XmlNodeType.Element)
        {
           try
           {
               switch (reader.LocalName)
               {
                   case "Caption":
                       ReadToText(reader);
                       retval.Caption = reader.ReadContentAsString();
                       break;
                   case "Description":
                       ReadToText(reader);
```

continues

Listing 3.6: Continued

```
                        retval.Description =
                           reader.ReadContentAsString();
                        break;
                    case "ImageBytes":
                        ReadToText(reader);
                        retval.ImageBytes = Convert.FromBase64String(
                           reader.ReadContentAsString());
                        break;
                    case "ImageId":
                        ReadToText(reader);
                        retval.ImageId = new
                           Guid(reader.ReadContentAsString());
                        break;
                    case "ImageUrl":
                        ReadToText(reader);
                        string tempUri = reader.ReadContentAsString();
                        if (Uri.IsWellFormedUriString(tempUri,
                           UriKind.Absolute))
                        {
                            retval.ImageUrl = new Uri(tempUri);
                        }
                        break;
                    case "PublicImage":
                        ReadToText(reader);
                        retval.PublicImage =
                           reader.ReadContentAsBoolean();
                        break;
                    case "UserName":
                        ReadToText(reader);
                        retval.UserName = reader.ReadContentAsString();
                        break;
                }
            }
            catch
            {
                // Parse failure—do nothing
            }
        }
    }
    return retval;
}
```

As you can see from Listing 3.6, that is an awful lot of code to translate the XML into something readable. If this RESTful services thing is ever going to take off, there needs to be a simpler solution. There is, and we will

get to it. For this very reason, we will be skipping how to serialize data back out. If you are writing the object, you have more control and better options than manipulating XML documents directly for most situations, so it doesn't make sense to even describe how that's done here. We'll save that topic for later in the chapter when we look at the XML serializers.

XDocument

If you are using .NET 3.5 or later, you have something truly wonderful at your disposal. You have Language Integrated Query (LINQ). LINQ is mostly syntactic sugar when represented in a .NET language like C# or VB.NET. The actual generated code is very procedural. But, from a developer point of view, the expressions are declarative. Fortunately, the technology has been written about enough that it is unnecessary to promote its use here. If you do any work with LINQ, you have to go out and buy the `LINQ Pocket Reference` by Joseph and Ben Albahari, ISBN 978-0-596-51924-7, from O'Reilly. It's a tiny 160-page book that will literally fit into the back pocket of your jeans.

Within LINQ, there are all sorts of variants. LINQ to objects lets you execute nifty queries over collections. LINQ to SQL generates SQL queries. And perhaps not surprisingly, LINQ to XML operates over XML documents. Specifically, LINQ to XML (XLINQ) operates on types known as `System.Linq.Xml.XDocument`. `XDocument` knows how to read and write using `System.Xml` data types: `XmlReader` and `XmlWriter`. As a result, you can use the `XmlReader` code from the preceding section to do your parsing. So we will skip that and move on to other concepts, like queries.

With XLINQ, we still need to think in terms of infosets. When we ask for an element with a particular name, we ask for the element by QName. The XLINQ type representing the QName is named `System.Linq.Xml.XName`. You create an `XName` by concatenating a `System.Linq.Xml.XNamespace` with a string representing the element name. To create the `XNamespace` for `ImageItem` and then create the appropriate `XName`, use the following bit of code:

```
XNamespace ns = "http://www.scottseely.com/RESTBook/2008";
XName imageItemName = ns + "ImageItem";
```

Most of the types we deal with in LINQ are `IEnumerable` types. In our case, we will normally want only the first item in that list (and, in fact, the XML document we're working with contains only one element). .NET 3.5 implements an extension method for `IEnumerable<T>` called `First` that provides this capability quite handily. Using all this basic knowledge, the code to parse the `XDocument` into an `ImageItem` becomes simpler. To select a named node within the current node, you pass the `XName` of the target node to the current node's `Elements(XName)` method. You can then select the first element from that list and pick off the `Value` of that item, as demonstrated in Listing 3.7.

Listing 3.7: Reading and consuming XML using `XDocument` and XLINQ

```
static ImageItem UseXDocument()
{
    XDocument doc = XDocument.Load(new XmlTextReader(DataStream));
    XNamespace ns = "http://www.scottseely.com/RESTBook/2008";
    var items = from imageItemNode in doc.Elements(ns + "ImageItem")
        select new ImageItem()
        {
            Caption = imageItemNode.Elements(ns +
               "Caption").First().Value,
            Description = imageItemNode.Elements(ns +
               "Description").First().Value,
            ImageBytes = Convert.FromBase64String(
               imageItemNode.Elements(ns + "ImageBytes").
                  First().Value),
            ImageId = new Guid(imageItemNode.Elements(ns +
               "ImageId").First().Value),
            ImageUrl = new Uri(imageItemNode.Elements(ns +
               "ImageUrl").First().Value),
            PublicImage = bool.Parse(imageItemNode.Elements(ns +
               "PublicImage").First().Value),
            UserName = imageItemNode.Elements(ns +
               "UserName").First().Value
        };
    return items.First();
}
```

The `XDocument` code has some advantages over the `XmlDocument` version. First and foremost, it is a lot shorter. Shorter code typically leads to fewer bugs since most developers implement code with a consistent ratio

of bugs to lines of code. This ratio is not intentional—as humans we just tend to make mistakes at a steady pace. An issue with the code in Listing 3.7 is that it has very little in the way of error handling. If the `ImageUrl`, `ImageId`, `ImageBytes`, or `PublicImage` items fail to parse, the code fails for all items. If we would rather load the `ImageItem` and leave fields blank on failure, the code could be rewritten as shown in Listing 3.8 to provide for deserialization failover.

Listing 3.8: Reading and consuming XML using `XDocument` and XLINQ with failover

```
static ImageItem UseXDocumentAlternate()
{
    XDocument doc = XDocument.Load(new XmlTextReader(DataStream));
    XNamespace ns = "http://www.scottseely.com/RESTBook/2008";
    var items = from imageItemNode in doc.Elements(ns + "ImageItem")
        select new
        {
            Caption = imageItemNode.Elements(ns +
               "Caption").First().Value,
            Description = imageItemNode.Elements(ns +
               "Description").First().Value,
            ImageBytes = imageItemNode.Elements(ns +
               "ImageBytes").First().Value,
            ImageId = imageItemNode.Elements(ns +
               "ImageId").First().Value,
            ImageUrl = imageItemNode.Elements(ns +
               "ImageUrl").First().Value,
            PublicImage = imageItemNode.Elements(ns +
               "PublicImage").First().Value,
            UserName = imageItemNode.Elements(ns +
               "UserName").First().Value
        };
    ImageItem retval = new ImageItem();
    foreach (var item in items)
    {
        retval.Caption = item.Caption;
        retval.Description = item.Description;
        retval.UserName = item.UserName;
        bool publicImage;
        if (bool.TryParse(item.PublicImage, out publicImage))
            retval.PublicImage = publicImage;

        if (Uri.IsWellFormedUriString(item.ImageUrl, UriKind.Absolute))
            retval.ImageUrl = new Uri(item.ImageUrl);
```

continues

Listing 3.8: Continued

```
            try
            {
                retval.ImageId = new Guid(item.ImageId);
            }
            catch { }
            try
            {
                retval.ImageBytes =
                   Convert.FromBase64String(item.ImageBytes);
            }
            catch { }
            break;
        }
        return retval;
    }
```

Again, this all depends on whether you want things to fail whenever bad input data is encountered. Most of the time, invalid data that appears anywhere in the object implies that you do not want to continue deserializing that XML stream. When that is the case, the short version of this code is completely appropriate. When failover is called for, however, you need the extra code shown in Listing 3.8.

`XmlSerializer`

Wouldn't it be great if you could just tell the runtime what your object looked like and then it could figure out how to read the XML and populate that object for you? As it happens, this is possible. To do this, we can go to one of my favorite .NET namespaces: `System.Xml.Serialization`. In the .NET world, it is the king of XML reading and writing within the bits shipped with the framework. It handles attributes, special serialization, and schema generation. Most of this work is directed with simple attributes placed on public classes, fields, and properties. And it implements the parts of the XML Schema specification that map into .NET.

> **NOTE**
>
> As it happens, .NET doesn't implement the full suite of schema constructs identified in the XML Schema specification. If you create a schema using a tool outside of .NET, you could be asking for trouble if you later try to incorporate those schemas into .NET code. For example, Altova has tools with XML Spy that implement all far corners of the XML Schema Document specification and provide features not found in .NET. If you need to support facets and other fancy features of XSD, you need to go beyond what Microsoft ships with the framework. In practice, this kind of specialization is needed only for sophisticated XML processing in a small set of scenarios.

The most common attributes used in the `System.Xml.Serialization` are listed here:

- `XmlElementAttribute`: Declares the XML element name and namespace for a given property when that property appears in an XSD or XML document.
- `XmlAttributeAttribute`: Declares the XML attribute name and namespace for a given property when that property appears in an XSD or XML document.
- `XmlTypeAttribute`: Declares the name and namespace for a given class or enum within an XSD.
- `XmlRootAttribute`: Declares the name and namespace for a given class or enum when that data type is used as the root of an XML document.
- `XmlEnumAttribute`: Declares the name of an enumeration value when that enum appears in an XSD or XML document.
- `XmlIgnoreAttribute`: Prevents serialization of this member. By default, all public members are serialized.

- `XmlArrayAttribute`: Allows the developer to control the names used when serializing arrays of items.
- `XmlArrayItemAttribute`: Collaborates with `XmlArrayAttribute`. `XmlArrayItemAttribute` allows for a given collection to contain more than one type of object.

This information is consumed by another class named `System.Xml.XmlSerializer`. `XmlSerializer` learns how to convert between XML and .NET types by reading these attributes via reflection. The initial instance of an `XmlSerializer` for a given type is expensive to build in terms of time, so quite often an instance is created early and held for the duration of the application's lifetime. After you have the serializer in hand, however, you can read XML into objects and write objects as XML into a stream or file with a single method call.

Using this knowledge, we need to decorate the `ImageItem` class with the right set of attributes to drastically reduce the amount of code we have to write to eventually serialize and deserialize it. The class should be decorated with an `XmlRootAttribute` and all properties with `XmlElementAttribute`. The code in Listing 3.9 demonstrates these attributes in action. I could have written even less code by not filling in the `ElementName` information as the XML element name defaults to the property name, but I wanted to demonstrate how you establish the tie between the property name and the corresponding XML element. If you want to reduce the size of the serialized XML stream, one way to do it is to provide very short XML element names, and `XmlElementAttribute` is the tool you'd use for this. We do run into one snag, though: `ImageItem.ImageUrl` is of type `System.Uri`. `Uri` does not have a parameterless constructor, which is a requirement for `XmlSerializer`. Because of this, `XmlSerializer` cannot automatically read and write `Uri` values. There is a workaround: We create a new `String` property, `ImageUrlString`. Within that property, we read and write the `ImageUrl` for everyone else to use. It's a small change to the class but results in simpler code overall than any mechanism seen so far.

Listing 3.9: `ImageItem` **with XML serialization attributes applied**

```
[XmlRoot(Namespace="http://www.scottseely.com/RESTBook/2008")]
public class ImageItem
{
    [XmlElement(ElementName = "Caption")]
    public string Caption { get; set; }

    [XmlElement(ElementName = "Description")]
    public string Description { get; set; }

    [XmlElement(ElementName = "ImageBytes")]
    public byte[] ImageBytes { get; set; }

    [XmlElement(ElementName = "ImageId")]
    public Guid ImageId { get; set; }

    [XmlIgnore]
    public Uri ImageUrl { get; set; }

    [XmlElement(ElementName = "ImageUrl")]
    public string ImageUrlString
    {
        get { return ImageUrl == null ? null : ImageUrl.ToString(); }
        set
        {
            Uri tempUri = null;
            if (Uri.TryCreate(value, UriKind.Absolute, out tempUri))
                ImageUrl = tempUri;
            else
                ImageUrl = null;
        }
    }
}
```

With the data type decorated with all these different attributes, what code do we implement to read an `ImageItem` now? Happily, deserialization distills down to two lines of code:

```
static ImageItem UseXmlSerialization()
{
    XmlSerializer ser = new XmlSerializer(typeof(ImageItem));
    return (ImageItem)ser.Deserialize(DataStream);
}
```

As you can see, we are no longer parsing the XML ourselves. This means that potential bugs in the code we write become less likely—less code means fewer defects. The great thing about `XmlSerializer` is that it handles elements being out of order, missing elements, and so on. There is one more serialization mechanism that has an extra benefit if you can use this mechanism: blindingly fast speed. Let's look at that topic next.

`DataContractSerializer`

`System.Runtime.Serialization.DataContractSerializer` is part of a family of serializers introduced with .NET 3.0 that all inherit from `System.Runtime.Serialization.XmlObjectSerializer`. These serializers specialize in reading and writing objects to XML, JSON, and other formats. In general, these serializers read and write XML faster than anything else on the .NET platform. When looking specifically at `DataContractSerializer`, however, understand that the speed comes at a price: `DataContractSerializer` does not handle XML attributes and it does not handle arbitrarily ordered elements. `DataContractSerializer` handles serialization of the following types of data:

- Classes that implement `System.Runtime.Serialization.ISerializable`.
- Types marked with the `System.SerializableAttribute`.
- Primitive types and enumerations. These types are implicitly serializable.
- Types marked with `System.Runtime.Serialization.DataContractAttribute`.
- Undecorated objects. This feature, new in .NET 3.5 SP1, allows for serializing objects that have no special attribution. In this case, only types with public, default constructors can be serialized. The `DataContractSerializer` will serialize only public fields and properties.

Like `XmlSerializer`, `DataContractSerializer`'s explicit serialization model relies on attributes.

- `DataContractAttribute`: Indicates that the data type has explicit serialization rules. Using this attribute, one can set the way the data type is represented in XML with a name and namespace. Note that the namespace information might be ignored by other serializers, such as the `System.Runtime.Serialization.Json.DataContractJsonSerializer`, because JSON has no equivalent of XML namespaces.
- `DataMemberAttribute`: Indicates that the field or property is read/write. Besides the usual name and namespace settings, this attribute lets the developer express the requested relative order of the element within any serialization scheme.
- `CollectionDataContractAttribute`: Used to indicate how a collection should be serialized. You can set the name of the collection, the name of elements within the collection, and the XML namespace associated with the collection.
- `EnumMemberAttribute`: Allows you to set the names and values of elements within an enumeration.

We are primarily interested in properties and related fields that use the explicit serialization rules, which is to say have serialization attributes applied. The rest of this section focuses on the explicit aspects alone. Elements are ordered following these rules:

1. All elements within a type are ranked according to the value of the `DataMemberAttribute.Order` property.
2. Within an `Order` value, members are serialized and deserialzed in alphabetical, ascending order. By default, the value of `Order` is 0. Members are always sorted first by `Order`, then by alphabetical order within a given `Order` value. Upon deserialization, if a member is out of order, it will appear with its default value (typically a 0 or null) within the deserialized object.

> **NOTE**
>
> If you need to use `DataContract` and must accept out-of-order parameters, consider using the `DataContractJsonSerializer` and the JSON format instead. You can also support both `XmlSerializer` and `DataContractJsonSerializer` on the same data type if you need to handle unordered XML.

Assuming that you are guaranteed the order of the elements, you can use the code shown in Listing 3.10 to read the XML information. You'll probably notice that this looks very similar to what we did for `XmlSerializer`. However, a benefit of `DataContractSerializer` is that it doesn't need a public, default constructor to create an object. As a result, it can instantiate `System.Uri` without the workaround needed for `XmlSerializer`.

Listing 3.10: `ImageItem` **with data contract attributes applied**

```
[DataContract(Namespace = "http://www.scottseely.com/RESTBook/2008")]
public class ImageItem
{
    [DataMember(Name = "Caption")]
    public string Caption { get; set; }

    [DataMember(Name = "Description")]
    public string Description { get; set; }

    [DataMember(Name="ImageBytes")]
    public byte[] ImageBytes { get; set; }

    [DataMember(Name = "ImageId")]
    public Guid ImageId { get; set; }

    [DataMember]
    public Uri ImageUrl { get; set; }

    [DataMember(Name = "PublicImage")]
    public bool PublicImage { get; set; }

    [DataMember(Name = "UserName")]
    public string UserName { get; set; }
}
```

As with `XmlSerializer`, we wind up with a fairly short code snippet to deserialize an `ImageItem` object:

```
static ImageItem UseDataContractSerialization()
{
    DataContractSerializer dcs = new
        DataContractSerializer(typeof(ImageItem));
    return (ImageItem)dcs.ReadObject(DataStream);
}
```

Writing the object is a matter of calling `DataContractSerializer`'s `WriteObject` method. With all this information in place, we are ready to start talking about building actual applications.

Working with WinForms

There are a lot of WinForms applications out there. These applications will be improved, extended, and maintained for a long time. Some of those enhancements and changes might need to make use of RESTful services. If they do, you will need to familiarize yourself with the following concepts:

- **Using `System.Net.WebRequest/WebResponse`**: This pair of classes provides the best overall developer tools for communicating with other HTTP-based services, although you can use the derived classes `HttpWebRequest` and `HttpWebResponse` as well. You also have at your disposal `System.Net.WebClient`, but I find I prefer `WebRequest` and `WebResponse`. I like the greater control I have over the request/response process when using `WebRequest` and `WebResponse`. For example: Credentials frequently need to be passed between the service and the client. The normal `WebClient` behavior issues a challenge/response pair of HTTP messages for any secured resource. `WebRequest`, on the other hand, allows you to set its `PreAuthenticate` property and save a round trip.
- **Sharing the UI thread**: .NET (actually, Windows itself) gives you only one thread for the UI. If you use that single thread for synchronously accessing a RESTful service, the user will feel the application is unresponsive.
- **Using asynchronous programming techniques**: Learn to use `System.Threading.ThreadPool`, the `BeginInvoke` method implemented by many data types, and what `System.IAsyncResult` is used

for. Something to keep in mind is that it's the client that implements asynchronous request processing. To the service, a synchronous client request is no different than an asynchronous one. But your application users will definitely notice the difference.

Our example here revolves around a simple WinForms application that interacts with the Photo Service described at the beginning of this chapter. This application does not include mechanisms for storing information to be synchronized later. It only shows how to connect with the Photo Service. The code for this example can be found in the `WinFormPhotoClient` project included with this chapter's sample solution.

This chapter also comes with a copy of the Photo Service presented later, in Chapter 8. To install the service, please do the following:

1. Open the Chapter 3 Solution file.
2. Right-click on PhotoWeb and select Publish. Publish the Web application to http://localhost/PhotoWeb.
3. Create the PhotoWeb database using the scripts found in the PhotoWebDb database project. You'll find one script for creating the database and another for creating all the associated tables. The database creation script, `CreatePhotoWebDb.sql`, assumes you're using SQL Express and creates the `.mdb` file accordingly. If you're not using SQL Express, simply edit the script and store the resulting `.mdb` file in an authorized location or create the database using the graphical tools found in SQL Server Management Studio, giving the new database the name PhotoWebDb.

Let's take a quick look at *Photo Client*, the WinForms sample application that demonstrates RESTful service access. When the Photo Client application executes, the user can either log in or create a new user account. Figure 3.1 shows the initial application user interface.

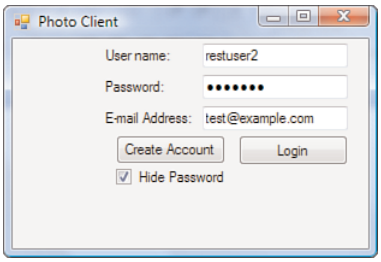


FIGURE 3.1: WinForms startup screen

Because users might want to see the password as they type it in, the application allows them to show or mask the text in the password input `TextBox` with a password-hiding feature they can turn on or off. If the login information is new, which is to say unknown to the service, users can create a new service account from the Photo Client application. Pressing the Login button only caches the credentials within the application. The credentials are not actually submitted until the user begins to interact with the Photo Service. If the user clicks Create Account, the following tasks need to take place:

- The application needs to ask the Photo Service for a new user account, providing the username, password, and e-mail address the user provided. The e-mail address might seem like an odd piece of information, but the service can use this to help the user reset the password in the event that the user forgets the password.
- The application waits for a response indicating that the new username was accepted and the password met the password strength requirements. If the username is already in use, or if the password does not meet password strength requirements, the user is notified and can resubmit new information.

These tasks are implemented in the sample's `LoginControl.cs` file. The code first constructs a new URI. Because the application knows exactly

what user to create, the code uses a `PUT` request (versus a `POST` request). As service application developers, we know that these are the service's rules for creating a new user:

- HTTP request verb: `PUT`
- URI template:
  ```
  http://localhost/PhotoWeb/UserManager.svc/CreateUser/
  ➥{username}?email={emailAddress}
  ```
- Request body: XML-serialized password. Because the password is issued in the HTTP entity body as clear text, you should use transport-level security, such as secure sockets (HTTPS), to protect it.

Although not a requirement, most services provide some form of documentation that tell you, the application developer, how to interact with the service. In this case, you would be given the URI template, the XML schema (or format) for the new account request, and so forth. If you're writing a service, you should provide this information.

Anyway, to call the RESTful service and create the new service user account, you use the `System.Net.WebRequest` class. You never create an instance of `WebRequest` directly, however. Instead, you use `WebRequest.Create`, providing an instance of `System.Uri` that represents the URI of the request. `Create` looks up the right type of request to create based on the URI scheme. In our case, the scheme is `http:` or `https:` (versus `ftp:` or `gopher:`). After the request object is returned, we can change existing HTTP header values, add new HTTP headers, establish the HTTP method, and assign other values necessary to issue the request over the network. If we do nothing special, the UI thread will be using the request object, but requests and associated responses over the network take time. To allow the UI to remain responsive while the user waits for the service response, the application should use asynchronous service request techniques for any RESTful service calls it makes. The asynchronous request for creating a new user is shown in Listing 3.11. You can tell that the request is made asynchronously because `BeginGetResponse` is used (its cousin `GetResponse` is its synchronous counterpart).

Listing 3.11: Photo Client asynchronous user account creation

```
private void btnCreateAccount_Click(object sender, EventArgs e)
{
    // Construct the call.
    Uri callUri = Utility.CreateUri(ServiceType.Manager,
      string.Format("CreateUser/{0}?email={1}",
      HttpUtility.UrlEncode(txtUserName.Text),
      HttpUtility.UrlEncode(txtEmailAddress.Text)));

    // Create the WebRequest
    WebRequest request = WebRequest.Create(callUri);
    request.Method = "PUT";
    Stream stream = request.GetRequestStream();
    request.ContentType = "application/xml";
    DataContractSerializer stringSer = new
        DataContractSerializer(typeof(string));
    stringSer.WriteObject(stream, txtPassword.Text);
    stream.Close();
    // Asynchronously wait for the response.
    request.BeginGetResponse(new AsyncCallback(CreateAccountResponse),
       request);

    // Disable the UI while waiting.
    this.Enabled = false;
}
```

Sometime later, the Photo Service responds, letting us know of any issues or if the account creation was successful. For any issues, we need to notify the user. If the account creation was successful, we let the user know that as well and ask if the user would like to log in.

A successful response will have an associated `200`-level status code. But how will we know if there were issues with the request? As it happens, `300`-level or higher response status codes will cause `WebRequest` to throw a `WebException`. `WebException` contains information about which HTTP response code was returned, as well as any information present in the response stream. We can use these pieces of information to find out what the server told us went wrong.

But hold on a minute…we're using an asynchronous request pattern (we initiated the request using `BeginGetResponse`). Whenever we receive responses via these asynchronous calling patterns, we have no knowledge about which application thread is being used for the response. It is almost

always a safe bet to assume that we aren't on the original UI thread. However, any code that updates the user interface *must* be invoked using the UI thread. This model is strictly enforced by the framework (as well as Windows) and serves to prevent race conditions while updating the UI.

This necessarily means we have to have some way to switch thread contexts and gain access to the UI thread. How is this accomplished? To execute code on the UI thread, one technique is to invoke a parameterless delegate. In these cases, I frequently prefer to pass along an anonymous delegate: The code is frequently short and easier to understand when viewed in the context of when the delegate is actually called. You can call this delegate method via the `System.Windows.Forms.Control.Invoke` or `BeginInvoke/EndInvoke` pair. Either mechanism will make sure that your code runs on the UI thread, allowing your application to update the screen. For short functions, I typically call `Invoke` since it involves less code and I'm generally not too concerned about a short pause on the background (asynchronous) thread. The callback for account creation, using `Control.Invoke` and an anonymous delegate, is shown in Listing 3.12.

> **NOTE**
>
> Failing to modify the state of the window controls via the original creating UI thread usually results in an `AsyncCallbackException` (or in some cases a `COMException`). This is by design and has been so since the first version of Windows. It's not something we can pick and choose to deal with—we must deal with it when creating multithreaded Windows applications, whether .NET-based or not.

Listing 3.12: Asynchronous response handling

```
void CreateAccountResponse(IAsyncResult result)
{
    // Capture the request from the state.
    WebRequest request = (WebRequest)result.AsyncState;
    try
    {
        // Ask for the response. Any exceptions will get thrown here.
        WebResponse response = request.EndGetResponse(result);
        this.Invoke((MethodInvoker)delegate
```

```
            {
                // Lack of exceptions means that we got a 2xx back.
                if (DialogResult.Yes == MessageBox.Show(this,
                    "Account created. Do you want to log in?",
                    "Success", MessageBoxButtons.YesNo,
                    MessageBoxIcon.Information))
                {
                    btnLogin_Click(null, EventArgs.Empty);
                }
            });
        }
        catch (WebException we)
        {
            // Let the user know what happened.
            HttpWebResponse httpResponse = (HttpWebResponse)we.Response;
            if (httpResponse.StatusCode == HttpStatusCode.Conflict)
            {
                // Conflict means the name was in use.
                this.BeginInvoke((MethodInvoker)delegate
                {
                    MessageBox.Show(this,
               "This username is already in use. Cannot create a new account.",
               "Username in use", MessageBoxButtons.OK,
               MessageBoxIcon.Exclamation);
                });
            }
            else
            {
                // Something else happened.
                this.Invoke((MethodInvoker)delegate
                {
                    MessageBox.Show(this, string.Format(
                       "Code: {0}\r\nDescription: {1}.",
                        httpResponse.StatusCode.ToString(),
                        httpResponse.StatusDescription),
                        "Failed to create a new user.",
                        MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
                });
            }
            Debug.WriteLine(we.ToString());
        }
        this.Invoke((MethodInvoker)delegate
        {
            // Re-enable the UI.
            this.Enabled = true;
        });
    }
```

After we have a valid account, we can log in. The application simply caches the username and password values in memory for use when contacting the Photo Service. Once authenticated, and if we have used the service before, we are greeted with the form shown in Figure 3.2. Note that the mechanisms used to pass authentication credentials use HTTP Basic Authentication for this service, which passes the username/password in clear text (as with account creation, which placed the password in the HTTP entity body). If your service uses HTTP Basic Authentication to authenticate users, make sure that this is done over a secure, HTTPS channel. Otherwise, the credentials can be viewed by any router, proxy server, or machine between the source and destination of the HTTP request.

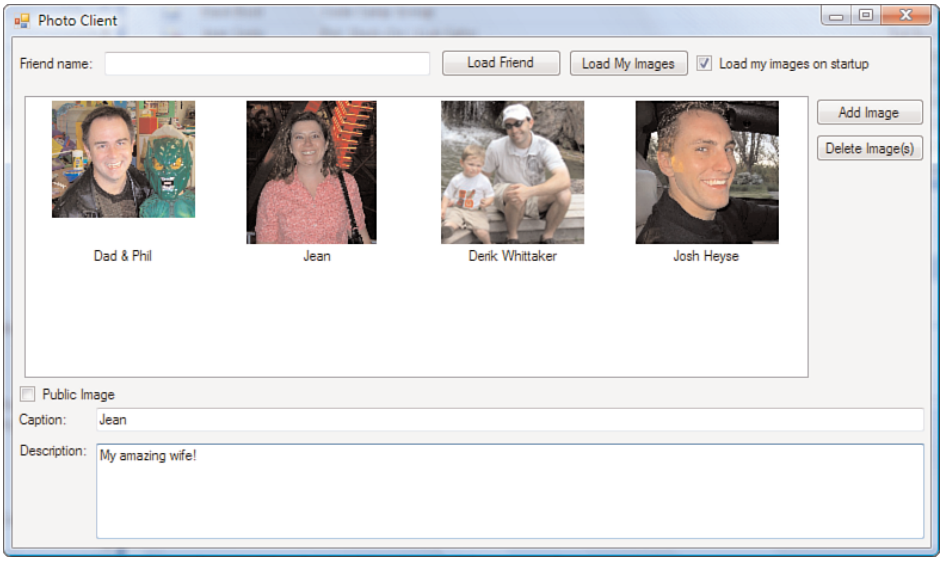


FIGURE 3.2: The Photo Client details view

To display images coming from the service, a couple of things have to happen. First, the user needs to authenticate with the service using HTTP Basic Authentication (this will be the case for each and every RESTful service invocation). Normally this involves one or more trips to the server, but in our case we'll preload the credentials in `WebRequest` by setting its `PreAuthenticate` property to `true`. You'll learn much more about HTTP Basic Authentication and its relationship to RESTful services in Chapter 6, "Building REST Services Using IIS and ASP.NET."

Then we make a request for images. This request asks for all the images associated with a particular user. If the authenticated user and the user whose images are being requested are the same, the returned list of images includes all images, both public and private. If the users differ, the returned list includes only public images associated with the requested user. (Actually, the list includes only the URLs that the application will use to fetch the images. The application needs to send out another batch of requests to actually retrieve the image bytes.) Because we are using a RESTful service, we treat each interaction as a brand-new request. This means that each request for secured information, as for the list of pictures, needs to include authentication information.

How does authentication work for us? `WebRequest` has a property named `Credentials` of type `System.Net.ICredentials`. This interface defines a lookup table of endpoints, authentication mechanisms, and username/password combinations. Fortunately, `System.Net.CredentialCache` implements the interface and handles our needs quite well. Our code makes use of a helper class, `PhotoShared.Utility`, in the PhotoShared project. This class holds a common `CredentialCache` and a shared method to handle setting up our `WebRequest` in a uniform way. When we click the Login button (refer to Figure 3.1), the small amount of code shown in Listing 3.13 (contained within `LoginControl.cs`) executes.

Listing 3.13: Photo Client Login button click handler

```
private void btnLogin_Click(object sender, EventArgs e)
{
    // Clear any stored credentials and store the new ones.
    Utility.ClearCredentialCache();
    NetworkCredential credential = new
            NetworkCredential(txtUserName.Text,
            txtPassword.Text);
    Utility.CredentialCache.Add(new
        Uri(Properties.Settings.Default.PhotoService),
        "Basic", credential);
    if (this.LoggedIn != null)
    {
        // Raise the LoggedIn event
        this.LoggedIn(this, EventArgs.Empty);
    }
}
```

Whenever the application needs to send an authenticated request, it can use the `Utility` class's `CreateRequest` method, shown in Listing 3.14. This utility method encapsulates and standardizes the code necessary for setting the credentials. It also makes sure that any outgoing requests send credentials on the first request—alleviating a possible challenge/response pair of messages between the server and the client as previously mentioned.

> **NOTE**
>
> Some examples later in the book, such as the client application for Chapter 6, don't use the credential cache as implemented here but rather insert the credentials directly into the HTTP request. This is done for illustrative purposes, and practically speaking is functionally equivalent to the technique used in this chapter. Feel free to use whichever technique you prefer.

Listing 3.14: The `CreateRequest` method

```
public static WebRequest CreateRequest(Uri uri)
{
    WebRequest retval = WebRequest.Create(uri);
    retval.Credentials = CredentialCache;
    retval.PreAuthenticate = true;
    return retval;
}
```

To add an image to the collection on the server, we use the HTTP `POST` method. Why `POST`? In our case, the server controls the issuance of new photo identifiers. Clients have no idea what identifier the server will assign to the new image. The new identifier might be a string, an integer, or something else—to service consumers it is just an opaque identifier. When this situation arises, from Chapter 2, "The HyperText Transfer Protocol and the Universal Resource Identifier," we know that RFC 2616 tells us the HTTP `POST` method is appropriate. After the data is posted, the client waits for the response so that it can discover the identifier for the newly added image.

Our user interface allows the user to select files using the `System.Windows.Forms.OpenFileDialog`. When the user clicks Open, the list of selected files is returned to the application. This list is then passed to a

function named `AddImages`. `AddImages` is executed in a background thread obtained from the .NET thread pool:

```
ThreadPool.QueueUserWorkItem(new WaitCallback(AddImages),
    ofdOpenFile.FileNames);
```

As I mentioned earlier, the `ThreadPool` is a class you often use with asynchronous programming. It allows you to spin up a thread, execute a task, and return the thread to the system with little effort on your part. Unlike spinning up your own threads, the `ThreadPool` automatically increases and decreases the number of threads in an application depending on needs and available resources. Threads in the pool come into being, are used, and are reused using patterns that are efficient for the operating system. For us as application developers, the good part is that all of this behavior is free. The `WaitCallback` accepted by the worker item takes a method that has a signature of `void (System.Object)`, and the `AddImages` implementation of this is shown in Listing 3.15.

Listing 3.15: The `AddImages` method

```
void AddImages(object data)
{
    // Cast the data to a list of filenames.
    string[] filenames = (string[])data;

    // Create a serializer.
    DataContractSerializer itemSerializer =
        new DataContractSerializer(typeof(ImageItem));
    foreach (string filename in filenames)
    {
        // If the string doesn't map to an existing filename,
        // go the next string in the list.
        if (!File.Exists(filename))
        {
            continue;
        }

        // Create the URI for the service via the helper
        Uri callUri = Utility.CreateUri(ServiceType.Image, "AddImage/");


        WebRequest request = Utility.CreateRequest(callUri);
        request.ContentType = "application/xml";
        request.Method = "POST";
```

continues

Listing 3.15: Continued

```
            Stream requestStream = request.GetRequestStream();
            ImageItem imageItem = new ImageItem()
            {
                PublicImage = false,
                ImageBytes = File.ReadAllBytes(filename),
                UserName = LoggedInUserId
            };
            itemSerializer.WriteObject(requestStream, imageItem);
            requestStream.Close();
            IAsyncResult result =
                request.BeginGetResponse(ReadResponse, new object[]
                   {request, imageItem});
        }
        this.Invoke((MethodInvoker)delegate
        {
            // Enable the control.
            this.Enabled = true;
        });
    }
```

`AddImages` works in concert with the application's `ReadResponse` method to retrieve the new image identifier from the response stream, to set the identifier to an accessible location, and then to signal `MessagingEvent`, a `ManualResetEvent` object, which allows the blocked `AddImages` to continue, as shown in Listing 3.16.

Listing 3.16: Photo Client's `ReadResponse` method

```
    void ReadResponse(IAsyncResult result)
    {
        DataContractSerializer guidSerializer = new
            DataContractSerializer(typeof(Guid));
        object[] values = (object[])result.AsyncState;
        WebRequest request = (WebRequest)values[0];
        ImageItem imageItem = (ImageItem)values[1];

        try
        {
            WebResponse response = request.EndGetResponse(result);
            Guid lastImageId = (Guid)guidSerializer.ReadObject(
                response.GetResponseStream());
            response.Close();
            // Set the ID
            if (lastImageId != Guid.Empty)
            {
                imageItem.ImageId = lastImageId;
```

```
                ImageDocument.Add(imageItem);
                this.Invoke((MethodInvoker)delegate
                {
                    lstImages.Items.Add(new ListViewItem(imageItem.Caption,
                        AddImageToList(imageItem.Image)) {
                          Tag = imageItem });
                });
            }
        }
        catch (WebException we)
        {
            Debug.WriteLine(we.Message);
        }
    }
```

The only HTTP method we haven't looked at yet is `DELETE`. Like the other HTTP methods Photo Client deals with, it is a matter of setting the `WebRequest.Method` property to `DELETE` and then sending a request to the right resource. When the Delete button is clicked (refer to Figure 3.2), this line sets things into motion:

```
ThreadPool.QueueUserWorkItem(new WaitCallback(DeleteImage),
    lstImages.SelectedItems[i].Tag);
```

The `ThreadPool` then calls `DeleteImage`, which asynchronously executes the `WebRequest`. Since this is a `DELETE`, we aren't too worried about the response. We just need to make sure that `EndGetResponse` is called so that no operating system resources are left hanging and so that the `WebResponse` is closed, as shown in Listing 3.17.

Listing 3.17: Photo Client's `DeleteImage` and `DeleteResponse` methods

```
void DeleteImage(object data)
{
    ImageItem imageItem = (ImageItem)data;
    if (data == null)
    {
        return;
    }

    // Set the URI
    Uri callUri = Utility.CreateUri(ServiceType.Image,
        string.Format("Image/{0}", imageItem.ImageId));

    // Execute the request.
```

continues

Listing 3.17: Continued

```
        WebRequest request = Utility.CreateRequest(callUri);
        request.ContentType = "application/xml";
        request.Method = "DELETE";
        request.BeginGetResponse(DeleteResponse, request);
    }

    void DeleteResponse(IAsyncResult result)
    {
        try
        {
            WebRequest request = (WebRequest)result.AsyncState;
            // nothing to read, so just complete the request.
            request.EndGetResponse(result).Close();
        }
        catch (WebException)
        {
            Invoke((MethodInvoker)delegate
            {
                MessageBox.Show(this,
                 "Failed to delete an image. Try logging back in again.",
                 "Delete Failed", MessageBoxButtons.OK,
                 MessageBoxIcon.Error);
            });
        }
    }
```

As you read through the sample code, you'll probably notice that the code always calls `WebResponse.Close()`. You must do this so that connections to the server can be returned to the server's connection pool. This allows the server to serve more clients and is simply good programming practice. Failure to do so can generate sluggish performance for your client application and others that consume the RESTful service.

Working with Windows Presentation Foundation

Thankfully, the RESTful service communication tools you use with WinForms and WPF are identical. The only difference when creating WPF client applications is that the data binding mechanisms change and the mechanisms to apply updates to the user interface using the single UI thread change. If you are a WPF developer, read the WinForms section to understand the RESTful communications tools introduced there if you're not already familiar with them.

As mentioned, the only real difference in these two sample applications is the mechanism by which you update the user interface when a RESTful response comes back. As with WinForms, you can execute the update tasks on a background thread using an asynchronous request or by making the request on a background thread via `QueueUserWorkItem`. When the response is returned, you still need to be sure to process updates on the main user interface thread. WPF user interface objects have a property named `Dispatcher` of type `System.Windows.Threading.Dispatcher`. Through the `Dispatcher` property, we can make sure that our code executes on the main UI thread just as we used `Invoke` with the WinForms application logic.

For much of the application code in the WPF version of the Photo Sharing application, we use a simple delegate of the form:

```
delegate void PlainMethod();
```

This delegate is used to cast the anonymous delegates used by our code to a known type when we need to update the user interface in some way. After the list of images is known, for example, the code needs to retrieve the images and place them into the list for viewing. Each image might take a bit of time to load. Because of this, `ShowImages` is executed using a background thread obtained from the thread pool, which is shown in Listing 3.18.

Listing 3.18: Photo Client's `ShowImages` method (WPF version)

```
private void ShowImages()
{
    foreach (ImageItem imageItem in ImageDocument)
    {
        // This is executing on a background thread, so we block
        // and wait for each image to load.
        BitmapImage image = new BitmapImage(new
           Uri(imageItem.ImageUrl));

        // Add the image to the screen.
        this.Dispatcher.Invoke((PlainMethod)delegate
        {
            StackPanel sp = new StackPanel();
            sp.Children.Add(new Image() { Source = image, Height=100,
               Width=100 });
            sp.Children.Add(new Label() {
```

continues

Listing 3.18: Continued

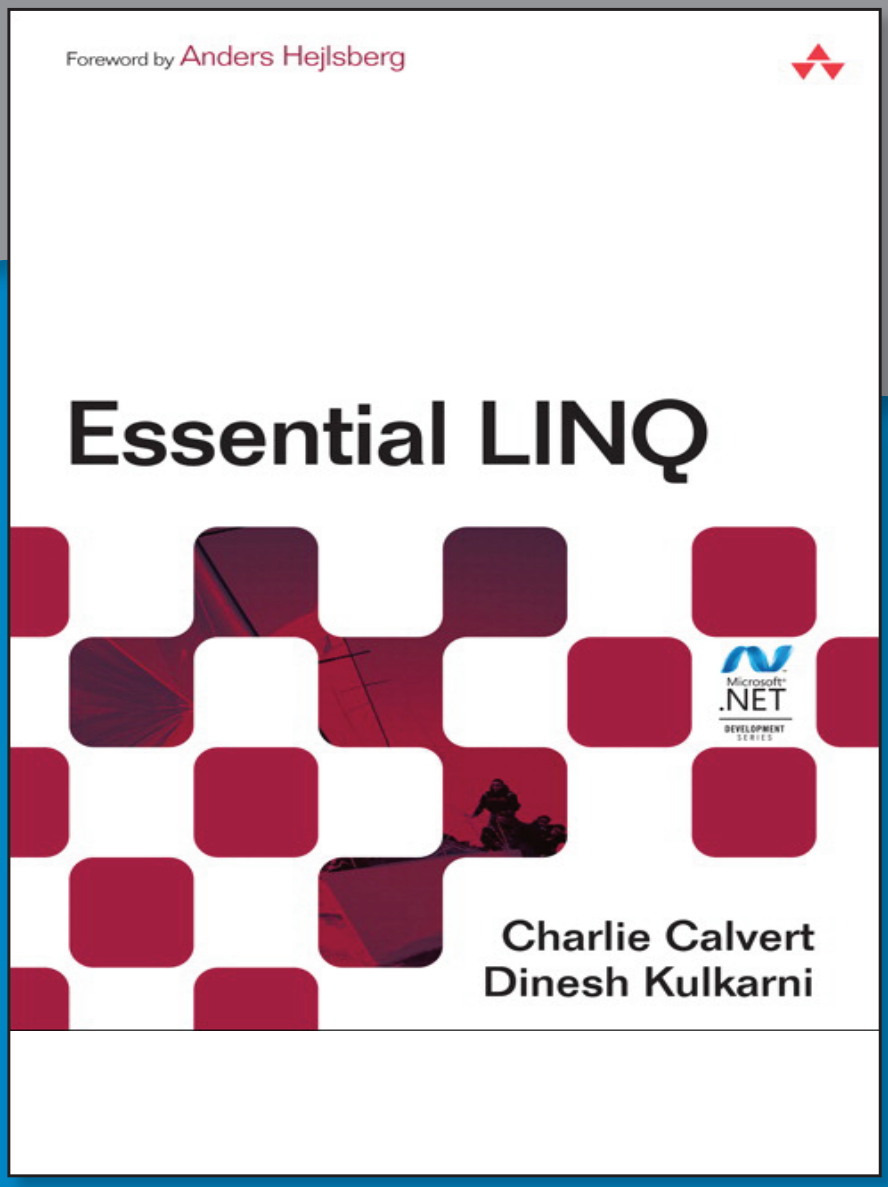
```
                Content = imageItem.Caption });
            lstImages.Items.Add(new ListViewItem() {
                Content = sp, Tag = imageItem });
        });
    }
}
```

Although we bind to data using normal WPF mechanisms for most of this example application, the images themselves are bound a bit differently, as you see in Listing 3.18. Otherwise, the application logic is nearly identical to that in the WinForms version, including the mechanisms for service authentication and invocation.

Where Are We?

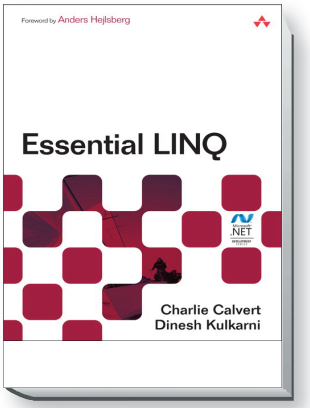
In this chapter we looked at techniques to read and write RESTful service data. RESTful Web services frequently use a neutral data format like XML or JSON, but our application code uses classes and objects to represent the data. Client applications need to know how to translate between those representations. You also need to be able to send and receive representational data between your application and the RESTful service. Fortunately, `System.Net` includes plenty of classes and makes this easy. For the most part, you will use `WebRequest/WebResponse`, or their HTTP-based derivatives, to send and receive information from RESTful services. Because you'll want to keep your application responsive to user inputs when making long-running network requests, you should become familiar with the various mechanisms to call remote methods asynchronously (in the background). When service invocations return information to be presented to the user, be sure to update the user interface via the application's UI thread.

This chapter focused on desktop applications, although Windows services and console applications would use the same techniques to invoke RESTful services as well. But how are RESTful services consumed when the client application is Web-based? In the next chapter we'll look at both browser-based and Silverlight clients and see how this is accomplished.



Charlie Calvert
Dinesh Kulkarni

BUY ME



Essential LINQ

LINQ is one of Microsoft's most exciting, powerful new development technologies. Essential LINQ is the first LINQ book written by leading members of Microsoft's LINQ and C# teams. Writing for architects, developers, and development managers, these Microsoft insiders share their intimate understanding of LINQ, revealing new patterns and best practices for getting the most out of it.

Calvert and Kulkarni begin by clearly explaining how LINQ resolves the long-time "impedance mismatch" between object-oriented code and relational databases. Next, they show how LINQ integrates querying into C# as a "first-class citizen." Using realistic code examples, they show how LINQ provides a strongly typed, IntelliSense-aware technology for working with data from any source, including SQL databases, XML files, and generic data structures.

Calvert and Kulkarni carefully explain LINQ's transformative, composable, and declarative capabilities. By fully illuminating these three concepts, the authors allow developers to discover LINQ's full power. In addition to covering core concepts and hands-on LINQ development in C# with LINQ to Objects, LINQ to XML, LINQ to SQL, and LINQ to Entities, they also present advanced topics and new LINQ implementations developed by the LINQ community. This book

- Explains the entire lifecycle of a LINQ project: design, development, debugging, and much more
- Teaches LINQ from both a practical and theoretical perspective
- Leverages C# language features that simplify LINQ development
- Offers developers powerful LINQ query expressions to perform virtually any data-related task
- Teaches how to query SQL databases for objects and how to modify those objects
- Demonstrates effective use stored procedures and database functions with LINQ
- Shows how to add business logic that reflects the specific requirements of your organization
- Teaches developers to create, query, and transform XML data with LINQ
- Shows how to transform object, relational, and XML data between each other
- Offers best patterns and practices for writing robust, easy-to-maintain LINQ code

AVAILABLE
- BOOK: 9780321564160
- SAFARI ONLINE
- EBOOK: 0321564189
- KINDLE: 0321604229

About the Authors

Charlie Calvert, Community Program Manager for the Microsoft C# team, currently focuses his technical energies on LINQ. He has periodically worked with LINQ Chief Architect Anders Hejlsberg both during the development of Delphi and during the development of LINQ. Calvert's ten technical books have sold more than 100,000 copies. They include Delphi 4 Unleashed, C++Builder 3 Unleashed, Delphi 2 Unleashed, Teach Yourself Windows 95 Programming in 21 Days, and Teach Yourself Windows Programming.

Dinesh Kulkarni is a Senior Program Manager on Microsoft's .NET Framework team. He was the Program Manager in charge of LINQ to SQL. He was deeply involved in LINQ's planning and implementation from the incubation stage and was lead author for MSDN's authoritative LINQ to SQL paper. Before joining Microsoft, he worked in diverse technical roles ranging from architecting and implementing front-end CASE tools for IBM to designing databases and middleware for a Wall Street hedge fund.

Addison Wesley

informit.com/aw

3
The Essence of LINQ

Now that you've seen several practical examples of LINQ's syntax, it is time to view the technology from a more theoretical perspective. This chapter covers the seven foundations on which an understanding of LINQ can be built. LINQ is

- Integrated
- Unitive
- Extensible
- Declarative
- Hierarchical
- Composable
- Transformative

These ideas may sound esoteric at first, but I believe you will find them quite easy to understand. LINQ has a fundamental simplicity and elegance. In this chapter and the next, we explore LINQ's architecture, giving you a chance to understand how it was built and why it was built that way. This chapter explains goals that LINQ aims to achieve. The next chapter explains each of the pieces of the LINQ architecture and shows how they come together to achieve those goals.

Integrated

LINQ stands for Language Integrated Query. One of the central, and most important, features of LINQ is its integration of a flexible query syntax into the C# language.

Developers have many tools that have been crafted to neatly solve difficult tasks. Yet there are still dark corners in the development landscape. Querying data is one area in which developers frequently encounter problems with no clear resolution. LINQ aims to remove that uncertainty and to show a clearly defined path that is well-lit and easy to follow.

In Visual Studio 2005, attempts to query data in a SQL database from a C# program revealed an impedance mismatch between code and data. SQL is native to neither .NET nor C#. As a result, SQL code embedded in a C# program is neither type-checked nor IntelliSense-aware. From the perspective of a C# developer, SQL is shrouded in darkness.

Here is an example of one of several different techniques developers used in the past when querying data:

```
SqlConnection sqlConnection = new SqlConnection(connectString);
sqlConnection.Open();
System.Data.SqlClient.SqlCommand sqlCommand = new SqlCommand();
sqlCommand.Connection = sqlConnection;
sqlCommand.CommandText = "Select * from Customer";
return sqlCommand.ExecuteReader(CommandBehavior.CloseConnection)
```

Of these six lines of code, only the last two directly define a query. The rest of the lines involve setup code that allows developers to connect and call objects in the database. The query string shown in the next-to-last line is neither type-checked nor IntelliSense-aware.

After these six lines of code execute, the developers may have more work to do, because the data returned from the query is not readily addressable by an object-oriented programmer. You might have to write more lines of code to access this data, or convert it into a format that is easier to use.

The LINQ version of this same query is shorter, easier to read, color-coded, fully type-checked, and IntelliSense-aware. The result set is cleanly converted into a well-defined object-oriented format:

```
Northwind db = new Northwind(@"C:\Data\Northwnd.mdf");

var query = from c in db.Customers
            select c;
```

By fully integrating the syntax for querying data into .NET languages such as C# and VB, LINQ resolves a problem that has long plagued the development world. Queries become first-class citizens of our primary languages; they are both type-checked and supported by the powerful IntelliSense technology provided inside the Visual Studio IDE. LINQ brings the experience of writing queries into the well-lit world of the 21st century.

A few benefits accrue automatically as a result of integrating querying into the C# language:

- The syntax highlighting and IntelliSense support allow you to get more work done in less time. The Visual Studio editor automatically shows you the tables in your database, the correctly spelled names and types of your fields, and the operators you can use when querying data. This helps you save time and avoid careless mistakes.
- LINQ code is shorter and cleaner than traditional techniques for querying data and, therefore, is much easier to maintain.
- LINQ allows you to fully harness the power of your C# debugger while writing and maintaining queries. You can step through your queries and related code in your LINQ projects.

If language integration were the only feature that LINQ offered, that alone would have been a significant accomplishment. But we are only one-seventh of the way through our description of the foundations of LINQ. Many of the best and most important features are still to be covered.

Unitive

Before LINQ, developers who queried data frequently needed to master multiple technologies. They needed to learn the following:

- SQL to query a database
- XPath, Dom, XSLT, or XQuery to query and transform XML data

- Web services to access some forms of remote data
- Looping and branching to query the collections in their own programs

These diverse APIs and technologies forced developers to frantically juggle their tight schedules while struggling to run similar queries against dissimilar data sources. Projects often encountered unexpected delays simply because it was easier to talk about querying XML, SQL, and other data than it was to actually implement the queries against these diverse data sources. If you have to juggle too many technologies, eventually something important will break.

LINQ simplifies these tasks by providing a single, unified method for querying diverse types of data. Developers don't have to master a new technology simply because they want to query a new data source. They can call on their knowledge of querying local collections when they query relational data, and vice versa.

This point was illustrated in the preceding chapter, where you saw three very similar queries that drew data from three different data sources: objects, an SQL database, and XML:

```
var query = from c in GetCustomers()
            where c.City == "Mexico D.F."
            select new { City = c.City, ContactName = c.ContactName };

var query = from c in db.Customers
            where c.City == "Mexico D.F."
            select new { City = c.City, ContactName = c.ContactName };

var query = from x in customers.Descendants("Customer")
            where x.Attribute("City").Value == "Mexico D.F."
            select x;
```

As you can see, the syntax for each of these queries is not identical, but it is very similar. This illustrates one of LINQ's core strengths: a single, unitive syntax can be used to query diverse types of data. It is not that you never have to scale a learning curve when approaching a new data source, but only that the principles, overall syntax, and theory are the same even if some of the details differ.

You enjoy two primary benefits because LINQ is unitive:

- The similar syntax used in all LINQ queries helps you quickly get up to speed when querying new data sources.
- Your code is easier to maintain, because you are using the same syntax regardless of the type of data you query.

Although it arises naturally from this discussion, it is worth noting that SQL and other query languages do not have this capability to access multiple data sources with a single syntax. Those who advocate using SQL or the DOM instead of LINQ often forget that their decision forces their team to invest additional time in learning these diverse technologies.

Extensible Provider Model

In this text I have tended to define LINQ as a tool for querying SQL, XML, and the collections in a program. Strictly speaking, this is not an accurate description of LINQ. Although such a view is useful when you first encounter LINQ, it needs to be abandoned if you want to gain deeper insight. LINQ is not designed to query any particular data source; rather, it is a technology for defining *providers* that can be used to access any arbitrary data source. LINQ happens to ship with providers for querying SQL, XML, and objects, but this was simply a practical decision, not a preordained necessity.

LINQ provides developers with a syntax for querying data. This syntax is enabled by a series of C# 3.0 and C# 2.0 features. These include lambdas, iterator blocks, expression trees, anonymous types, type inference, query expressions, and extension methods. All of these features are covered in this book. For now you need only understand that they make LINQ possible.

When Visual Studio 2008 shipped, Microsoft employees frequently showed the image shown in Figure 3.1. Although people tend to think of LINQ as a means of enabling access to these data sources, this diagram actually depicts nothing more than the set of LINQ providers that were implemented by Microsoft at the time Visual Studio shipped. Granted, the team carefully planned which providers they wanted to ship, but their decisions were based on strategic, rather than technical, criteria.

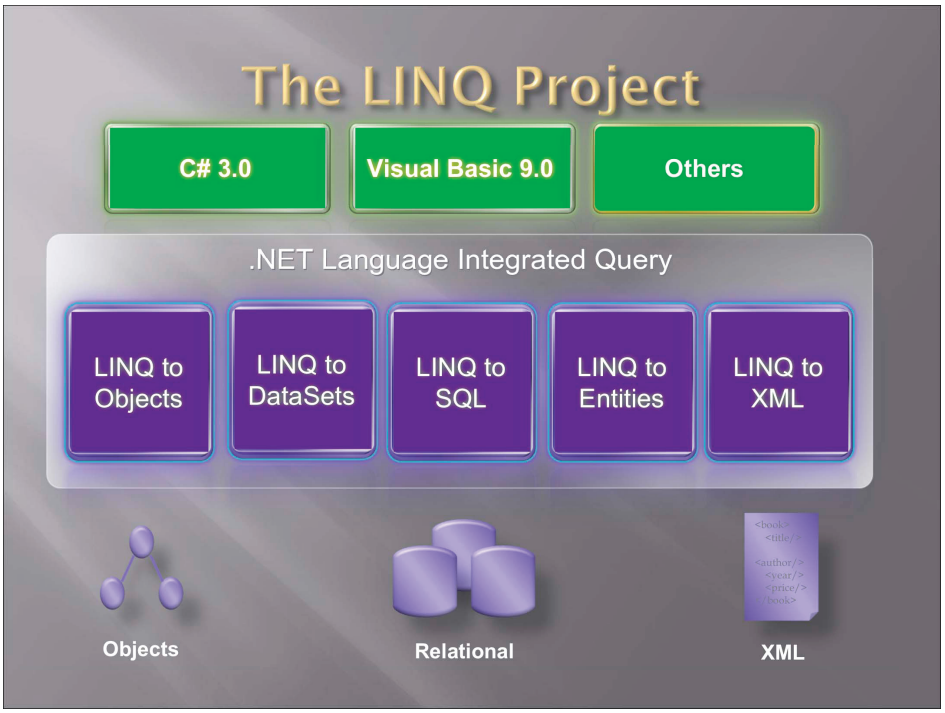


Figure 3.1 VB and C# ship with LINQ providers for databases, XML, and data structures found in a typical program.

Using the LINQ provider model, developers can extend LINQ to query other data sources besides those shown in Figure 3.1. The following are a few of the data sources currently enabled by third-party LINQ providers:

LINQ Extender
LINQ over C# project
LINQ to Active Directory
LINQ to Amazon
LINQ to Bindable Sources
LINQ to CRM
LINQ to Excel
LINQ to Expressions
LINQ to Flickr
LINQ to Geo
LINQ to Google
LINQ to Indexes
LINQ to `IQueryable`
LINQ to JavaScript
LINQ to JSON
LINQ to LDAP
LINQ to LLBLGen Pro
LINQ to Lucene
LINQ to Metaweb
LINQ to MySQL

LINQ to NCover	LINQ to Sharepoint
LINQ to NHibernate	LINQ to SimpleDB
LINQ to Opf3	LINQ to Streams
LINQ to Parallel (PLINQ)	LINQ to WebQueries
LINQ to RDF Files	LINQ to WMI

These projects are of varying quality. Some, such as the LINQ Extender and LINQ to `IQueryable`, are merely tools for helping developers create providers. Nevertheless, you can see that an active community is interested in creating LINQ providers, and this community is producing some interesting products. By the time you read this, I'm sure the list of providers will be longer. See Appendix A for information on how to get updated information on existing providers.

One easily available provider called LinqToTerraServer can be found among the downloadable samples that ship with Visual Studio 2008. You can download the VS samples from the release tab found at http://code.msdn.microsoft.com/csharpsamples.

After unzipping the download, if you look in the ...\LinqSamples\WebServiceLinqProvider directory, you will find a sample called LinqToTerraServer. The TerraServer web site, http://terraserver-usa.com, is a vast repository of pictures and information about geographic information. The LinqToTerraServer example shows you how to create a LINQ provider that queries the web services provided on the TerraServer site. For example, the following query returns all U.S. cities and towns named Portland:

```
var query1 = from place in terraPlaces
             where place.Name == "Portland"
             select new { place.Name, place.State };
```

This query returns a number of locations, but here are a few of the more prominent:

```
{ Name = Portland, State = Indiana }
{ Name = Portland, State = Maine }
{ Name = Portland, State = Michigan }
{ Name = Portland, State = Oregon }
{ Name = Portland, State = Texas }
```

```
{ Name = Portland, State = Alabama }
{ Name = Portland, State = Arkansas }
{ Name = Portland, State = Colorado }
```

In Chapter 17, "LINQ Everywhere," you will see examples of several other providers, including LINQ to Flickr and LINQ to SharePoint. It is not easy to create a provider.. After the code is written, however, it is easy to use the provider. In fact, you should already have enough familiarity with LINQ to see that it would be easy to modify the preceding query to suit your own purposes.

The LINQ provider model has hidden benefits that might not be evident at first glance:

- It is relatively open to examination and modification. As you read the next few chapters, you will find that most of the LINQ query pipeline is accessible to developers.
- It allows developers to be intelligent about how queries execute. You can get a surprising degree of control over the execution of a query. If you care about optimizing a query, in many cases you can optimize it, because you can see how it works.
- You can create a provider to publicize a data source that you have created. For instance, if you have a web service that you want C# developers to access, you can create a provider to give them a simple, extensible way to access your data.

I will return to the subject of LINQ providers later in the book. In this chapter, my goal is simply to make it clear that LINQ is extensible, and that its provider model is the basis on which each LINQ query model is built.

Query Operators

You don't always need to use a LINQ provider to run queries against what might—at least at first—appear to be nontraditional data sources. By using the LINQ to Objects provider, and a set of built-in LINQ operators, you can run queries against a data source that does not look at all like XML or SQL data. For instance, LINQ to Objects gives you access to the reflection model that is built into C#.

The following query retrieves all the methods of the `string` class that are static:

```
var query = from m in typeof(string).GetMethods()
            where m.IsStatic == true
            select m;
```

The following are a few of the many results that this query returns:

```
System.String Join(System.String, System.String[])
System.String Join(System.String, System.String[], Int32, Int32)
Boolean Equals(System.String, System.String)
Boolean Equals(System.String, System.String, System.StringComparison)
Boolean op_Equality(System.String, System.String)
Boolean op_Inequality(System.String, System.String)
Boolean IsNullOrEmpty(System.String)
Int32 Compare(System.String, System.String)
Int32 Compare(System.String, System.String, Boolean)
Int32 Compare(System.String, System.String, System.StringComparison)
```

Using the power of LINQ, it is easy to drill into these methods to find out more about them. In particular, LINQ uses the extension methods mentioned in the preceding section to define a set of methods that can perform specific query operations such as ordering and grouping data. For instance, the following query retrieves the methods of the `string` class that are static, finds out how many overloads each method has, and then orders them first by the number of overloads and then alphabetically:

```
var query = from m in typeof(string).GetMethods()
            where m.IsStatic == true
            orderby m.Name
            group m by m.Name into g
            orderby g.Count()
            select new { Name = g.Key, Overloads = g.Count() };

foreach (var item in query)
{
    Console.WriteLine(item);
}
```

The results of this query look like this:

```
{ Overloads = 1, Name = Copy }
{ Overloads = 1, Name = Intern }
{ Overloads = 1, Name = IsInterned }
{ Overloads = 1, Name = IsNullOrEmpty }
```

```
{ Overloads = 1, Name = op_Equality }
{ Overloads = 1, Name = op_Inequality }
{ Overloads = 2, Name = CompareOrdinal }
{ Overloads = 2, Name = Equals }
{ Overloads = 2, Name = Join }
{ Overloads = 5, Name = Format }
{ Overloads = 9, Name = Concat }
{ Overloads = 10, Name = Compare }
```

This makes it obvious that `Format`, `Compare`, and `Concat` are the most frequently overloaded methods of the `string` class, and it presents all the methods with the same number of overloads in alphabetical order.

You can run this code in your own copy of Visual Studio because the LINQ to Objects provider ships with C# 3.0. Other third-party extensions to LINQ, such as LINQ to Amazon, are not included with Visual Studio. If you want to run a sample based on LINQ to Amazon or some other provider that does not ship with Visual Studio, you must download and install the provider before you can use it.

Declarative: Not How, But What

LINQ is declarative, not imperative. It allows developers to simply state what they want to do without worrying about how it is done.

Imperative programming requires developers to define step by step how code should be executed. To give directions in an imperative fashion, you say, "Go to 1st Street, turn left onto Main, drive two blocks, turn right onto Maple, and stop at the third house on the left." The declarative version might sound something like this: "Drive to Sue's house." One says *how* to do something; the other says *what* needs to be done.

The declarative style has two advantages over the imperative style:

- It does not force the traveler to memorize a long set of instructions.
- It allows the traveler to optimize the route when possible.

It should be obvious that there is little opportunity to optimize the first set of instructions for getting to Sue's house: You simply have to follow them by rote. The second set, however, allows the traveler to use his or her knowledge of the neighborhood to find a shortcut. For instance, a bike

might be the best way to travel at rush hour, whereas a car might be best at night. On occasion, going on foot and cutting through the local park might be the best solution.

Here is another example of the difference between declarative and imperative code:

```
// imperative style
List<int> imperativeList = new List<int>();
imperativeList.Add(1);
imperativeList.Add(2);
imperativeList.Add(3);

// declarative style
List<int> declarativeList = new List<int> { 1, 2, 3 };
```

The first example details exactly how to add items to a list. The second example states what you want to do and allows the compiler to figure out the best way to do it. As you will learn in the next chapter, both styles are valid C# 3.0 syntax. The declarative form of this code, however, is shorter, easier to understand, easier to maintain, and, at least in theory, leaves the compiler free to optimize how a task is performed.

These two styles differ in both the amount of detail they require a developer to master and the amount of freedom that each affords the compiler. Detailed instructions not only place a burden on the developer, but also restrict the compiler's capability to optimize code.

Let's consider another example of the imperative style of programming. As developers, we frequently end up in a situation where we are dealing with a list of lists:

```
List<int> list01 = new List<int> { 1, 2, 3 };
List<int> list02 = new List<int> { 4, 5, 6 };
List<int> list03 = new List<int> { 7, 8, 9 };

List<List<int>> lists = new List<List<int>> { list01, list02, list03 };
```

Here is imperative code for accessing the members of this list:

```
List<int> newList = new List<int>();

foreach (var item in lists)
{
    foreach (var number in item)
```

```
    {
        newList.Add(number);
    }
}
```

This code produces a single list containing all the data from the three nested lists:

```
1
2
3
4
5
6
7
8
9
```

Notice that we have to write nested `for` loops to allow access to our data. In a simple case like this, nested loops are not terribly complicated to use, but they can become very cumbersome in more complex problem domains.

Contrast this code with the declarative style used in a LINQ program:

```
var newList = from list in lists
              from num in list
              select num;
```

You can access the results of these two "query techniques" in the same way:

```
foreach (var item in newList)
{
    Console.WriteLine(item);
}
```

This code writes the results of either query, producing identical results, regardless of whether you used the imperative or declarative technique to query the data:

```
1
2
3
4
5
6
7
8
9
```

The difference here is not in the query's results, or in how we access the results, but in how we compose our query against our nested list. The imperative style can sometimes be verbose and hard to read. The declarative code is usually short and easy to read and scales more easily to complex cases. For instance, you can add an `orderby` clause to reverse the order of the integers in your result set:

```
var query = from list in lists
            from num in list
            orderby num descending
            select num;
```

You probably know how to achieve the same results using the imperative style. But it was knowledge that you had to struggle to learn, and it is knowledge that applies only to working with sequences of numbers stored in a `List<T>`. The LINQ code for reordering results, however, is easy to understand. It can be used to reorder not only nested collections, but also SQL data, XML data, or the many other data sources we query using LINQ.

To get the even numbers from our nested lists, we need only do this:

```
var query = from list in lists
            from num in list
            where num % 2 == 0
            orderby num descending
            select num;
```

Contrast this code with the imperative equivalent:

```
List<int> newList = new List<int>();

foreach (var item in lists)
{
    foreach (var number in item)
    {
        if (number % 2 == 0)
        {
            newList.Add(number);
        }
    }
}

newList.Reverse();
```

This imperative style of programming now has an `if` block nested inside the nested `foreach` loops. This is not only verbose and applicable to only a specific type of data, it also can be like a straight jacket for both the compiler and the developer. Commands must be issued and followed in a rote fashion, leaving little room for optimizations.

The equivalent LINQ query expression does not describe in a step-by-step fashion how to query our list of lists. It simply lets the developer state what he wants to do and lets the compiler determine the best path to the destination.

After nearly 50 years of steady development, the possibilities inherent in imperative programming have been extensively explored. Innovations in the field are now rare. Declarative programming, on the other hand, offers opportunities for growth. Although it is not a new field of study, it is still rich in possibilities.

Use the Right Tool for the Job

In extolling the virtues of LINQ's declarative syntax, I should be careful not to overstate my case. For instance, the LINQ operator called `ToList` is provided to allow developers to easily translate the sequence of results returned by a LINQ query into a traditional `List<T>`. This functionality is useful because some operations, such as randomly accessing items in a list (`myList[2]`), are more easily performed using the imperative syntax. One of the great virtues of C# 3.0 is that it allows you to easily move between imperative and declarative syntax, allowing you to choose the best tool for the job. My job right now is to help you understand the value of LINQ and the declarative style of programming. LINQ is indeed a very powerful and useful tool, but it is not the solution to all your problems.

Because LINQ is a new technology from Microsoft, you might find it a bit jarring to see me write that declarative programming is not new. In fact, declarative code has been with us nearly as long as imperative code. Some older languages such as LISP (which was first specified in 1958) make heavy use of the declarative style of programming. Haskel and F# are examples of

other languages that use it extensively. One reason LINQ and SQL look so much alike is that they are both forms of declarative programming.

The point of LINQ is not that it will replace SQL, but that it will bring the benefits of SQL to C# developers. LINQ is a technology for enabling a SQL-like declarative programming style inside a native C# program. It brings you the benefits of SQL but adds declarative syntax, as well as syntax highlighting, IntelliSense support, type checking, debugging support, the ability to query multiple data sources with the same syntax, and much more.

Hierarchical

Complex relationships can be expressed in a relational database, but the results of a SQL query can take only one shape: a rectangular grid. LINQ has no such restrictions. Built into its very foundation is the idea that data is hierarchical (see Figure 3.2). If you want to, you can write LINQ queries that return flat, SQL-like datasets, but this is an option, not a necessity.

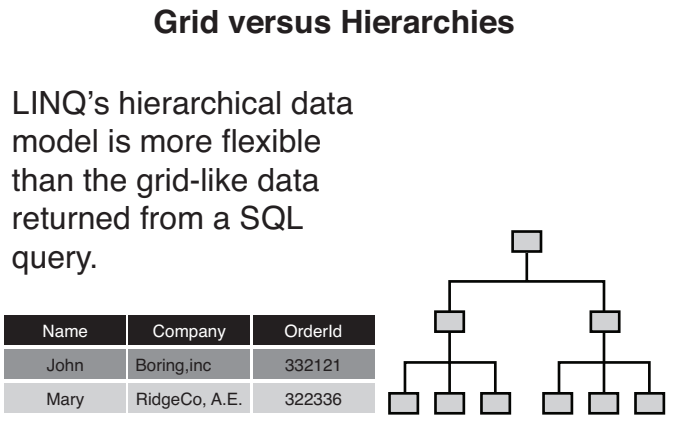


Figure 3.2 Both object-oriented languages and the developers who use them have a natural tendency to think in terms of hierarchies. SQL data is arranged in a simple grid.

Consider a simple relational database that has tables called Customers, Orders, and OrderDetails. It is possible to capture the relationship between these tables in a SQL database, but you cannot directly depict the relationship

in the results of a single query. Instead, you are forced to show the result as a join that binds the tables into a single array of columns and rows.

LINQ, on the other hand, can return a set of `Customer` objects, each of which owns a set of 0-to-*n* `Orders`. Each `Order` can be associated with a set of `OrderDetails`. This is a classic hierarchical relationship that can be perfectly expressed with a set of objects:

Customer
    Orders
        OrderDetails

Consider the following simple hierarchical query that captures the relationship between two objects:

```
var query = from c in db.Customers
            select new { City = c.City,
                         orders = from o in c.Orders
                         select new { o.OrderID }
                       };
```

This query asks for the city in which a customer lives and a list of the orders the person has made. Rather than returning a rectangular dataset as a SQL query would, this query returns hierarchical data that lists the city associated with each customer and the ID associated with each order:

```
City=Helsinki   orders=...
  orders: OrderID=10615
  orders: OrderID=10673
  orders: OrderID=10695
  orders: OrderID=10873
  orders: OrderID=10879
  orders: OrderID=10910
  orders: OrderID=11005
City=Warszawa   orders=...
  orders: OrderID=10374
  orders: OrderID=10611
  orders: OrderID=10792
  orders: OrderID=10870
  orders: OrderID=10906
  orders: OrderID=10998
```

This result set is multidimensional, nesting one set of columns and rows inside another set of columns and rows.

Look again at the query, and notice how we gain access to the Orders table:

```
orders = from o in c.Orders
```

The identifier `c` is an instance of a `Customer` object. As you will learn later in the book, LINQ to SQL has tools for automatically generating `Customer` objects given the presence of the Customer table in the database. Here you can see that the `Customer` object is not flat; instead, it contains a set of nested `Order` objects.

Listing 3.1 shows a simplified version of the `Customer` object that is automatically generated by the LINQ to SQL designer. Notice how LINQ to SQL wraps the fields of the Customer table. Later in this book, you will learn how to automatically generate `Customer` objects that wrap the fields of a Customer table.

Listing 3.1 A Simplified Version of the `Customer` Object That the LINQ to SQL Designer Generates Automatically

```
public partial class Customer
{
     ... // Code omitted here
     private string _CustomerID;
     private string _CompanyName;
     private string _ContactName;
     private string _ContactTitle;
     private string _Address;
     private string _City;
     private string _Region;
     private string _PostalCode;
     private string _Country;
     private string _Phone;
     private string _Fax;
     private EntitySet<Order> _Orders;
     ... // Code omitted here
}
```

The first 11 private fields of the `Customer` object simply reference the fields of the Customer table in the database. Taken together, they provide a location to store the data from a single row of the Customer table. Notice, however, the last item, which is a collection of `Order` objects. Because it is

bound to the Orders table in a one-to-many relationship, each customer has from 0-to-*n* orders associated with it, and LINQ to SQL stores those orders in this field. This automatically gives you a hierarchical view of your data.

The same thing is true of the Order table, only it shows not a one-to-many relationship with the Customer table, but a one-to-one relationship:

```
public partial class Order
{
     ... // Code omitted here
     private int _OrderID;
     private string _CustomerID;
     private System.Nullable<int> _EmployeeID;
     private System.Nullable<System.DateTime> _OrderDate;
     private System.Nullable<System.DateTime> _RequiredDate;
     private System.Nullable<System.DateTime> _ShippedDate;
     private System.Nullable<int> _ShipVia;
     private System.Nullable<decimal> _Freight;
     private string _ShipName;
     private string _ShipAddress;
     private string _ShipCity;
     private string _ShipRegion;
     private string _ShipPostalCode;
     private string _ShipCountry;
     private EntityRef<Customer> _Customer;
     ... // Code omitted here
}
```

Again we see all the fields of the Orders table, their types, and whether they can be set to Null. The difference here is that the last field points back to the Customer table not with an `EntitySet<T>`, but an `EntityRef<T>`. This is not the proper place to delve into the `EntitySet` and `EntityRef` classes. However, it should be obvious to you that an `EntitySet` refers to a set of objects, and an `EntityRef` references a single object. Thus, an `EntitySet` captures a one-to-many relationship, and an `EntityRef` captures a one-to-one relationship.

The point to take away from this discussion is that LINQ to SQL captures not a flat view of your data, but a hierarchical view. A `Customer` class is connected to a set of orders in a clearly defined hierarchical relationship, and each order is related to the customer who owns it. LINQ gives you a hierarchical view of your data.

In a simple case like this, such a hierarchical relationship has obvious utility, but it is possible to imagine getting along without it. More complex

queries, however, are obviously greatly simplified by this architecture. Consider the following LINQ to SQL query:

```
var query = from c in db.Customers
            where c.CompanyName == companyName
            from o in c.Orders
            from x in o.Order_Details
            where x.Product.Category.CategoryName == "Confections"
            orderby x.Product.ProductName
            group x by x.Product.ProductName into g
            orderby g.Count()
            select new { Count = g.Count(), Product = g.Key };
```

Here we use LINQ's hierarchical structure to move from the Customers table to the Orders table to the Order_Details table without breaking a sweat:

```
var query = from c in db.Customers
            from o in c.Orders
            from x in o.Order_Details
```

The next line really helps show the power of LINQ hierarchies:

```
where x.Product.Category.CategoryName == "Confections"
```

The identifier `x` represents an instance of a class containing the data from a row of the Order_Details table. Order_Details has a relationship with the Product table, which has a relationship with the Category table, which has a field called `CategoryName`. We can slice right through that complex relationship by simply writing this:

```
x.Product.Category.CategoryName
```

LINQ's hierarchical structure shines a clarifying light on the relational data in your programs. Even complex relational models become intuitive and easy to manipulate.

We can then order and group the results of our query with a few simple LINQ operators:

```
orderby x.Product.ProductName
group x by x.Product.ProductName into g
orderby g.Count()
```

Trying to write the equivalent code using a more conventional C# style of programming is an exercise that might take two or three pages of convoluted code and involve a number of nested loops and `if` statements. Even writing the same query in standard SQL would be a challenge for many developers. Here we perform the whole operation in nine easy-to-read lines of code.

In this section, I have introduced you to the power of LINQ's hierarchical style of programming without delving into the details of how such queries work. Later in this book you will learn how easy it is to compose your own hierarchical queries. For now you only need to understand two simple points:

- There is a big difference between LINQ's hierarchical structure and the flat, rectangular columns and rows returned by an SQL query.
- Many benefits arise from this more powerful structure. These include the intuitive structure of the data and the ease with which you can write queries against this model.

Composable

The last two foundations of LINQ shed light on its flexibility and power. If you understand these two features and how to use them, you will be able to tap into some very powerful technology. Of course, this chapter only introduces these features; they are discussed in more detail in the rest of the book.

LINQ queries are composable: You can combine them in multiple ways, and one query can be used as the building block for yet another query. To see how this works, let's look at a simple query:

```
var query = from customer in db.Customers
            where customer.City == "Paris"
            select customer;
```

The variable that is returned from the query is sometimes called a computation. If you write a `foreach` loop and display the address field from the customers returned by this computation, you see the following output:

```
265, boulevard Charonne
25, rue Lauriston
```

You can now write a second query against the results of this query:

```
query2 = from customer in query
         where customer.Address.StartsWith("25")
         select customer;
```

Notice that the last word in the first line of this query is the computation returned from the previous query. This second query produces the following output:

```
25, rue Lauriston
```

LINQ to Objects queries are composable because they operate on and usually return variables of type `IEnumerable<T>`. In other words, LINQ queries typically follow this pattern:

```
IEnumerable<T> query = from x in IEnumerable<T>
                       select x;
```

This is a simple mechanism to understand, but it yields powerful results. It allows you to take complex problems, break them into manageable pieces, and solve them with code that is easy to understand and easy to maintain. You will hear much more about `IEnumerable<T>` in the next chapter.

The next chapter also details a feature called deferred execution. Although it can be confusing to newcomers, one of the benefits of deferred execution is that it allows you to compose multiple queries and string them together without necessarily needing to have each query entail an expensive hit against the server. Instead, three or four queries can "execute" without ever sending a query across the wire to your database. Then, when you need to access the result from your query, a SQL statement is written that combines the results of all your queries and sends it across the wire only once. Deferred execution is a powerful feature, but you need to wait until the next chapter for a full explanation of how and why it works. The key point to grasp now is that it enables you to compose multiple queries as shown here, without having to take an expensive hit each time one "executes."

Discreet Computations and PLINQ

LINQ queries are not only composable, but also discreet. In other words, the computation returned by a query is a single self-contained expression with only a single entry point. This has important consequences for a field of study called Parallel LINQ (PLINQ). Because each computation returned by a query is discreet, it can easily be run concurrently on its own thread. PLINQ is discussed briefly in Chapter 17, "LINQ Everywhere.

Transformative

SQL is poor at transformations, so we are unaccustomed to thinking about query languages as a tool for converting data from one format to another. Instead, we usually use specialized tools such as XSLT or brute-force techniques to transform data.

LINQ, however, has transformational powers built directly into its syntax. We can compose a LINQ query against a SQL database that effortlessly performs a variety of transforms. For instance, with LINQ it is easy to transform the result of a SQL query into a hierarchical XML document. You can also easily transform one XML document into another with a different structure. SQL data is transformed into a hierarchical set of objects automatically when you use LINQ to SQL. In short, LINQ is very good at transforming data, and this adds a new dimension to our conception of what we can do with a query language.

Listing 3.2 shows code that takes the results of a query against relational data and transforms it into XML.

Listing 3.2 A Simple Query That Transforms the Results of a LINQ to SQL Query into XML

```
var query = new XElement("Orders", from c in db.Customers
              where c.City == "Paris"
              select new XElement("Order",
                 new XAttribute("Address", c.Address),
                 new XAttribute("City", c.City)));
```

Embedded in this query is a simple LINQ to SQL query that returns the `Address` and `City` fields from all the customers who live in Paris. In Listing 3.3 I've stripped away the LINQ to XML code from Listing 3.2 to show you the underlying LINQ to SQL query.

Listing 3.3 The Simple LINQ to SQL Query Found at the Heart of Listing 3.2

```
var query = from c in db.Customers
            where c.City == "Paris"
            select new { c.Address, c.City };
```

Here is the output from Listing 3.3:

```
265, boulevard Charonne
25, rue Lauriston
```

Here is the output from Listing 3.2:

```
<Orders>
  <Order Address="265, boulevard Charonne" City="Paris" />
  <Order Address="25, rue Lauriston" City="Paris" />
</Orders>
```

As you can see, the code in Listing 3.2 performs a transform on the results of the LINQ to SQL query, converting it into XML data.

Because LINQ is composable, the following query could then be used to run a second transform on this data:

```
var query1 = new XElement("Orders", new XAttribute("City", "Paris"),
    from x in query.Descendants("Order")
    where x.Attribute("City").Value == "Paris"
    select new XElement("Address", x.Attribute("Address").Value));
```

This query takes the XML results of the first query and transforms that XML into the following format:

```
<Orders City="Paris">
  <Address>265, boulevard Charonne</Address>
  <Address>25, rue Lauriston</Address>
</Orders>
```

LINQ is constantly transforming one type of data into another type. It takes relational data and transforms it into objects; it takes XML and transforms it into relational data. Because LINQ is extensible, it is at least theoretically possible to use it to tear down the walls that separate any two arbitrary data domains.

Because LINQ is both composable and transformative, you can use it in a number of unexpected ways:

- You can compose multiple queries, linking them in discrete chunks. This often allows you to write code that is easier to understand and maintain than traditional nested SQL queries.
- You can easily transform data from one data source into some other type. For instance, you can transform SQL data into XML.
- Even if you do not switch data sources, you can still transform the shape of data. For instance, you can transform one XML format into another format. If you look back at the section "Declarative: Not How, But What," you will see that we transformed data that was stored in nested lists into data that was stored in a single list. These kinds of transformations are easy with LINQ.

Summary

In this chapter you have read about the foundations of LINQ. These foundations represent the core architectural ideas on which LINQ is built. Taken together, they form the essence of LINQ. We can summarize these foundations by saying the following about LINQ:

- It is a technique for querying data that is *integrated* into .NET languages such as C# and VB. As such, it is both strongly typed and IntelliSense-aware.
- It has a single *unitive* syntax for querying multiple data sources such as relational data and XML data.
- It is *extensible*; talented developers can write providers that allow LINQ to query any arbitrary data source.
- It uses a *declarative* syntax that allows developers to tell the compiler or provider what to do, not how to do it.
- It is *hierarchical*, in that it provides a rich, object-oriented view of data.
- It is *composable*, in that the results of one query can be used by a second query, and one query can be a subclause of another query. In many cases, this can be done without forcing the execution of any one query until the developer wants that execution to take place.

- It is *transformative*, in that the results of a LINQ query against one data source can be morphed into a second format. For instance, a query against a SQL database can produce an XML file as output.

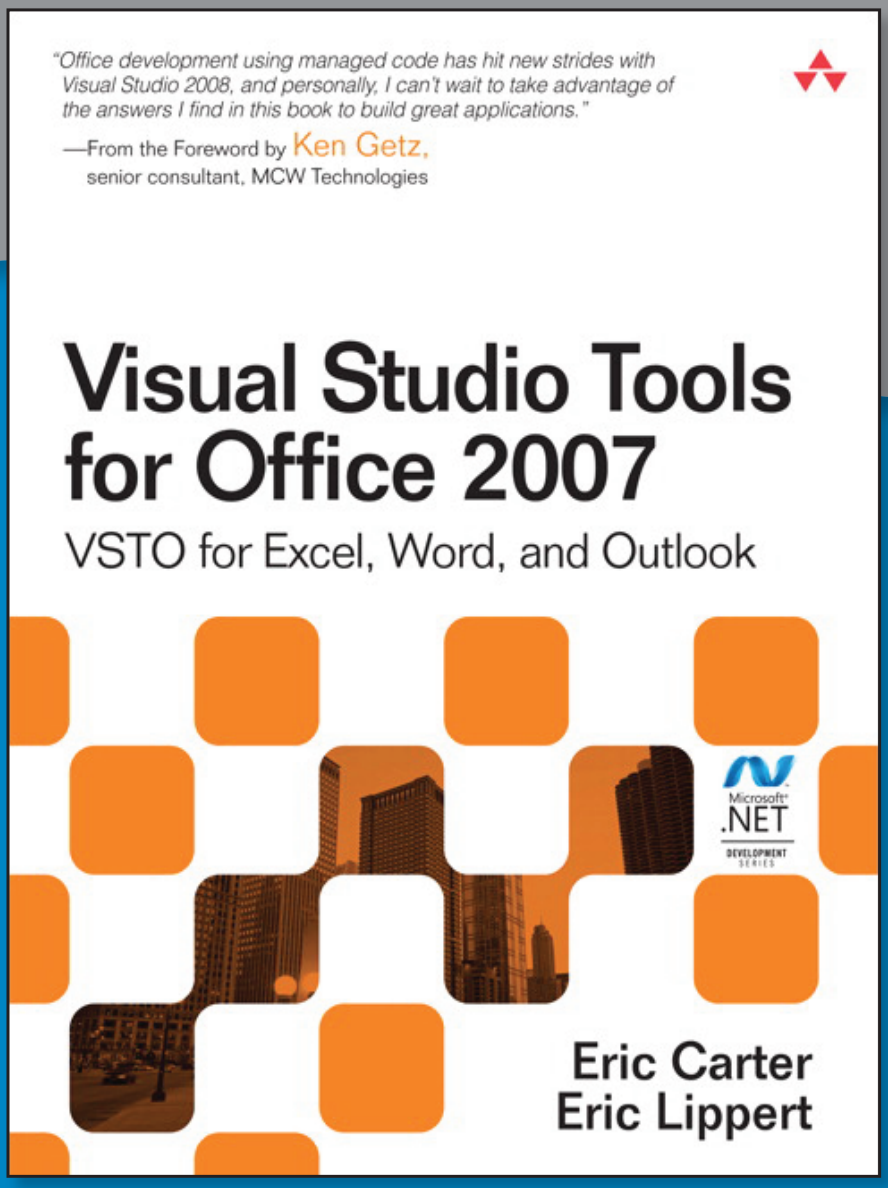
Scattered throughout this chapter are references to some of the important benefits of LINQ that emerge from these building blocks. Although these benefits were mentioned throughout this chapter, I'll bring them together here in one place as a way of reviewing and summarizing the material discussed in this chapter:

- Because LINQ is integrated into the C# language, it provides syntax highlighting and IntelliSense. These features make it easy to write accurate queries and to discover mistakes at design time.
- Because LINQ queries are integrated into the C# language, it is possible for you to write code much faster than if you were writing old-style queries. In some cases, developers have seen their development time cut in half.
- The integration of queries into the C# language also makes it easy for you to step through your queries with the integrated debugger.
- The hierarchical feature of LINQ allows you to easily see the relationship between tables, thereby making it easy to quickly compose queries that join multiple tables.
- The unitive foundation of LINQ allows you to use a single LINQ syntax when querying multiple data sources. This allows you to get up to speed on new technologies much more quickly. If you know how to use LINQ to Objects, it is not hard to learn how to use LINQ to SQL, and it is relatively easy to master LINQ to XML.
- Because LINQ is extensible, you can use your knowledge of LINQ to make new types of data sources queriable.
- After creating or discovering a new LINQ provider, you can leverage your knowledge of LINQ to quickly understand how to write queries against these new data sources.
- Because LINQ is composable, you can easily join multiple data sources in a single query, or in a series of related queries.

- The composable feature of LINQ also makes it easy to break complex problems into a series of short, comprehensible queries that are easy to debug.
- The transformational features of LINQ make it easy to convert data of one type into a second type. For instance, you can easily transform SQL data into XML data using LINQ.
- Because LINQ is declarative, it usually allows you to write concise code that is easy to understand and maintain.
- The compiler and provider translate declarative code into the code that is actually executed. As a rule, LINQ knows more than the average developer about how to write highly optimized, efficient code. For instance, the provider might optimize or reduce nested queries.
- LINQ is a transparent process, not a black box. If you are concerned about how a particular query executes, you usually have a way to examine what is taking place and to introduce optimizations into your query.

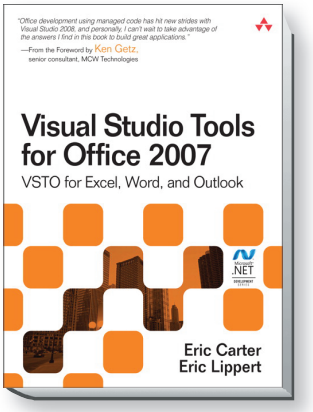
This chapter touched on many other benefits of LINQ. These are described throughout this book. This entire text is designed to make you aware of the benefits that LINQ can bring to your development process. It also shows you how to write code that makes those benefits available to you and the other developers on your team.

The more you understand LINQ, the more useful it will be to you. As I have dug more deeply into this technology, I have found myself integrating LINQ into many different parts of my development process. When I use LINQ, I can get more work done in less time. The more I use it, the more completely these benefits accrue.



Eric Carter
Eric Lippert

BUY ME



Visual Studio Tools for Office 2007

VSTO for Excel, Word, and Outlook

Visual Studio Tools for Office 2007: VSTO for Excel, Word, and Outlook is the definitive book on VSTO 2008 programming, written by the inventors of the technology. VSTO is a set of tools that allow professional developers to use the full power of Microsoft Visual Studio 2008 and the .NET Framework to program against Microsoft Office 2007.

This book delivers in one place all the information you need to succeed using VSTO to program against Word 2007, Excel 2007, and Outlook 2007, and provides the necessary background to customize Visio 2007, Publisher 2007, PowerPoint 2007, and InfoPath 2007. It introduces the Office 2007 object models, covers the most commonly used objects in those object models, and will help you avoid the pitfalls caused by the COM origins of the Office object models. Developers who wish to program against Office 2003 should consult Carter and Lippert's previous book, Visual Studio Tools for Office.

In VSTO 2008, you can build add-ins for all the major Office 2007 applications, build application-level custom task panes, customize the new Office Ribbon, modify Outlook's user interface using Forms Regions, and easily deploy everything you build using ClickOnce.

Carter and Lippert cover their subject matter with deft insight into the needs of .NET developers learning VSTO, based on the deep knowledge that comes from the authors' unique perspective of living and breathing VSTO for the past three years. This book

Explains the architecture of Microsoft Office programming and introduces the object models

Covers the main ways Office applications are customized and extended

Explores the ways of customizing Excel, Word, and Outlook, and plumbs the depths of programming with their events and object models

Introduces the VSTO programming model

Teaches how to use Windows Forms and WPF in VSTO and how to work with the Document Actions Pane and application-level task panes

Delves into VSTO data programming and server data scenarios

Teaches ClickOnce VSTO deployment

This is the one book you need to succeed in programming against Office 2007.

AVAILABLE
- BOOK: 9780321533210
- SAFARI ONLINE
- EBOOK: 032153333X
- KINDLE: 0321533348

About the Authors

Eric Carter is a development manager on the Visual Studio team at Microsoft. He helped invent, design, and implement many of the features that are in VSTO today. Previously at Microsoft he worked on Visual Studio for Applications, the Visual Studio Macros IDE, and Visual Basic for Applications for Office 2000 and Office 2003.

Eric Lippert's primary focus during his twelve years at Microsoft has been on improving the lives of developers by designing and implementing useful programming languages and development tools. He has worked on the Windows Scripting family of technologies, Visual Studio Tools for Office, and, most recently, on the C# compiler team.

Addison Wesley

informit.com/aw

3
Programming Excel

Ways to Customize Excel

Excel is the application most frequently programmed against in the Office family. Excel has a very rich object model with 247 objects that combined have more than 5,000 properties and methods. It supports several models for integrating your code, including add-ins and code behind documents. Most of these models were originally designed to allow the integration of COM components written in Visual Basic 6, VBA, C, or C++. However, through COM interop, managed objects written in C# or Visual Basic can masquerade as COM objects and participate in most of these models. This chapter briefly considers several of the ways that you can integrate your code with Excel and refers you to other chapters that discuss these approaches in more depth. This chapter also explores building user-defined functions for Excel and introduces the Excel object model.

New Objects in Excel 2007

Excel 2007 introduces 51 new objects to the object model. These new objects are, listed alphabetically: AboveAverage, Action, Actions, ChartFormat, ChartView, ColorScale, ColorScaleCriteria, ColorScaleCriterion,

ColorStop, ColorStops, ConditionValue, Connections, Databar, DialogSheetView, FormatColor, HeaderFooter, Icon, IconCriteria, IconCriterion, IconSet, IconSetCondition, IconSets, LinearGradient, ModuleView, MultiThreadedCalculation, ODBCConnection, OLEDBConnection, Page, Pages, PivotAxis, PivotFilter, PivotFilters, PivotLine, PivotLineCells, PivotLines, Ranges, RectangularGradient, Research, ServerViewableItems, SheetViews, Sort, SortField, SortFields, TableStyle, TableStyleElement, TableStyleElements, TableStyles, Top10, UniqueValues, WorkbookConnection, and WorksheetView.

Automation Executable

As mentioned in Chapter 2, "Introduction to Office Solutions," the simplest way to integrate with Excel is to start Excel from a console application or Windows Forms application and automate it from that external program. Chapter 2 provides a sample of an automation executable that automates Word and an automation executable that automates Excel.

VSTO Add-Ins

When building add-ins for Excel, you have two choices: You can build a COM add-in (sometimes called a shared add-in) or a VSTO Excel add-in. A VSTO Excel add-in solves many of the problems associated with COM add-in development and is the preferred model for Excel 2003 and Excel 2007 add-in development. The only time you would want to consider building a COM add-in instead is if you need to target versions of Excel older than Excel 2003.

An add-in is typically written to add application-level functionality—functionality that is available to any workbook opened by Excel. For example, you might write an add-in that adds a Ribbon button to convert a currency in the selected Excel worksheet cell to another currency based on current exchange rates.

Excel has a COM Add-Ins dialog box that enables users to turn COM and VSTO add-ins on and off. To access the COM Add-Ins dialog box, you must perform the following steps:

1. Click the Office menu (the large circle with the Office logo on it in the top-left corner of the Excel window), and choose Excel Options from the bottom of the menu that drops down.
2. Click the Add-Ins tab on the left side of the Excel Options dialog box.
3. Choose COM Add-Ins from the drop-down Manage menu, and click the Go button.

After you complete these steps, the COM Add-Ins dialog box appears. Figure 3-1 shows this dialog box.

The COM Add-ins dialog box shows both older-style COM add-ins and newer-style VSTO add-ins. In Figure 3-1, you can tell that ExcelAddIn1 is a VSTO add-in because of the Location information shown below; rather than showing the path to a DLL, which would indicate a COM add-in, the dialog box shows the path to a .vsto file, which indicates a VSTO add-in.

You can add COM add-ins by using the Add button and remove them by using the Remove button. Typically, you will not have your users use this dialog box to manage COM add-ins. Instead, you will install and remove a COM add-in by manipulating registry settings with the installer you create for your COM add-in. You can't add VSTO add-ins by using the Add button in the COM Add-Ins dialog box; you must install them by double-clicking a .vsto file or a setup.exe file associated with the VSTO add-in.

Excel discovers the installed add-ins by reading from the registry. You can view the registry on your computer by going to the Windows Start

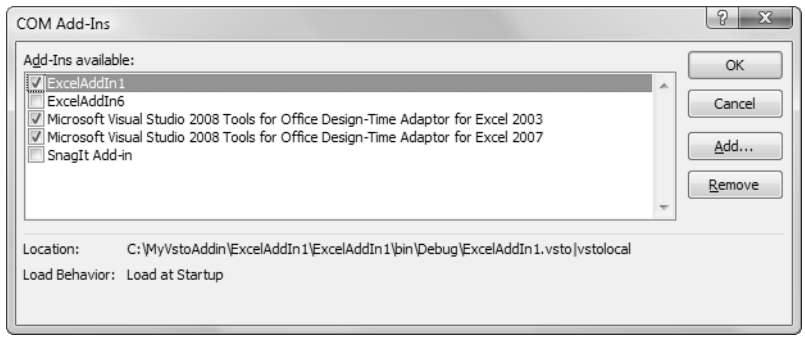


Figure 3-1: The COM Add-Ins dialog box in Excel.

menu and choosing Run. In the Run dialog box, type `regedit` for the program to run, and then click the OK button. In Vista, type `regedit` in the search box that appears at the bottom of the Windows Start menu. Excel looks for VSTO and COM add-ins in the registry keys under HKEY_CURRENT_USER\Software\Microsoft\Office\Excel\Addins. Excel also looks for COM add-ins in the registry keys under HKEY_LOCAL_MACHINE\Software\Microsoft\Office\Excel\Addins. COM add-ins registered under HKEY_LOCAL_MACHINE are not shown in the COM Add-Ins dialog box and cannot be turned on or off by users. It is recommended you do not register your COM add-in under HKEY_LOCAL_MACHINE because it hides the COM add-in from the user.

VSTO add-ins are discussed in detail in Chapter 12, "The VSTO Programming Model."

Automation Add-Ins

Automation add-ins are classes that are registered in the registry as COM objects that expose public functions that can be used in Excel formulas. Automation add-ins that have been installed are shown in the Add-Ins dialog box, which you can access by following these steps:

1. Click the Office menu (the large circle with the Office logo on it in the top-left corner of the Excel window), and choose Excel Options from the bottom of the menu that drops down.
2. Click the Add-Ins tab on the left side of the Excel Options dialog box.
3. Choose Excel Add-Ins from the drop-down Manage menu, and click the Go button.

This chapter examines automation add-ins in more detail during the discussion of how to create user-defined Excel functions for use in Excel formulas.

Visual Studio Tools for Office Code Behind

Visual Studio 2008 Tools for Office 2007 (VSTO 3.0) enables you to put C# or Visual Basic code behind Excel templates and workbooks. The code-behind model was designed from the ground up for managed code, so this

model is the most ".NET" of all the models used to customize Excel. This model is used when you want to customize the behavior of a particular workbook or a particular set of workbooks created from a common template. For example, you might create a template for an expense-reporting workbook that is used whenever anyone in your company creates an expense report. This template can add commands and functionality that are always available when the workbook created with it is opened.

VSTO's support for code behind a workbook is discussed in detail in Part III of this book.

Smart Documents and XML Expansion Packs

Smart documents are another way to associate your code with an Excel template or workbook. Smart documents rely on attaching an XML schema to a workbook or template and associating your code with that schema. The combination of the schema and associated code is called an XML Expansion Pack. An XML Expansion Pack can be associated with an Excel workbook by choosing XML Expansion Packs from the XML menu in the Data menu. Figure 3-2 shows the XML Expansion Packs dialog box.

When an XML Expansion Pack is attached to a workbook, Excel loads the associated code and runs it while that workbook is opened. Smart document solutions can create a custom user interface in the Document Actions task pane. Figure 3-3 shows a custom Document Actions task pane in Excel.

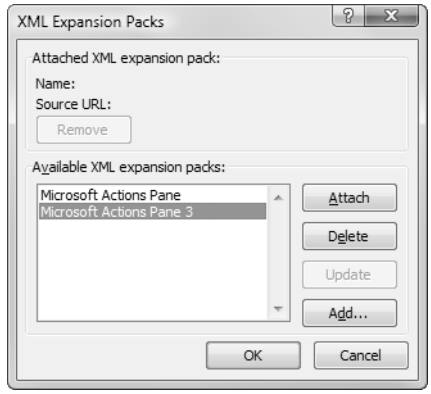


Figure 3-2: The XML Expansion Packs dialog box in Excel.

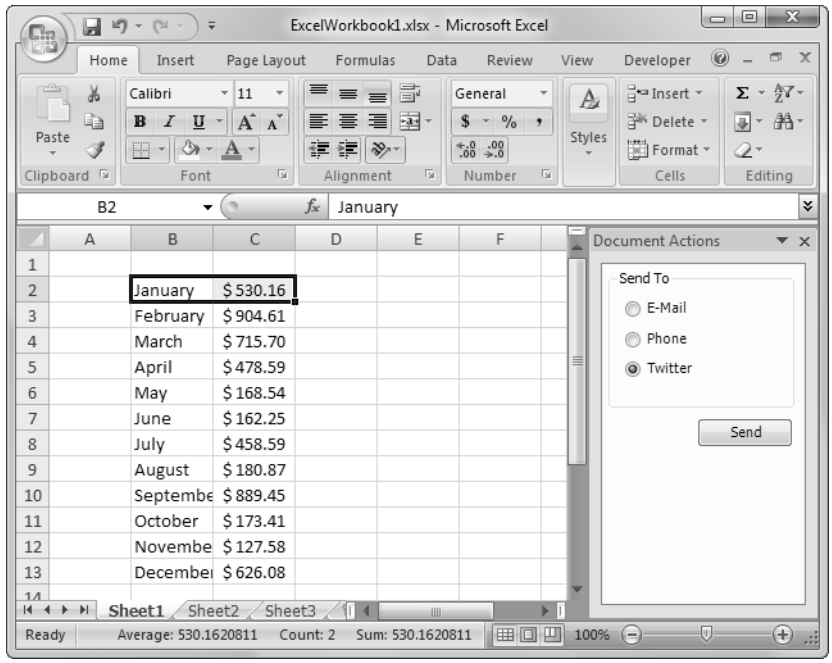


Figure 3-3: A custom Document Actions task pane in Excel.

It is possible to write smart document solutions "from scratch" in C# or Visual Basic. This book does not cover this approach. Instead, this book focuses on the VSTO approach, which was designed to make smart document development much easier and allow you to create a custom Document Actions task pane using Windows Forms. Chapter 14, "Working with Document-Level Actions Pane," discusses this capability in more detail.

Smart Tags

Smart Tags enable a pop-up menu to be displayed containing actions relevant for a recognized piece of text in a workbook. You can control the text that Excel recognizes and the actions that are made available for that text by creating a Smart Tag DLL or by using VSTO code behind a document or a VSTO add-in.

A Smart Tag DLL contains two types of components that are used by Excel: a recognizer and associated actions. A recognizer determines what

text in the workbook is recognized as a Smart Tag. An action corresponds to a menu command displayed in the pop-up menu.

A recognizer could be created that tells Excel to recognize stock-ticker symbols (such as the MSFT stock symbol) and display a set of actions that can be taken for that symbol: buy, sell, get the latest price, get history, and so on. A "get history" action, for instance, could launch a Web browser to show a stock history Web page for the stock symbol that was recognized.

When a recognizer recognizes some text, Excel displays a little triangle in the lower-right corner of the associated cell. If the user hovers over the cell, a pop-up menu icon appears next to the cell that the user can click to drop down a menu of actions for the recognized piece of text. Figure 3-4 shows an example menu. When an action is selected, Excel calls back into the associated action to execute your code.

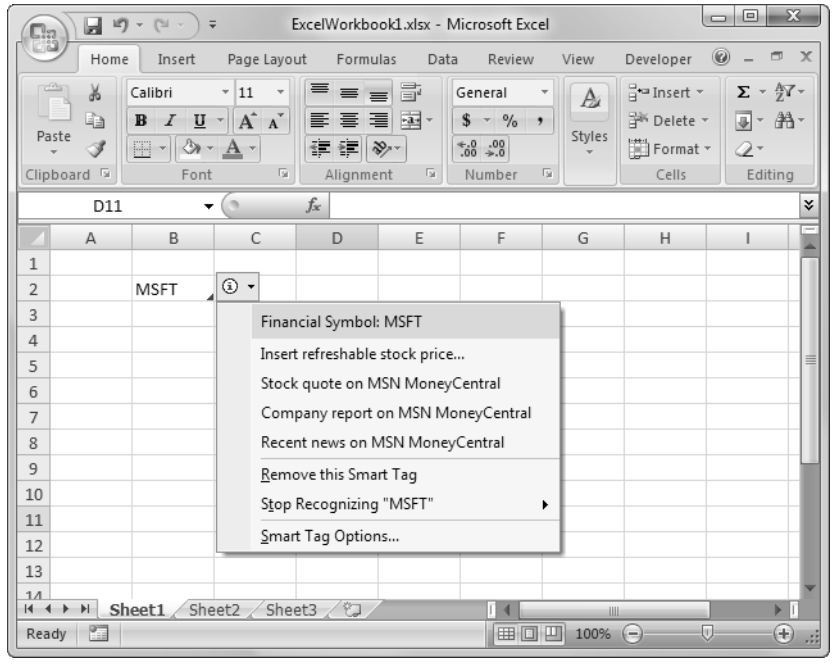


Figure 3-4: Smart Tags in Excel.

Smart Tags are managed from the Smart Tags page of the AutoCorrect dialog box, as shown in Figure 3-5. You can display the Smart Tags page by following these steps:

1. Click the Office button (the large circle with the Office logo on it in the top-left corner of the Excel window), and choose Excel Options from the bottom of the menu that drops down.
2. Click the Proofing tab on the left side of the Excel Options dialog box.
3. Click the AutoCorrect Options button.
4. Click the Smart Tags page.

On the Smart Tags page of the AutoCorrect dialog box, the user can turn on and off individual recognizers as well as control other options relating to how Smart Tags display in the workbook.

It is possible to write Smart Tags from scratch in C# or Visual Basic, and the first edition of this book covers this approach. VSTO provides a simpler model for creating a Smart Tag that works at the workbook, template, and (new in VSTO 2008 SP1) add-in levels. Chapter 18, "Working with

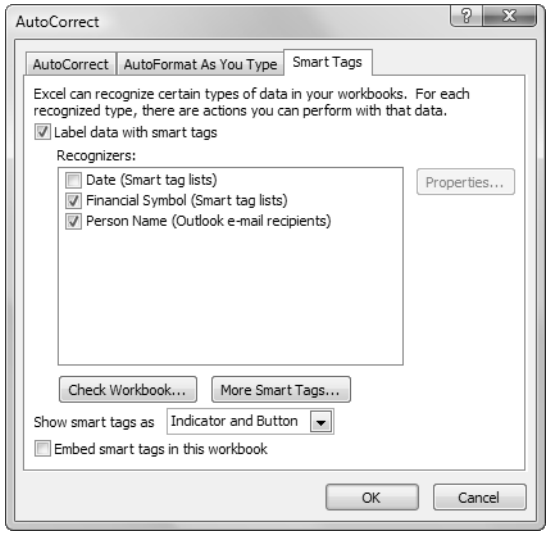


Figure 3-5: The Smart Tags page in the AutoCorrect dialog box.

Smart Tags in VSTO," describes the VSTO model for working with Smart Tags in more detail.

XLA Add-Ins

Also listed in the Excel Add-Ins dialog box (follow the steps for showing the dialog box for Automation add-ins, described earlier in this chapter) are XLA add-ins. An XLA add-in starts life as a workbook that has VBA code behind it. The developer can then save the workbook as an XLA or Excel add-in file by using Save As from the File menu and selecting XLA as the file format. An XLA file acts as an application-level add-in in the form of an invisible workbook that stays open for the lifetime of Excel. Although it is possible to save a workbook customized with VSTO as an XLA file, many of the features of VSTO do not work when the workbook is converted to an XLA file. Some of the features that do not work include VSTO's support for the Document Actions task pane and for Smart Tags. For this reason, Microsoft does not support or recommend saving a workbook customized with VSTO as an XLA file. Therefore, this book does not cover it further.

Server-Generated Documents

VSTO enables you to write code on the server that populates an Excel workbook with data without starting Excel on the server. You might create an ASP.NET page that reads some data out of a database and then puts it in an Excel workbook and returns that workbook to the client of the Web page. VSTO provides a class called ServerDocument that makes this process easy. Chapter 20, "Server Data Scenarios," describes generating documents on the server using the ServerDocument class.

You can also use the XML features and file formats of Office to generate Excel workbooks on the server—an approach that is more complex but that allows you to generate more complex documents.

Research Services

Excel has a task pane, called the Research task pane, that enables you to enter a search term and search various sources for that search term. To show the Research task pane, click the Research button in the Review tab. Figure 3-6 shows the Research task pane.

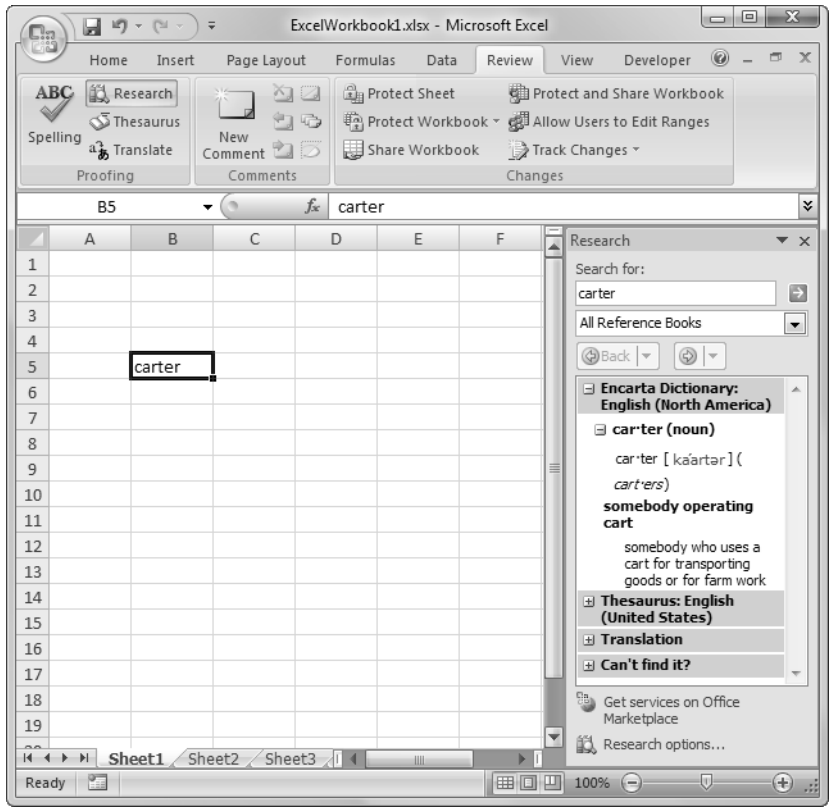


Figure 3-6: The Research task pane.

Excel enables developers to write a special Web service, called a research service, that implements a set of Web methods defined by Excel. A research service can be registered with Excel and used in Office's Research task pane. For example, you might write a research service that searches for a search term in a company database. Chapter 6, "Programming Word," discusses creating a research service in more detail.

Programming User-Defined Functions

Excel enables the creation of user-defined functions that can be used in Excel formulas. A developer must create a special kind of DLL called an XLL. Excel also allows you to write custom functions in VBA that can be

used in Excel formulas. Unfortunately, Excel does not support writing an XLL that uses managed code.

Building a Managed Automation Add-In That Provides User-Defined Functions

Fortunately, there is an easier way to create a user-defined function that does not require you to create an XLL. Excel 2007 supports a customization technology called an automation add-in that can easily be created in C# or Visual Basic.

First, launch Visual Studio and create a new C# class library project. Name the project AutomationAddin. In your Class1.cs file created for you in the new project, replace Class1 with the code shown in Listing 3-1. Replace the GUID string in the listing with your own GUID by choosing Tools > Generate GUID. In the Generate GUID dialog box, pick option 4, Registry Format. Click the Copy button to put the new GUID string on the clipboard. Click Exit to exit the Generate GUID tool. Finally, select the GUID string in the listing (`"5268ABE2-9B09-439d-BE97-2EA60E103EF6"`), and replace it with your new GUID string. You'll also have to remove the { } brackets that get copied as part of the GUID.

Listing 3-1 defines a class called `MyFunctions` that implements a function called `MultiplyNTimes`. We will use this function as a custom formula. Our class also implements `RegisterFunction` and `UnregisterFunction`, which are attributed with the ComRegisterFunction attribute and ComUnregister-Function attribute respectively. The `RegisterFunction` will be called when the assembly is registered for COM interop. The `UnregisterFunction` will be called when the assembly is unregistered for COM interop. These functions put a necessary key in the registry (the Programmable key) that allows Excel to know that this class can be used as an automation add-in. `Register-Function` also works around another issue that occurs when registering the class. Excel needs a full path to the .NET component that loads the automation add-in—a component called mscoree.dll. So `RegisterFunction` contains some code to put the full path to mscoree.dll in the registry.

Listing 3-1: A C# Class Called MyFunctions That Exposes a User-Defined Function MultiplyNTimes

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Text;
using System.Runtime.InteropServices;
using Microsoft.Win32;

namespace AutomationAddin
{
  // Replace the GUID below with your own GUID that
  // you generate using Create GUID from the Tools menu
  [Guid("5268ABE2-9B09-439d-BE97-2EA60E103EF6")]
  [ClassInterface(ClassInterfaceType.AutoDual)]
  [ComVisible(true)]
  public class MyFunctions
  {
    public MyFunctions()
    {
    }

    public double MultiplyNTimes(double number1,
      double number2, double timesToMultiply)
    {
      double result = number1;
      for (double i = 0; i < timesToMultiply; i++)
      {
        result = result * number2;
      }
      return result;
    }

    [ComRegisterFunctionAttribute]
    public static void RegisterFunction(Type type)
    {
      Registry.ClassesRoot.CreateSubKey(
        GetSubKeyName(type, "Programmable"));
      RegistryKey key = Registry.ClassesRoot.OpenSubKey(
        GetSubKeyName(type, "InprocServer32"), true);
      key.SetValue("",
        System.Environment.SystemDirectory + @"\mscoree.dll",
        RegistryValueKind.String);
    }

    [ComUnregisterFunctionAttribute]
    public static void UnregisterFunction(Type type)
    {
      Registry.ClassesRoot.DeleteSubKey(
        GetSubKeyName(type, "Programmable"), false);
    }

    private static string GetSubKeyName(Type type,
      string subKeyName)
    {
```

```
      System.Text.StringBuilder s =
        new System.Text.StringBuilder();
      s.Append(@"CLSID\{");
      s.Append(type.GUID.ToString().ToUpper());
      s.Append(@"}\");
      s.Append(subKeyName);
      return s.ToString();
    }
  }
}
```

With this code written (remember to replace the GUID in the listing with your own GUID that you generate by choosing Tools > Generate GUID), you need to configure the project to be registered for COM interop so that Excel can see it. Go to the properties for the project by double-clicking the Properties node in Solution Explorer. In the properties designer that appears, click the Build tab, and check the check box labeled Register for COM Interop, as shown in Figure 3-7. This step causes Visual Studio to register the assembly for COM interop when the project is built.

If you are running under Vista or later, you need to run Visual Studio as administrator, because registering for COM interop requires administrative privileges. If you aren't already running Visual Studio as administrator, save your project and exit Visual Studio. Then find the Visual Studio 2008 icon in the Start menu, right-click it, and choose Run as Administrator (see Figure 3-8). When you're running Visual Studio as administrator, reopen your project and choose Build > Rebuild Solution. Visual Studio will do the necessary registration to make your class visible to Excel.

Using Your Managed Automation Add-In in Excel

Now that the add-in is built and registered, to load the managed automation add-in into Excel, follow these steps:

1. Launch Excel, and click the Microsoft Office button in the top-left corner of the window.
2. Choose Excel Options.
3. Click the Add-Ins tab in the Excel Options dialog box.
4. Choose Excel Add-Ins from the combo box labeled Manage, and click the Go button.

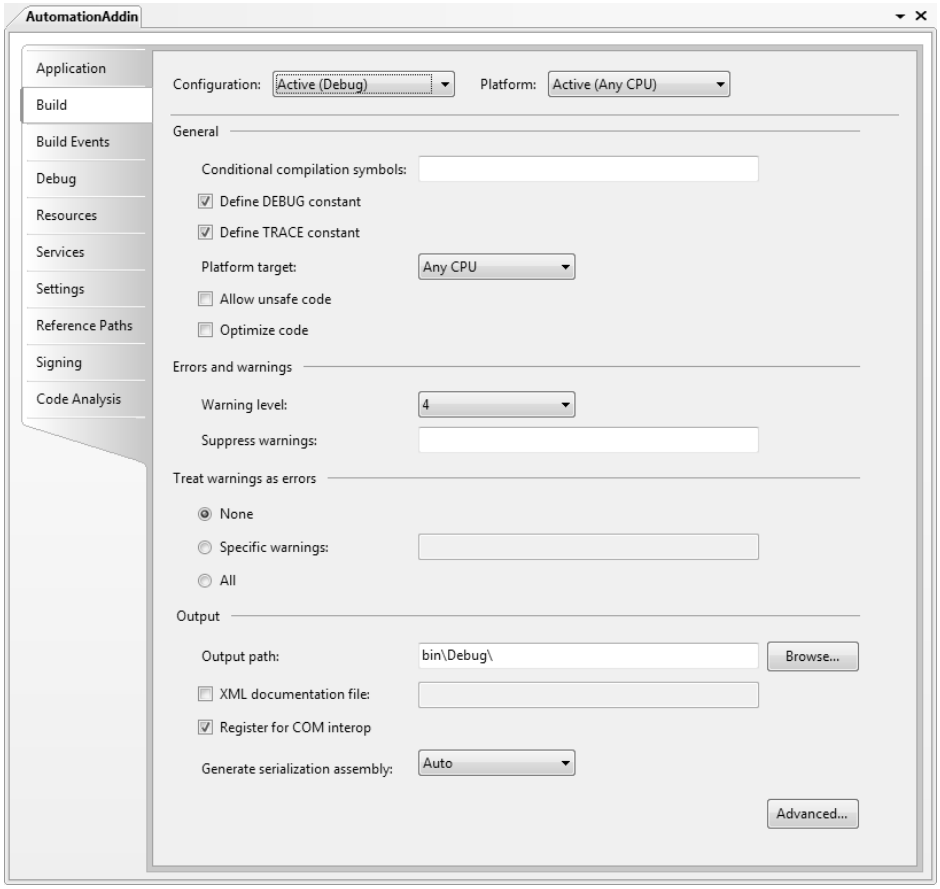


Figure 3-7: Checking the Register for COM Interop check box in the project properties designer.

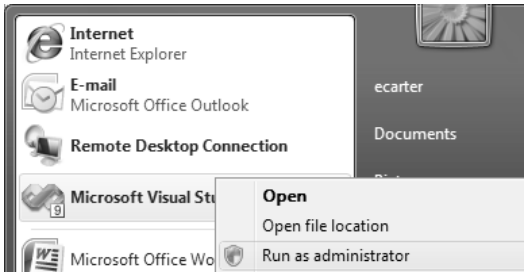


Figure 3-8: Launching Visual Studio 2008 as administrator under Vista.

5. Click the Automation button in the Add-Ins dialog box.
6. Look through the list of Automation Servers, and find the class you created; it will be listed as AutomationAddin.MyFunctions, as shown in Figure 3-9.

By clicking OK in this dialog box, you have added the `Automation-Addin.MyFunctions` class to the list of installed automation add-ins, as shown in Figure 3-10.

Now, try to use the function `MultiplyNTimes` in an Excel formula. First create a simple spreadsheet that has a number, a second number to multiply the first by, and a third number for how many times you want to multiply the first number by the second number. Figure 3-11 shows the spreadsheet.

Click an empty cell in the workbook below the numbers, and then click the Insert Function button (the button with the "fx" label) in the formula bar. From the dialog box of available formulas, drop down the "Or select a category" drop-down box and choose `AutomationAddin.MyFunctions`. Then click the `MultiplyNTimes` function, as shown in Figure 3-12.

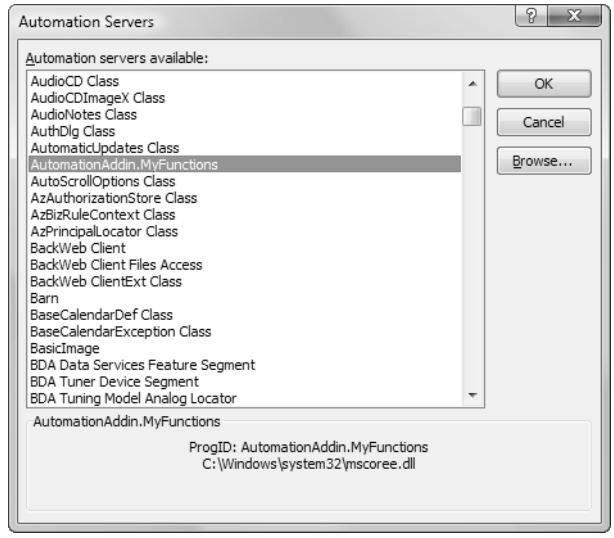


Figure 3-9: Selecting AutomationAddin.MyFunctions from the Automation Servers dialog box.

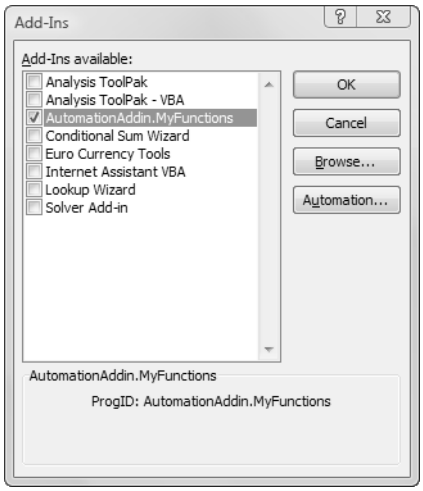


Figure 3-10: AutomationAddin.MyFunctions is now installed.

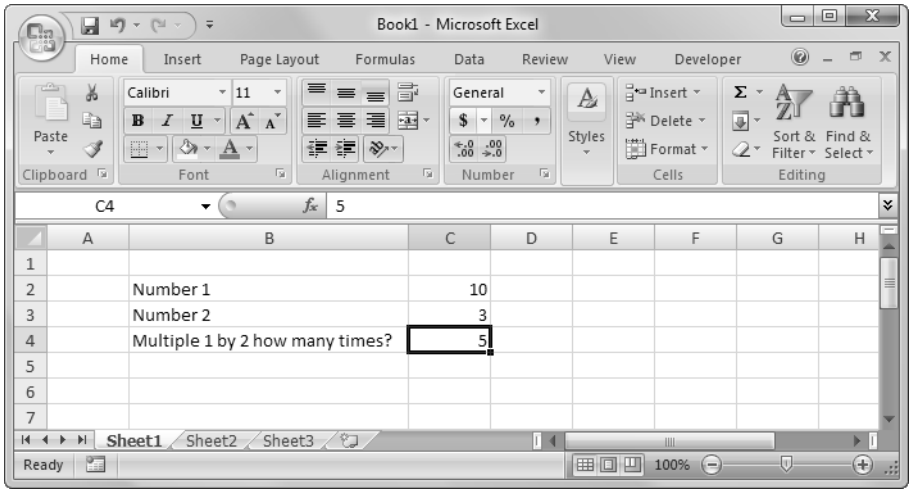


Figure 3-11: A simple spreadsheet to test the custom formula in.

When you click the OK button, Excel pops up a dialog box to help select function arguments from cells in the spreadsheet, as shown in Figure 3-13.

After you have selected function arguments from the appropriate cells, click OK to create the final spreadsheet, as shown in Figure 3-14, with the custom formula in cell C5.

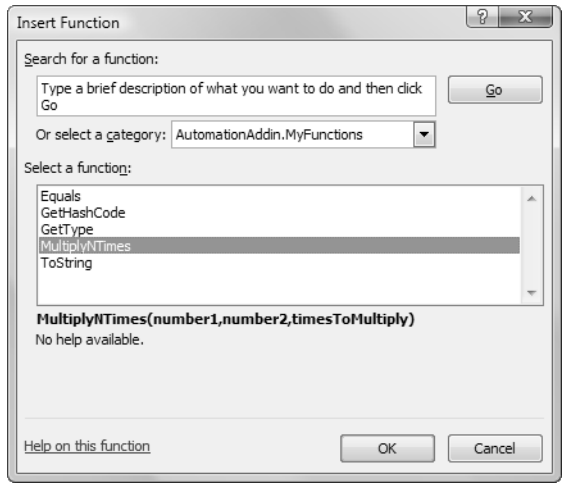


Figure 3-12: Picking MultiplyNTimes from the Insert Function dialog box.

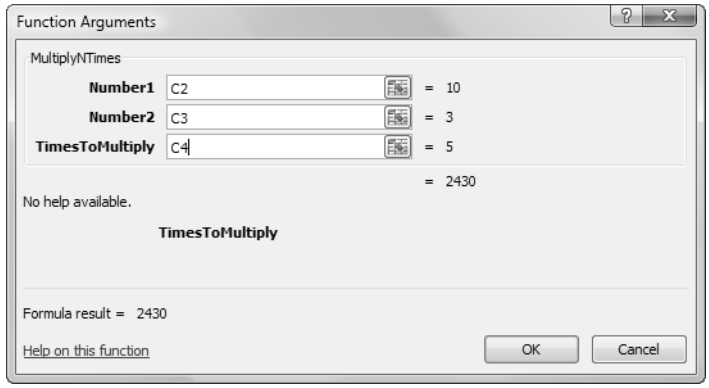


Figure 3-13: Setting arguments using the Function Arguments dialog box.

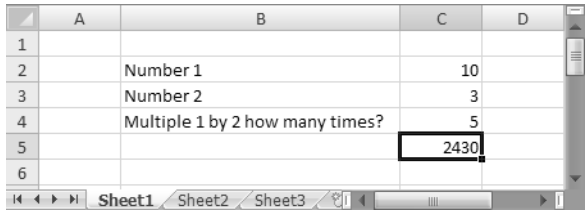


Figure 3-14: The final spreadsheet.

Non-English Locales and Excel

Excel and .NET have some special issues when running in a non-English locale that may cause an automation add-in to fail. For more information, see the section "Special Excel Issues" in Chapter 5, "Working with Excel Objects." VSTO add-ins have some additional features that protect you from these issues.

Some Additional User-Defined Functions

You might experiment with other functions that could be used in an Excel formula. For example, Listing 3-2 shows several other functions you could add to your `MyFunctions` class. To use Listing 3-2, you must add a reference to the Excel 12.0 Object Library and add the code `using Excel = Microsoft.Office.Interop.Excel` to the top of your class file. Note in particular that when you declare a parameter as an object, Excel passes you a Range object. Also note how optional parameters are supported by the `AddNumbers` function. When a parameter is omitted, `System.Type.Missing` is passed as the value of the parameter. Also be sure to restart Excel so that it loads the newest version of your automation add-in.

Listing 3-2: Additional User-Defined Function That Could Be Added to the MyFunctions Class

```
public string GetStars(int number)
{
  System.Text.StringBuilder s =
    new System.Text.StringBuilder();
  s.Append('*', number);
  return s.ToString();
}

public double AddNumbers(double number1,
  [Optional] object number2, [Optional] object number3)
{
  double result = number1;

  if (number2 != System.Type.Missing)
  {
    Excel.Range r2 = number2 as Excel.Range;
    double d2 = Convert.ToDouble(r2.Value2);
    result += d2;
  }
```

```
  if (number3 != System.Type.Missing)
  {
    Excel.Range r3 = number3 as Excel.Range;
    double d3 = Convert.ToDouble(r3.Value2);
    result += d3;
  }

  return result;
}

public double CalculateArea(object range)
{
  Excel.Range r = range as Excel.Range;
  return Convert.ToDouble(r.Width) *
    Convert.ToDouble(r.Height);
}

public double NumberOfCells(object range)
{
  Excel.Range r = range as Excel.Range;
  return r.Cells.Count;
}

public string ToUpperCase(string input)
{
  return input.ToUpper();
}
```

Debugging User-Defined Functions in Managed Automation Add-Ins

You can debug a C# class library project that is acting as an automation add-in by setting Excel to be the program your class library project starts when you debug. Show the properties for the project by double-clicking the Properties node under the project node in Solution Explorer. In the properties designer that appears, click the Debug tab, and in the Start external program text box, type the full path to Excel.exe, as shown in Figure 3-15. Now, set a breakpoint on one of your user functions, press F5, and use the function in the spreadsheet. The debugger will stop in the implementation of your user function where the breakpoint was set.

Deploying Managed Automation Add-Ins

To deploy an automation add-in, right-click your solution in Solution Explorer and choose New Project from the Add menu. From the Add New Project dialog box, choose Setup Project from Other Project Types\Setup and Deployment in the Project Types tree.

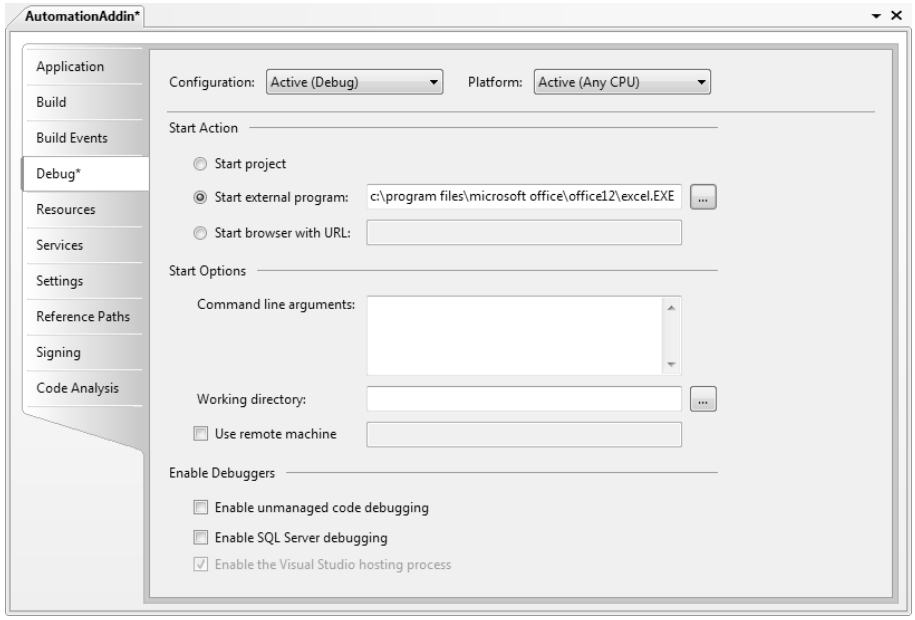


Figure 3-15: Setting Debug options to start Excel.

Right-click the newly added setup project in Solution Explorer and choose Project Output from the Add menu. From the Add Project Output Group dialog box, choose the AutomationAddin project and select Primary Output, as shown in Figure 3-16.

You must also configure the install project to register the managed object for COM interop at install time. To do this, click the Primary output from AutomationAddin node in the setup project. In the Properties window for the primary output (our C# DLL), make sure that Register is set to `vsdrpCOM`.

Introduction to the Excel Object Model

Regardless of the approach you choose to integrate your code with Excel, you will eventually need to talk to the Excel object model to get things done. It is impossible to completely describe the Excel object model in this book, but we try to make you familiar with the most important objects in the Excel object model and show some of the most frequently used methods, properties, and events on these objects.

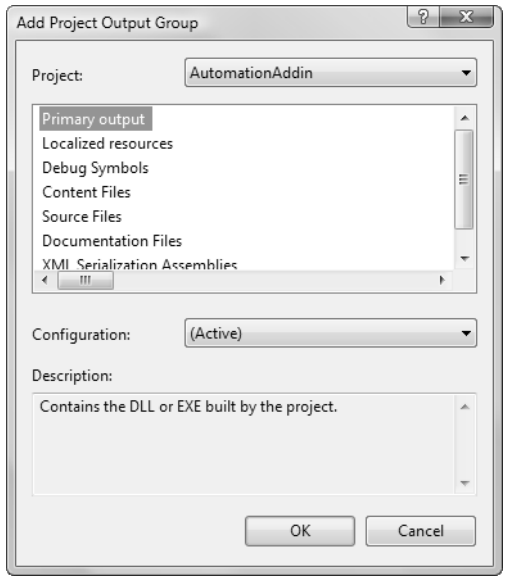


Figure 3-16: Adding the Primary Output of the AutomationAddin project to the setup project.

The first step in learning the Excel object model is getting an idea for the basic structure of the object model hierarchy. Figure 3-17 shows the most critical objects in the Excel object model and their hierarchical relationship.

A Workbook object has a collection called Sheets. The Sheets collection can contain objects of type Worksheet or Chart. A Chart is sometimes

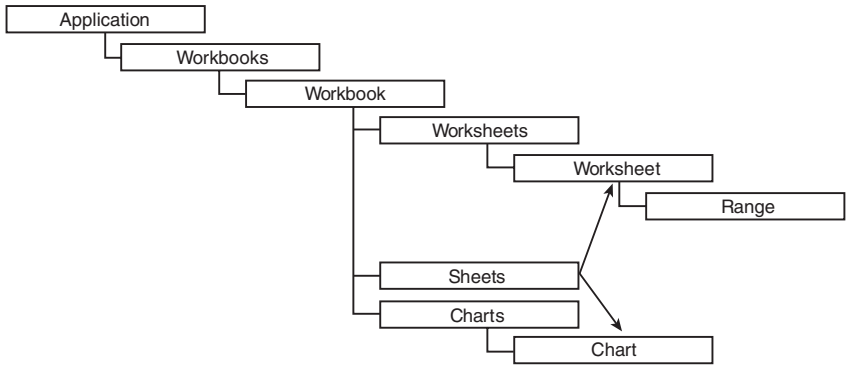


Figure 3-17: The basic hierarchy of the Excel object model.

called a chart sheet because it covers the entire area that a worksheet would cover. You can insert a chart sheet into a workbook by right-clicking the worksheet tabs in the lower-left corner of the Excel workbook and choosing Insert. Figure 3-18 shows the dialog box that appears. Note that two additional objects are found in the Sheets collection: MS Excel 4.0 macro sheets and MS Excel 5.0 dialog sheets. If you insert a macro sheet or dialog sheet into an Excel workbook, it is treated as a special kind of worksheet—there is not a special object model type corresponding to a macro sheet or a dialog sheet.

Because a workbook can contain these various kinds of objects, Excel provides several collections off of the Workbook object. The Worksheets collection contains just the Worksheet objects in the workbook. The Charts collection contains just the chart sheets in the workbook. The Sheets collection is a mixed collection of both. The Sheets collection returns members of the collection as type `object`—you must cast the returned object to a Worksheet or Chart. In this book, when we talk about an object that could be either a Worksheet or a Chart, we refer to it as a sheet.

Figure 3-19 shows a more complete hierarchy tree with the major objects associated with the objects in Figure 3-17. This starts to give you an

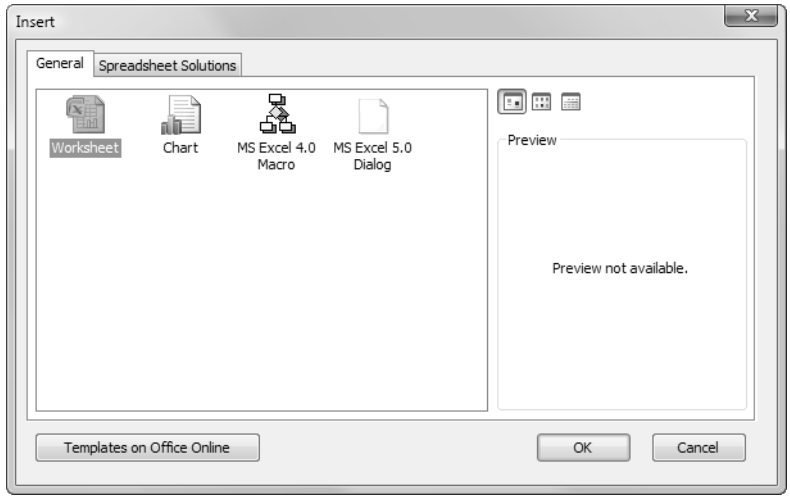


Figure 3-18: Inserting various kinds of "sheets" into an Excel Workbook.

Figure 3-19: A more detailed hierarchy of some major objects in the Excel object model.

idea of the extensive hierarchy of objects that is the Excel object model, especially when you realize that this diagram shows less than half of the objects available. The objects shown in gray are coming from the Microsoft.Office.Core namespace, which is associated with the Microsoft Office 12.0 PIA (office.dll). These objects are shared by all the Office applications.

Figure 3-20 shows the object hierarchy associated with Range, a very important object in Excel that represents a range of cells you want to work with in your code. We have already used the Range object in Listing 3-2.

Figure 3-21 shows the object hierarchy associated with Shape—a Shape represents things that float on the worksheet that are not cells, such as embedded buttons, drawings, comment bubbles, and so on.

Conclusion

This chapter introduced the various ways you can integrate your code into Excel. The chapter described how to build automation add-ins to create user-defined functions for Excel. You also learned the basic hierarchy of the Excel object model. Chapter 4, "Working with Excel Events," discusses the events in the Excel object model. Chapter 5, "Working with Excel Objects," covers the most important objects in the Excel object model.

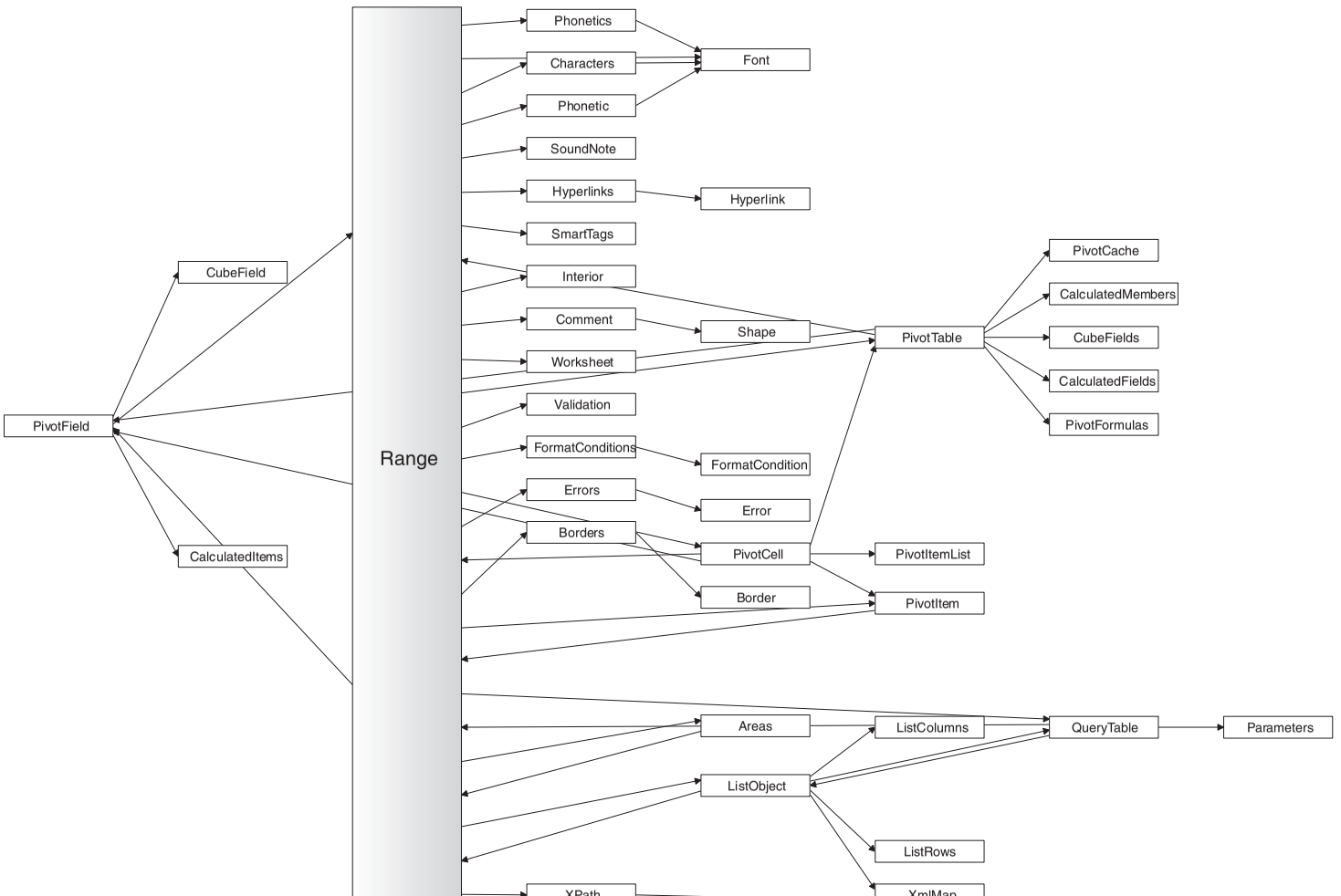
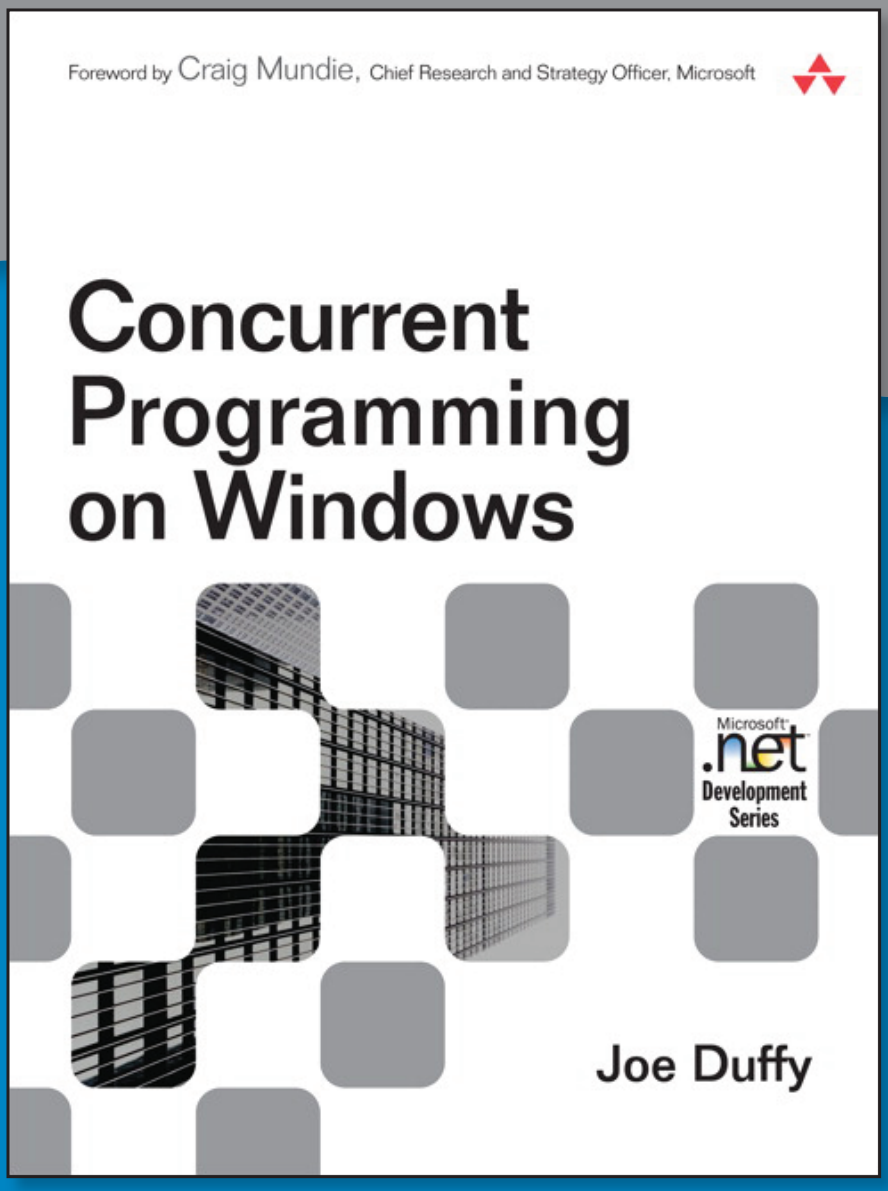


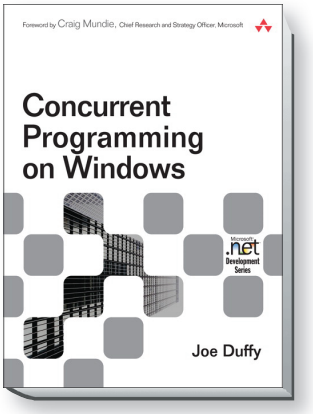
Figure 3-20: A more detailed hierarchy of objects associated with Range in the Excel object model.

Figure 3-21: A more detailed hierarchy of objects associated with Shape in the Excel object model.



Joe Duffy

BUY ME



Concurrent Programming on Windows

Author **Joe Duffy** has risen to the challenge of explaining how to write software that takes full advantage of concurrency and hardware parallelism. In Concurrent Programming on Windows, he explains how to design, implement, and maintain large-scale concurrent programs, primarily using C# and C++ for Windows.

Duffy aims to give application, system, and library developers the tools and techniques needed to write efficient, safe code for multicore processors. This is important not only for the kinds of problems where concurrency is inherent and easily exploitable–such as server applications, compute-intensive image manipulation, financial analysis, simulations, and AI algorithms–but also for problems that can be speeded up using parallelism but require more effort–such as math libraries, sort routines, report generation, XML manipulation, and stream processing algorithms.

Concurrent Programming on Windows has four major sections: The first introduces concurrency at a high level, followed by a section that focuses on the fundamental platform features, inner workings, and API details. Next, there is a section that describes common patterns, best practices, algorithms, and data structures that emerge while writing concurrent software. The final section covers many of the common system-wide architectural and process concerns of concurrent programming.

This is the only book you'll need in order to learn the best practices and common patterns for programming with concurrency on Windows and .NET.

AVAILABLE
- BOOK: 9780321434821
- SAFARI ONLINE
- EBOOK: 0321604423
- KINDLE: 0321604415

About the Author

Joe Duffy is the development lead, architect, and founder of the Parallel Extensions to the .NET Framework team at Microsoft. In addition to hacking code and managing a team of developers, he works on long-term vision and incubation efforts, such as language and type system support for concurrency safety. He previously worked on the Common Language Runtime team. Joe blogs regularly at www.bluebytesoftware.com/blog.

3
Threads

INDIVIDUAL PROCESSES ON Windows are sequential by default. Even on a multiprocessor machine, a program (by default) will only use one of them at a time. Running multiple processes at once creates concurrency at a very coarse level. Microsoft Word could be repaginating a document on one processor, while Internet Explorer downloads and renders a Web page on another, all while Windows Indexer is rebuilding search indexes on a third processor. This happens because each application is run inside its own distinct process with (one hopes) little interference between the two (again, one hopes), yielding better responsiveness and overall performance by virtue of running completely concurrently with one another.

The programs running inside of each process, however, are free to introduce additional concurrency. This is done by creating *threads* to run different parts of the program running inside a single program at once. Each Windows process is actually comprised of a single thread by default, but creating more than one in a program enables the OS to schedule many onto separate processors simultaneously. Coincidently, each .NET program is actually multithreaded from the start because the CLR garbage collector uses a separate *finalizer* thread to reclaim resources. As a developer, you are free to create as many additional threads as you want.

Using multiple threads for a single program can be done to run entirely independent parts of a program at once. This is classic **agents style** concurrency and, historically, has been used frequently in server-side

programs. Or, you can use threads to break one big task into multiple smaller pieces that can execute concurrently. This is **parallelism** and is increasingly important as commodity hardware continues to increase the number of available processors. Refer back to Chapter 1, Introduction, for a detailed explanation of this taxonomy.

Threads are the fundamental units of schedulable concurrency on the Windows platform and are available to native and managed code alike. This chapter takes a look at the essentials of scheduling and managing concurrency on Windows using threads. The APIs used to access threading in native and managed code are slightly different, but the fundamental architecture and OS support are the same. But before we go into the details, let's precisely define what a thread is and of what it consists. After that, we'll move on to how programs use them.

Threading from 10,001 Feet

A thread is in some sense just a virtual processor. Each runs some program's code as though it were independent from all other virtual processors in the system. There can be fewer, equal, or more threads than real processors on a system at any given moment due (in part) to the multitasking nature of Windows, wherein a user can run many programs at once, and the OS ensures that all such threads get a fair chance at running on the available hardware.

Given that this could be as much a simple definition of an OS process as a thread, clearly there has to be some interesting difference. And there is (on Windows, at least). Processes are the fundamental unit of concurrency on many UNIX OSs because they are generally lighter-weight than Windows processes. A Windows process always consists of at least one thread that runs the program code itself. But one process also may execute multiple threads during the course of its lifetime, each of which shares access to a set of process-wide resources. In short, having many threads in a single process allows one process to do many things at once. The resources shared among threads include a single virtual memory address space, permitting threads to share data and communicate easily by reading from and writing to common addresses and objects in memory. Shared resources also include

things associated with the Windows process, such as the handle table and security token information.

Most people get their first taste of threading by accident. Developers use a framework such as ASP.NET that calls their code on multiple threads simultaneously or write some GUI event code in Windows Forms, MFC, or Windows Presentation Foundation, in which there is a strong notion of particular data structures belonging to particular threads. (We discuss this fact and its implications in Chapter 16, Graphical User Interfaces.) These developers often learn about concurrency "the hard way" by accidentally writing unreliable code that crashes or by creating an unresponsive GUI by doing I/O on the GUI thread. Faced with such a situation, people are quick to learn some basic rules of thumb, often without deeply understanding the reasons behind them. This can give people a bad first impression of threads. But while concurrency is certainly difficult, threads are the key to exploiting new hardware, and so it's important to develop a deeper understanding.

What Is a Windows Thread?

We already discussed threads at a high level in previous chapters, but let's begin painting a more detailed picture.

Conceptually speaking, a thread is an execution context that represents in-progress work being performed by a program. A thread isn't a simple, physical thing. Windows must allocate and maintain a kernel object for each thread, along with a set of auxiliary data structures. But as a thread executes, some portion of its logical state is also comprised of hardware state, such as data in the processor's registers. A thread's state is, therefore, distributed among software and hardware, at least when it's running. Given a thread that is running, a processor can continue running it, and given a thread that is not running, the OS has all the information it needs so that it can schedule the thread to run on the hardware again.

Each thread is mapped onto a processor by the Windows thread scheduler, enabling the in-progress work to actually execute. Each thread has an instruction pointer (IP) that refers to the current executing instruction. "Execution" consists of the processor fetching the next instruction, decoding it, and issuing it, one instruction after another, from the thread's code,

incrementing the IP after ordinary instructions or adjusting it in other ways as branches and function calls occur. During the execution of some compiled code, program data will be routinely moved into and out of registers from the attached main memory. While these registers physically reside on the processor, some of this volatile state also abstractly belongs to the thread too. If the thread must be paused for any reason, this state will be captured and saved in memory so it can be later restored. Doing this enables the same IP fetch, decode, and issue process to proceed for the thread later as though it were never interrupted. The process of saving or restoring this state from and to the hardware is called a **context switch.**

During a context switch, the volatile processor state, which logically belongs to the thread, is saved in something called a **context.** The context switching behavior is performed entirely by the OS kernel, although the context data structure is available to user-mode in the form of a `CONTEXT` structure. Similarly, when the thread is rescheduled onto a processor, this state must be restored so the processor can begin fetching and executing the thread's instructions again. We'll look at this process in more detail later. Note that contexts arise in a few other places too. For example, when an exception occurs, the OS takes a snapshot of the current context so that exception handling code can inspect the IP and other state when determining how to react. Contexts are also useful when writing debugging and diagnostics tools.

As the processor invokes various function call instructions, a region of memory called the **stack** is used to pass arguments from the caller to the callee (i.e., the function being called), to allocate local variables, to save register values, and to capture return addresses and values. Code on a thread can allocate and store arbitrary data on the stack too. Each thread, therefore, has its own region of stack memory in the process's virtual address space. In truth, each thread actually has two stacks: a user-mode and a kernel-mode stack. Which gets used depends on whether the thread is actively running code in user- or kernel-mode, respectively. Each thread has a well-defined lifetime. When a new process is created, Windows also creates a thread that begins executing that process's entry-point code. A process doesn't execute anything, its threads do. After the magic of a process's first thread being created—handled by the OS's process creation routine—any

code inside that process can go ahead and create additional threads. Various system services create threads without you being involved, such as the CLR's garbage collector. When a new thread is created, the OS is told what code to begin executing and away it goes: it handles the bookkeeping, setting the processor's IP, and the code is then subsequently free to create additional threads, and so on.

Eventually a thread will exit. This can happen in a variety of ways—all of which we'll examine soon—including simply returning from the entry-point used to begin the thread's life an unhandled exception, or directly calling one of the platform's thread termination APIs.

The Windows thread scheduler takes care of tracking all of the threads in the system and working with the processor(s) to schedule execution of them. Once a thread has been created, it is placed into a queue of runnable threads and the scheduler will eventually let it run, though perhaps not right away, depending on system load. Windows uses preemptive scheduling for threads, which allows it to forcibly stop a thread from running on a certain processor in order to run some other code when appropriate. Preemption causes a context switch, as explained previously. This happens when a higher priority thread becomes runnable or after a certain period of time (called a **quantum** or a **timeslice**) has elapsed. In either case, the switch only occurs if there aren't enough processors to accommodate both threads in question running simultaneously; the scheduler will always prefer to fully utilize the processors available.

Threads can **block** for a number of reasons: explicit I/O, a hard page fault (i.e., caused by reading or writing virtual memory that has been paged out to disk by the OS), or by using one of the many synchronization primitives detailed in Chapters 5, Windows Kernel Synchronization and 6, Data and Control Synchronization. While a thread blocks, it consumes no processor time or power, allowing other runnable threads to make forward progress in its stead. The act of blocking, as you might imagine, modifies the thread data structure so that the OS thread scheduler knows it has become ineligible for execution and then triggers a context switch. When the condition that unblocks the thread arises, it becomes eligible for execution again, which places it back into the queue of runnable threads, and the scheduler will later schedule it to run using its ordinary thread scheduling

algorithms. Sometimes awakened threads are given priority to run again, something called a **priority boost,** particularly if the thread has awakened in response to a GUI event such as a button click. This topic will come up again later.

There are five basic mechanisms in Windows that routinely cause nonlocal transfer of control to occur. That is to say, a processor's IP jumps somewhere very different from what the program code would suggest should happen. The first is a context switch, which we've already seen. The second is exception handling. An exception causes the OS to run various exception filters and handlers in the context of the current executing thread, and, if a handler is found, the IP ends up inside of it.

The next mechanism that causes nonlocal transfer of control is the hardware interrupt. An interrupt occurs when a significant hardware event of interest occurs, like some device I/O completing, a timer expiring, etc., and provides an interrupt dispatch routine the chance to respond. In fact, we've already seen an example of this: preemption based context switches are initiated from a timer based interrupt. While an interrupt borrows the currently executing thread's kernel-mode stack, this is usually not noticeable: the code that runs typically does a small amount of work very quickly and won't run user-mode code at all.

(For what it's worth, in the initial SMP versions of Windows NT, all interrupts ran on processor number 0 instead of on the processor executing the affected thread. This was obviously a scalability bottleneck and required large amounts of interprocessor communication and was remedied for Windows 2000. But I've been surprised by how many people still believe this is how interrupt handling on Windows works, which is why I mention it here.)

Software based interrupts are commonly used in kernel and system code too, bringing us to the fourth and fifth methods: deferred procedure calls (DPCs) and asynchronous procedure calls (APCs). A DPC is just some callback that the OS kernel queues to run later on. DPCs run at a higher Interrupt Request Level (IRQL) than hardware interrupts, which simply means they do not hold up the execution of other higher priority hardware based interrupts should one happen in the middle of the DPC running. If anything meaty has to occur during a hardware interrupt, it usually gets

done by the interrupt handler queuing a DPC to execute the hard work, which is guaranteed to run before the thread returns back to user-mode. In fact, this is how preemption based context switches occur. An APC is similar, but can execute user-mode callbacks and only run when the thread has no other useful work to do, indicated by the thread entering something called an **alertable wait.** When, specifically, the thread will perform an alertable wait is unknowable, and it may never occur. Therefore, APCs are normally used for less critical and less time sensitive work, or for cases in which performing an alertable wait is a necessary part of the programming model that users program against. Since APCs also can be queued programmatically from user-mode, we'll return to this topic in Chapter 5, Windows Kernel Synchronization. Both DPCs and APCs can be scheduled across processors to run asynchronously and always run in the context of whatever the thread is doing at the time they execute.

Threads have a plethora of other interesting aspects that we'll examine throughout this chapter and the rest of the book, such as priorities, thread local storage, and a lot of API surface area. Each thread belongs to a single process that has other interesting and relevant data shared among all of its threads—such as the handle table and a virtual memory page table—but the above definition gives us a good roadmap for exploring at a deeper level.

Before all of that, let's review what makes a managed CLR thread different from a native thread. It's a question that comes up time and time again.

What Is a CLR Thread?

A CLR thread is the same thing as a Windows thread—usually. Why, then, is it popular to refer to CLR threads as "managed threads," a very official term that makes them sound entirely different from Windows threads? The answer is somewhat complicated. At the simplest level, it effectively changes nothing for developers writing concurrent software that will run on the CLR. You can think of a thread running managed code as precisely the same thing as a thread running native code, as described above. They really aren't fundamentally different except for some esoteric and exotic situations that are more theoretical than practical.

First, the pragmatic difference: the CLR needs to track each thread that has ever run managed code in order for the CLR to do certain important jobs. The state associated with a Windows thread isn't sufficient. For example, the CLR needs to know about the object references that are live so that the garbage collector can determine which objects in the heap are still live. It does this in part by storing additional per-thread information such as how to find arguments and local variables on the stack. The CLR keeps other information on each managed thread, like event kernel objects that it uses for its own internal synchronization purposes, security, and execution context information, etc. All of these are simply implementation details.

Since the OS doesn't know anything about managed threads, the CLR has to convert OS threads to managed threads, which really just populates the thread's CLR-specific information. This happens in two places. When a new thread is created inside a managed program, it begins life as a managed thread (i.e., CLR-specific state is associated before it is even started). This is easy. If a thread already exists, however—that is it was created in native code and native-managed interoperability is being used—then the first time the thread runs managed code, the CLR will perform this conversion on-demand at the interoperability boundary.

Just to reiterate, all of this is transparent to you as a developer, so these points should make little difference. Knowing about them can come in useful, however, when understanding the CLR architecture and when debugging your programs.

Aside from that very down-to-earth explanation, the CLR has also decoupled itself from Windows threads from day one because there has always been the goal of allowing CLR hosts to override the default mapping of CLR threads directly to Windows threads. A CLR host, like SQL Server or ASP.NET, implements a set of interfaces, allowing it to override various policies, such as memory management, unhandled exception handling, reliability events of interest, and so on. (See Further Reading, Pratschner, for a more detailed overview of these capabilities.) One such overridable policy is the implementation of managed threads. When the CLR 2.0 was being developed, in fact, SQL Server 2005 experimented very seriously with mapping CLR threads to Windows fibers instead of threads, something they called **fiber-mode.** We'll explore in Chapter 9, Fibers, the

advantages fibers offer over threads, and how the CLR intended to support them. SQL Server has had a lot of experience in the past employing fiber based user-mode scheduling. We will also discuss We will also discuss a problem called **thread affinity**, which is related to all of this: a piece of work can take a dependency on the identity of the physical OS thread or can create a dependency between the thread and the work itself, which inhibits the platform's ability to decouple the CLR and Windows threads.

Just before shipping the CLR 2.0, the CLR and SQL Server teams decided to eliminate fiber-mode completely, so this whole explanation now has little practical significance other than as a possibly interesting historical account. But, of course, who knows what the future holds? User-mode scheduling offers some promising opportunities for building massively concurrent programs for massively parallel hardware, so the distinction between a CLR thread and a Windows thread may prove to be a useful one. That's really the only reason you might care about the distinction and why I labeled the concern "theoretical" at the outset.

Unless explicitly stated otherwise in the pages to follow, all of the discussions in this chapter pertain to behavior when run normally (i.e., no host) or inside a host that doesn't override the threading behavior. Trying to explain the myriad of possibilities simultaneously would be nearly impossible because the hosting APIs truly enable a large amount of the CLR's behavior to be extended and customized by a host.

Explicit Threading and Alternatives

We'll start our discussion about concurrency mechanisms at the bottom of the architectural stack with the Windows thread management facilities in Win32 and in the .NET Framework. This is called **explicit threading** in this book because you must be explicit about the creation and use of threads. This is a very low-level way to write concurrent software. Sometimes thinking at this low level is unavoidable, particularly for systems-level programming and, sometimes, also in application and library. Thinking about and managing threads is tricky and can quickly steal the focus from solving real algorithmic domain and business problems. You'll find that explicit threading quickly can become intrusive and pervasive in your program's architecture and implementation. Alternatives exist.

Thread pools abstract away the management of threads, amortizing the cost of creating and deleting them over the life of your process and optimizing the total number of threads to achieve superior all-around performance and scaling. Using a thread pool instead of explicit threading gets you away from thread management minutia and back to solving your business or domain problems. Most programmers can be very successful at concurrent programming without ever having to create a single thread by hand, thanks to carefully engineered Windows and CLR thread pool implementations.

Identifying patterns that emerge, abstracting them away, and hiding the use of threads and thread pools are also other useful techniques. It's common to layer systems so that most of the threading work is hidden inside of concrete components. A server program, for example, usually doesn't have any thread based code in callbacks; instead, there is a top-level processing loop that is responsible for moving work to run on threads. No matter what mechanisms you use, however, synchronization requirements are always pervasive unless alternative state management techniques (such as isolation) are employed.

Nevertheless, threads are a basic ingredient of life. Examining them in depth before looking at the abstractions that sit atop them will give you a better understanding of the core mechanisms in the OS, and from there, we can build up those (important and necessary) layers of abstraction without sacrificing knowledge of what underlies them. And perhaps you'll find yourself one day building such a layer of abstraction.

Last, a word of caution. Deciding precisely when it's a good idea to introduce additional threads is not as straightforward as you might imagine. Introducing too many can negatively impact your program's performance due to various fixed overheads and because the OS will spend increasingly more time trying to schedule them fairly as the ratio of threads to processors grows (we'll see details on this later). At the same time, introducing too few will lead to underutilized hardware and wasted opportunity. In some cases, the platform will help you create additional concurrency by using separate threads for some core system services (the CLR's ability to perform multithreaded garbage collections is one example), but more often than not, it's left to you to decide and manage.

The Life and Death of Threads

As with most things, threads have a beginning and an end. Let's take a look at what causes the creation of a new thread, what causes the termination of an existing thread, and what precisely goes on during these two events. We'll also look at the `DllMain` method, which is a way for native code to receive notifications of thread creation and termination events.

Thread Creation

During the creation of a new process, Windows will automatically create a new thread to run the program's entry point code. That's typically your main function in your programming language of choice (i.e., `(w)main in C++, Main` in C#, and so forth). Without at least one thread, the process wouldn't be able to do anything because processes themselves don't execute code—threads do. Once the process has been bootstrapped, additional threads may be created by code run within the process itself by the mechanisms we're about to review.

Programmatically Creating Threads

When creating a new thread, you must specify a few pieces of information, including the function at which the thread should begin running—the thread start routine—and the Windows kernel takes care of everything thereafter. When the creation request returns successfully, the new thread will have been initialized, and, so long as it wasn't created as suspended (specified by an optional flag), registered into a queue of threads to be run and later scheduled onto a processor. When the thread actually gets to run on a processor is subject to the thread scheduler and, therefore, system load and available resources. In fact, the new thread may have already begun (or finished) running by the time the request for creation returns.

Once the new thread runs, its thread start routine can call any other code in the process, and so forth, accessing any shared memory in the process's address space, using other process-wide resources, and perhaps even creating additional threads of its own. The thread start routine can return normally or throw an unhandled exception, both of which terminate the thread, or alternatively the thread can be terminated via some

other more explicit mechanism. We'll take a look at each of these termination mechanisms momentarily. But first, let's see the APIs used to create threads.

Win32 and the .NET Framework offer different but very similar ways to create a new thread. If you're writing native C programs, there is also a separate set of C APIs you must use to ensure the C Runtime Library (CRT) is initialized properly. We'll start by looking at Win32. Both the .NET Framework and CRT thread creation routines effectively build directly on top of Win32.

In Win32. Kernel32 offers the `CreateThread` API to create a new thread.

```
HANDLE WINAPI CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    SIZE_T dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter,
    DWORD dwCreationFlags,
    LPDWORD lpThreadId
);
```

`CreateThread` returns a `HANDLE` to the new thread kernel object, which can be passed to various other interesting Win32 APIs to later retrieve information about, interact with, or manipulate the newly created thread. (A `HANDLE`, by the way, is just an opaque pointer-sized value that indexes into a process-wide handle table. It's commonly used to refer to kernel objects. Managed code uses `IntPtrs` and `SafeHandles` to represent `HANDLEs`.) It must be closed when the creating thread no longer must interact with the new thread to avoid keeping the thread object's state alive indefinitely. The parameters to `CreateThread` are numerous:

- `LPSECURITY_ATTRIBUTES lpThreadAttributes`: a pointer to a `SECURITY_ATTRIBUTES` data structure. If `NULL`, the security attributes are inherited by the calling thread (which, if a thread along the way didn't specify overrides, in turn inherits them from the process). We will not discuss Windows object security in detail in this book; please refer to MSDN documentation and/or a book on Windows security for more details (see Further Reading, Brown).

- `SIZE_T dwStackSize`: the amount of user-mode stack, in bytes, to commit, in the virtual memory sense. If the `STACK_SIZE_PARAM_IS_A_RESERVATION` flag is present in the `dwCreationFlags` parameter, then this size represents the number of reserved bytes instead of committed bytes. `0` can be passed for `dwStackSize` to request that Windows use the process-wide default stack size. We discuss stack reservation, commit, and where this default comes from in the next chapter.
- `LPTHREAD_START_ROUTINE lpStartAddress`: a function pointer to the thread start routine. When Windows runs your thread, this is where it will begin execution. The type of function has the following signature:

  ```
  DWORD WINAPI ThreadProc(LPVOID lpParameter);
  ```

  The return value is captured and stored as the thread's exit code, which is then retrievable programmatically.
- `LPVOID lpParameter`: a pointer to memory you'd like to make accessible to the thread once it begins execution. This is opaque to Windows and is merely passed through as the value of your thread start routine's `lpParameter` argument. It's "opaque" because Windows will not attempt to dereference, validate it, or otherwise use it in any way. `NULL` is a valid argument value; without passing a pointer to some program data, the only valid way the thread will be able to find program data will be through accessing static or global variables.
- `DWORD dwCreationFlags`: a bit-flags value that enables you to indicate optional flags: that the stack size is for reservation rather than commit purposes (`STACK_SIZE_PARAM_IS_A_RESERVATION`), and/or that the thread should be left in a suspended state after `CreateThread` returns (`CREATE_SUSPENDED`). A thread that remains suspended must be resumed with a call to the Kernel32 `ResumeThread` API before it will be registered with the runnable thread queue and begin running. This can be useful if extra state must be prepared before the thread is able to begin executing. We look at thread suspension (`SuspendThread`) and resumption later.

- `LPDWORD lpThreadId`: An output pointer into which the `CreateThread` routine will store the newly created thread's process-wide unique identifier. As with the `HANDLE` returned, this can sometimes be used to subsequently interact with the thread. More often than not, it's just useful for diagnostics purposes. If you don't care about the thread's ID, as is fairly common, you can simply pass `NULL` (though on Windows 9X a valid non-`NULL` pointer must be supplied, otherwise `CreateThread` will attempt to dereference it and fail).

`CreateThread` can fail for a number of reasons, in which case the return value will be `NULL` and `GetLastError` may be used to retrieve details about the failure. Remember, each thread consumes a notable amount of system resources, including some amount of nonpageable memory, so if system resources are low, thread creation is very likely to fail: your code must be written to handle such cases gracefully, which may mean anything from choosing an alternative code-path or even terminating the program cleanly.

As a simple example of using `CreateThread`, consider Listing 3.1. In this code, the `main` routine is automatically called from the process's primary thread, which then invokes `CreateThread` to create a second program thread, supplying a function pointer to `MyThreadMain` as `lpStartAddress` and a pointer to the `"Hello, World"` string as `lpParameter`. Windows creates and enters the new thread into the scheduler's queue, at which point `CreateThread` returns and we make a call to the Win32 `WaitForSingleObject` API, passing the newly created thread's `HANDLE` as the argument. Though we don't look at the various Win32 wait functions Chapter 5, Windows Kernel Synchronization, this API call just causes the primary thread wait for the second thread to exit, allowing us to access and print the thread's exit code before exiting the program.

Listing 3.1: Creating a new OS thread with Win32's CreateThread function

```
WIN32 - C++  CREATETHREAD.CPP
#include <stdio.h>
#include <windows.h>

DWORD WINAPI MyThreadStart(LPVOID);
```

```
int main(int argc, wchar_t * argv[])
{
    HANDLE hThread;
    DWORD  dwThreadId;

    // Create the new thread.
    hThread = CreateThread(NULL,           // lpThreadAttributes
                           0,              // dwStackSize
                           &MyThreadStart, // lpStartAddress
                           "Hello, World", // lpParameter
                           0,              // dwCreationFlags
                           &dwThreadId);   // lpThreadId
    if (!hThread)
    {
        fprintf(stderr, "Thread creation failed: %d\r\n",
            GetLastError());
        return -1;
    }

    printf("%d: Created thread %x (ID %d)\r\n",
        GetCurrentThreadId(), hThread, dwThreadId);

    // Wait for it to exit and then print the exit code.
    WaitForSingleObject(hThread, INFINITE);

    DWORD dwExitCode;
    GetExitCodeThread(hThread, &dwExitCode);
    printf("%d: Thread exited: %d\r\n",
        GetCurrentThreadId(), dwExitCode);
    CloseHandle(hThread);

    return 0;
}

DWORD WINAPI MyThreadStart(LPVOID lpParameter)
{
    printf("%d: Running: %s\r\n",
        GetCurrentThreadId(), reinterpret_cast<char *>(lpParameter));
    return 0;
}
```

Notice that we use a few other APIs that haven't been described yet. First, `GetCurrentThreadId` retrieves the ID of the currently executing thread. This is the same ID that was returned from `CreateThread's lpThreadId` output parameter:

```
DWORD WINAPI GetCurrentThreadId();
```

And GetExitCodeThread retrieves the specified thread's exit code. We'll describe how exit codes are set when we discuss thread termination, but if you run this example, you'll see that when the thread terminates by its thread routine returning, the return value from the thread start is used as the exit code (which in this case means the value 0):

```
BOOL GetExitCodeThread(HANDLE hThread, LPDWORD lpExitCode);
```

GetExitCodeThread sets the memory location behind the lpExitCode output pointer to contain the thread's exit code. Both the ExitThread and TerminateThread APIs, used to explicitly terminate threads, allow a return code to be specified at the time of termination. It is generally accepted practice to use non-0 return values to indicate that a thread exit was caused due to an abnormal or unexpected condition, while 0 is usually used to indicate that termination was caused by ordinary business. If you try to access a thread's exit code before it has finished executing, a value of STILL_ACTIVE (0x103) is returned: clearly you should avoid using this error code for meaningful values because it could be interpreted wrongly.

This example isn't very interesting, but it shows some simple coordination between threads. There is little concurrency here, as our primary thread just waits while the new thread runs. We'll see more interesting uses as we progress through the book.

Another API is worth mentioning now. As we've seen, CreateThread returns a HANDLE to the newly created thread. In some cases you'll want to retrieve the current thread's HANDLE instead. To do that, you can use the GetCurrentThread function.

```
HANDLE WINAPI GetCurrentThread();
```

The returned value can be passed to any HANDLE based functions. But note that the value returned is actually special—something called a **pseudo-handle**—which is just a constant value (-2) that no real HANDLE would ever contain. GetCurrentProcess works similarly (returns -1 instead). Not having to manufacture a real handle is more efficient, but more importantly, pseudo-handles do not need to be closed. That means you needn't call CloseHandle on the returned value. But because the pseudo-handle is always interpreted as "the current thread" by Windows,

you can't just share the pseudo-handle value with other threads (it would be subsequently interpreted by that thread as referring to itself). To convert it into a real handle that is shareable, you can call `DuplicateHandle`, which returns a new shareable `HANDLE` that must be closed when you are through with it. Here is a sample snippet of code that converts a pseudo-handle into a real handle, printing out the two values.

```
#include <stdio.h>
#include <windows.h>

int main(int argc, wchar_t * argv[])
{
    HANDLE h1 = GetCurrentThread();
    printf("pseudo:\t%x\r\n", h1);

    HANDLE h2;
    DuplicateHandle(
        GetCurrentProcess(), h1, GetCurrentProcess(), &h2,
        0, FALSE, DUPLICATE_SAME_ACCESS);

    printf("real:\t%x\r\n", h2);

    CloseHandle(h2);
}
```

If all you've got is a thread's ID and you need to retrieve its `HANDLE`, you can use the `OpenThread` function. This also can be used if you need to provide a `HANDLE` that has been opened with only very specific access rights, that is, because you need to share it with another component.

```
HANDLE WINAPI OpenThread(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    DWORD dwThreadID
);
```

The `bInheritHandle` parameter specifies whether a `HANDLE` can be used by child processes (i.e., processes created by the one issuing the `OpenThread` call), and `dwThreadID` specifies the ID of the thread to which the `HANDLE` is to refer.

Finally, there is also a `CreateRemoteThread` function with nearly the same signature as `CreateThread`, with the difference that it accepts a process `HANDLE` as the first argument. As its name implies, this function

creates a new thread inside a process other than the caller's. This is a rather obscure capability, but can come in useful for tools like debuggers.

In C Programs. When you're programming with the C Runtime Library (CRT), you should use the `_beginthread` or `_beginthreadex` functions for thread creation in your C programs. These are defined in the header file `process.h`. These functions internally call `CreateThread`, but also perform some additional CRT initialization steps. If these steps are skipped, various CRT functions will begin failing in strange and unpredictable ways.

For example, the `strtok` function tokenizes a string. If you pass `NULL` as the string argument, it means "continue retrieving tokens from the previously tokenized string." In the original CRT—which was written long before multithreading was commonplace on Windows—the ability to remember "the previous string" was implemented by storing the tokens in global variables. This was fine with single-threaded programs, but clearly isn't for ones with multiple threads: imagine thread t1 tokenizes a string, then another thread t2 runs and tokenizes a separate string; when t1 resumes and tries to obtain additional tokens, it will be inadvertently sharing the token information from t2. Just about anything can happen, such as global state corruption, which can cause crashes or worse. Other functions do similar things: for example, `errno` stores and retrieves the previous error (similar to Win32's `GetLastError`) as global state.

With the introduction of the multithreaded CRT, `LIBCMT.LIB` (versus `LIBC.LIB`, usually accessed via the Visual C++ compiler switch `/MT`), all such functions now use thread local storage (TLS), which is just a collection of memory locations specific to each thread in the process. We'll review TLS in more detail later. To ensure the TLS state that these routines rely on has been initialized properly, the thread calling `strtok` or any of the other TLS based functions must have been created with either `_beginthread` or `_beginthreadex`. If the thread wasn't created in this way, these functions will try to access TLS slots that haven't been properly initialized and will behave unpredictably.

The `_beginthread` and `_beginthreadex` functions are quite similar in form to the `CreateThread` function reviewed earlier. Because of the similarities, we'll review them quickly.

```
uintptr_t _beginthread(
    void (__cdecl * start_address)(void *),
    unsigned stack_size,
    void * arglist
);
uintptr_t _beginthreadex(
    void * security,
    unsigned stack_size,
    unsigned (__stdcall * start_address)(void *),
    void * arglist,
    unsigned initflag,
    unsigned * thrdaddr
);
```

Each takes a function pointer, `start_address`, to the routine at which to begin execution. The `_beginthread` function differs from `_beginthreadex` and `CreateThread` in that the function's calling convention must be `__cdecl` instead of `__stdcall`, as you would expect for a C based program versus a Win32 based one, and the return type is `void` instead of a `DWORD` (i.e., it doesn't return a thread exit code). Each takes a `stack_size` argument whose value is used the same as in `CreateThread` (`0` means the process-wide default) and an `arglist` pointer that is subsequently accessible via the thread start's first and only argument.

The `_beginthreadex` function takes two additional arguments. The value `CREATE_SUSPENDED` can be passed for the `initflag` parameter, which, just as with the `CreateThread` API, ensures that the thread is created in a suspended state and must be manually resumed with `ResumeThread` before it runs. There are no special CRT functions for thread suspend and resume. The `thrdaddr` argument, if non-`NULL`, receives the resulting thread identifier as an output argument.

In both cases, the function returns a handle to the thread (of type `uintptr_t`, which can safely be cast to `HANDLE`) or `0` if there was an error during creation. Be extremely careful when using `_beginthread`, as the thread's handle is automatically closed when the thread start routine exits. If the thread runs quickly, the `uintptr_t` returned could represent an invalid handle by the time `_beginthread` even returns. This is in contrast to `_beginthreadex` and `CreateThread`, which require that the code creating the thread closes the returned handle if it's not needed and makes `_beginthread` nearly useless unless the creating thread has no need to subsequently interact with the newly created thread.

We will discuss more about exiting threads in a CRT safe way later, when we talk about thread termination and the `_endthread` and `_endthreadex` functions.

In the .NET Framework. In managed code you can use the `System.Threading.Thread` class's constructors and `Start` methods to create a new managed thread. The primary difference between this mechanism and Win32's `CreateThread` is just that the CLR has a chance to set up various bookkeeping data structures, as described previously, and, of course, the use of a CLR object to represent the thread in your programs instead of an opaque `HANDLE`.

(There also is a corresponding class `System.Diagnostics.ProcessThread`, which also offers access to various thread information and attributes in managed code. This type exposes additional capabilities that the managed `Thread` object doesn't. However, you cannot retrieve an instance of `ProcessThread` from a `Thread` instance, and vice versa, so, as its name implies, this is much more useful as a diagnostics tool rather than something you will use in production code. Hence, most of this chapter ignores `ProcessThread` and instead focuses on the actual `Thread` class itself.)

First the thread object must be constructed using one of `Thread`'s various constructors.

```
public delegate void ThreadStart();
public delegate void ParameterizedThreadStart(object obj);

public class Thread
{
    public Thread(ThreadStart start);
    public Thread(ThreadStart start, int maxStackSize);
    public Thread(ParameterizedThreadStart start);
    public Thread(ParameterizedThreadStart start, int maxStackSize);
    ...
}
```

Assuming an unhosted CLR, each `Thread` object is just a thin object oriented veneer over an OS thread kernel object. Note that when you instantiate a new `Thread` object, the CLR hasn't actually created the underlying OS thread kernel object, user- or kernel-mode stack, and so on, just yet. This constructor just allocates some tiny internal data structures necessary to

store your constructor arguments so that they can be used should you decide to start the thread later. If you never get around to starting the thread, there will never be any OS resources backing it.

After creating the object, you must call the `Start` method on it to actually create the OS thread object and schedule it for execution. As you might imagine, the unhosted CLR uses the `CreateThread` API internally to do that.

```
public class Thread
{
    public void Start();
    public void Start(object parameter);
    ...
}
```

A thread created with the `ParameterizedThreadStart` based constructor allows a caller to pass an object reference argument to the `Start` method (as `parameter`), which is then accessible from the new thread's start routine as `obj`. This is similar to the `CreateThread` API, seen above, and provides a simple way of communicating state between the creator and createe. A similar effect can be achieved by passing a thread start delegate that refers to an instance method on some object, in which case that object's instance state will be accessible from the thread start via `this`. If a thread created with a `ParameterizedThreadStart` delegate is subsequently started with the parameterless `Start` overload, the value of the thread start's `obj` argument will be `null`.

There are a couple of constructor overloads that accept a `maxStackSize` parameter. This specifies the size of the thread's reserved and committed stack size (because in managed code both are the same). We return to more details about stacks in the next chapter, including why you might want to change the default.

It's also worth pointing out that many of `Thread's` methods (in addition to most synchronization related methods), including `Start`, are protected by a Code Access Security `HostProtection` link demand for `Synchronization` and `ExternalThreading` permissions. This ensures that, while untrusted code can create a new CLR thread object (because its constructors are not protected), most code hosted inside a program like SQL Server cannot start or control a thread's execution. Deep examinations of security and hosting are both outside of the scope of this book. Please refer to Further Reading, Brown and Pratschner, for excellent books on the topics.

Listing 3.2 illustrates an example comparable to the Win32 code in Listing 3.1 earlier. Just as we had used the `WaitForSingleObject` Win32 API to wait for the thread to exit, we use `Thread`'s `Join` method. We'll review `Join` in more detail later, though it doesn't get much more complicated than what is shown here. You'll notice that the CLR doesn't expose any sort of thread exit code capability.

Listing 3.2: Creating a new OS thread with the .NET Framework's Thread class

```
using System;
using System.Threading;

class Program
{
    public static void Main()
    {
        Thread newThread = new Thread(
            new ParameterizedThreadStart(MyThreadStart));

        Console.WriteLine("{0}: Created thread (ID {1})",
            Thread.CurrentThread.ManagedThreadId,
            newThread.ManagedThreadId);

        newThread.Start("Hello world"); // Begin execution.

        newThread.Join(); // Wait for the thread to finish.

        Console.WriteLine("{0}: Thread exited",
            Thread.CurrentThread.ManagedThreadId);
    }

    private static void MyThreadStart(object obj)
    {
        Console.WriteLine("{0}: Running: {1}",
            Thread.CurrentThread.ManagedThreadId, obj);
    }
}
```

You can write this code more succinctly using C# 2.0's anonymous delegate syntax.

```
Thread newThread = new Thread(delegate(object obj)
{
    Console.WriteLine("{0}: Running {1}",
        Thread.CurrentThread.ManagedThreadId, obj);
});
newThread.Start("Hello world (with anon delegates)");
newThread.Join();
```

Using lambda syntax in C# 3.0 makes writing similar code even slightly more compact.

```
Thread newThread = new Thread(obj =>
    Console.WriteLine("{0}: Running {1}",
        Thread.CurrentThread.ManagedThreadId, obj)
);
newThread.Start("Hello, world (with lambdas)");
newThread.Join();
```

We make use of the `CurrentThread` static property on the `Thread` class, which retrieves a reference to the currently executing thread, much like `GetCurrentThread` in Win32. We then use the instance property `ManagedThreadId` to retrieve the unique identifier assigned by the CLR to this thread. This identifier is completely different than the one assigned by the OS. If you were to P/Invoke to `GetCurrentThreadId`, you'll likely see a different value.

```
public class Thread
{
    public static Thread CurrentThread { get; };
    public int ManagedThreadId { get; }
    ...
}
```

Again, this code snippet isn't very illuminating. We'll see more complex examples. But as you can see, the idea of a thread as seen by Win32 and managed code programmers is basically the same. That's good as it means most of what we've discussed and are about to discuss pertains to native and managed code alike.

Thread Termination

A thread goes through a complex lifetime, from runnable to running to possibly waiting, possibly being suspended, and so forth, but it will eventually terminate. Termination might occur as a result of any one of a number of particular events.

1. The thread start routine can return normally.
2. An unhandled exception can escape the thread start routine, "crashing" that thread.

3. A call can be made to one of the Win32 functions `ExitThread` or `TerminateThread`, either by the thread itself (synchronous) or by another thread (asynchronous). There is no direct equivalent to these functions in the .NET Framework, and P/Invoking to them will lead to much trouble.
4. A managed thread abort can be triggered by a call to the .NET Framework method `Thread.Abort`, either by the thread itself (synchronous) or by another thread (asynchronous). There is no equivalent in Win32. This approach in fact looks a lot like `ExitThread`, though you can argue that it is a "cleaner" way to shut down threads. We'll see why shortly. That said, aborting threads is still (usually) a bad practice.

   A managed thread may also be subject to a thread abort induced by the CLR infrastructure or a CLR host. Aborts also occur on all threads running code in an AppDomain when it is being unloaded. This is different from the previous item because it's initiated by the infrastructure, which knows how to do this safely.
5. The process may exit.

Of course, the machine could get unplugged, in which case threads terminate, but since there's not much our software can do in response to such an event, we'll set this aside.

After a thread terminates, assuming the process remains alive, its data structures continue to live on until all of the `HANDLE`s referring to the thread object have been closed. The CLR thread object, for example, uses a finalizer to close this handle, which means that the OS data structures will continue to live until the GC collects the `Thread` object and then runs its finalizer, even though the thread is no longer actively running any code.

Several of the techniques mentioned are brute force methods for thread termination and can cause trouble (namely 3 and 4). Higher-level coordination must be used to cooperatively shut down threads or else program and user data can become corrupt.

Note that the termination of a thread may cause termination of its owning process. In native code, the process will exit automatically when the last thread in a process exits. In managed code, threads can be marked as a

background thread (with the `IsBackground` property), which ensures that a particular thread won't keep the process alive. A managed process will automatically exit once its last nonbackground thread exits. As with thread termination, there are other brute force (and problematic) ways to shut down a process, such as with a call to `TerminateProcess`.

Method 1: Returning from the Thread Start Routine

Any thread start routine that returns will cause the thread to exit. This is by far the cleanest way to trigger thread exit. The top of each thread's callstack is actually a Windows internal function that calls the thread start routine and, once it returns, calls the `ExitThread` API. This is true for both native and managed threads and is imposed by Windows. This is the cleanest shutdown method because the thread start routine is able to run to completion without being interrupted part way through some application specific code.

While not exposed through the managed thread object, each OS thread remembers an exit code, much like a process does. The `CreateThread` start routine function pointer type returns a `DWORD` value and the callback for `_beginthreadex` returns an `unsigned` value. Managed threading doesn't support exit codes and is evidenced by the fact that `ThreadStart` and `ParameterizedThreadStart` are typed as returning `void`. Programs can use exit codes to communicate the reason for thread termination. Windows stores the return value as part of the thread object so that it can be later retrieved with `GetExitCodeThread`, as we saw just a bit earlier. Most alternative forms of thread termination also supply a way to set this code.

Method 2: Unhandled Exceptions

If an exception reaches the top of a thread's stack without having been caught, the thread will be terminated. The default Windows and CLR behavior is to terminate the process when such an unhandled exception occurs (for most cases), though a custom exception filter can be installed to change this behavior. Of course, many exceptions are handled before getting this far, in which case there is no impact on the life of the thread. Additionally, some programs install custom top-level handlers that catch all exceptions, perform error logging, and attempt some level of data recovery before letting the process crash.

Process termination works by installing at the base of every Windows thread's stack an SEH exception filter. This filter decides what to do with unhandled exceptions. The details here differ slightly between native and managed code, because managed code wraps everything in its own exception filter and handler too.

The default filter in native code will display a dialog when the exception has been deemed to go unhandled during the first pass. It asks the user to choose whether to debug or terminate the process (the latter of which just calls `ExitProcess`). All of this occurs in the first pass of exception handling, so by default, no stacks have been unwound at this point. Anybody who has written code on Windows knows what this dialog looks like. Though it tends to change from release to release, it offers the same basic functionality: debug or terminate the process and, now in Windows Vista, check for solutions online.

The CLR installs its own top-level unhandled exception filter, which performs debugger notification, integrates with Dr. Watson to generate proper crash dumps, raises an event in the AppDomain so that custom managed code can execute shutdown logic, prints out more friendly failure information (including a stack trace) to the console, and unwinds the crashing thread's stack, letting managed finally blocks run. One interesting difference is that finally blocks are run when a managed thread crashes, while in native they are not (by default). This custom exception logic is run regardless of whether it was a managed or native thread in the process that caused the unhandled exception because the CLR overrides the process-wide unhandled exception behavior.

There are two special exceptions to the rule that any unhandled exception causes the process to exit: an unhandled `ThreadAbortException` or `AppDomainUnloadedException` will cause the thread on which it was thrown to exit, but will not actually trigger a process exit (unless it's the last nonbackground thread in the process). Instead, the exception will be swallowed and the process will continue to execute as normal. This is done because these exceptions are regularly used by the runtime and CLR hosts to carefully unload an AppDomain while still keeping the rest of the process alive.

Overriding the Default Unhandled Exception Behavior. There are a few ways in which you may override the default unhandled exception behavior. Doing so is seldom necessary. The first way allows you to turn off the default dialog in Win32 programs by passing the `SEM_NOGPFAULTERRORBOX` flag to the `SetErrorMode` function. This is usually a bad idea if you want to be able to debug your programs, but it can be useful for noninteractive programs:

```
UINT SetErrorMode(UINT uMode);
```

A change was made in the CLR 2.0 to make unhandled exceptions on the finalizer thread, thread pool threads, and user created threads exit the process. In the CLR 1.X, such exceptions were silently swallowed by the runtime. An unhandled exception is more often than not an indication that something wrong has happened and, therefore, the old policy tended to lead to many subtle and hard to diagnose errors. Swallowing the exception merely masked a problem that was sure to crop up later in the program's execution. At the same time, this change in policy can cause compatibility problems for those migrating from 1.X to 2.0 and above. A configuration setting enables you to recover the 1.X behavior.

```
<system>
    <runtime>
        <legacyUnhandledExceptionPolicy enabled="1" />
    </runtime>
</system>
```

Using this configuration setting is highly discouraged for anything other than as an (one hopes temporary) application compatibility crutch. It can create debugging nightmares. CLR hosts can also override (some of) this unhandled exception behavior, so what has been described in this section strictly applies only to unhosted managed programs. Please refer to Pratschner (see Further Reading) for details on how this is done.

Some of you might be wondering how the CLR is able to hook itself into the whole Windows unhandled exception process so easily. Any user-mode code can install a custom top-level SEH exception filter that will be called instead of the default OS filter when an unhandled exception occurs. `SetUnhandledExceptionFilter` installs such a filter.

```
LPTOP_LEVEL_EXCEPTION_FILTER SetUnhandledExceptionFilter(
    LPTOP_LEVEL_EXCEPTION_FILTER lpTopLevelExceptionFilter
);
```

`LPTOP_LEVEL_EXCEPTION_FILTER` is just a function pointer to an ordinary SEH exception filter.

```
LONG WINAPI UnhandledExceptionFilter(
    struct _EXCEPTION_POINTERS * ExceptionInfo
);
```

The `_EXCEPTION_POINTERS` data structure is passed by the OS—and is the same value you'd see if you were to call `GetExceptionInformation` by hand during exception handling—which provides you with an `EXCEPTION_RECORD` and `CONTEXT`. The record provides exception details and the `CONTEXT` is a collection of the processor's volatile state (i.e., registers) at the time the exception occurred. We review contexts later in this chapter. As with any filter, this routine can inspect the exception information and decide what to do. At the end, it returns `EXCEPTION_CONTINUE_SEARCH` or `EXCEPTION_EXECUTE_HANDLER` to instruct SEH whether to execute a handler or not.

(The details of the CLR and Windows SEH exception systems are fascinating, but are fairly orthogonal to the topic of concurrency. Therefore we won't review them here, and instead readers are encouraged to read Pietrek (see Further Reading) for a great overview.)

If you return `EXCEPTION_CONTINUE_SEARCH` from this top-level filter, the exception goes completely unhandled and the OS will perform the default unhandled exception behavior. That entails showing the dialog (assuming it has not been disabled via `SetErrorMode`) and calling `ExitProcess` without unwinding the crashing thread's stack. All of this happens during the first pass. If you return `EXCEPTION_EXECUTE_HANDLER`, however, a special OS-controlled handler is run. This SEH handler sits at the base of all threads and will call `ExitProcess` without displaying the standard error dialog. And because we have told SEH to execute a handler, the thread's stack is unwound normally, and, hence, the call to `ExitProcess` occurs during the second pass after finallys blocks have been run.

Method 3: `ExitThread` and `TerminateThread` (Native Code Only)

If you're writing native code, you can explicitly terminate a thread (although it is generally very dangerous to do so and should be done only after this is understood). This can be done for the current thread (synchronous) or another thread running in the system (asynchronous). There are two Win32 APIs to initiate explicit thread termination

```
VOID WINAPI ExitThread(DWORD dwExitCode);
BOOL WINAPI TerminateThread(HANDLE hThread, DWORD dwExitCode);
```

Calling `ExitThread` will immediately cause the thread to exit, without unwinding its stack, meaning that finally blocks and destructors will not execute. It changes the thread's exit code from `STILL_ACTIVE` to the value supplied as the `dwExitCode` argument. The thread's user- and kernel-mode stack memory is de-allocated, pending asynchronous I/O is canceled (see Chapter 15, Input and Output), thread detach notifications are delivered to all DLLs in the process that have defined a `DllMain` entry point, and the kernel thread object becomes signaled (see Chapter 5, Windows Kernel Synchronization). The thread may continue to use resources because the kernel object and its associated memory remains allocated until all outstanding `HANDLEs` to it have been closed.

If you created threads with the CRT's `_beginthread` or `_beginthreadex` function, then you must use the `_endthread` or `_endthreadex` function instead of `ExitThread`.

```
void _endthread();
void _endthreadex(unsigned retval);
```

Internally, these both call `ExitThread`, but they additionally provide a chance for the CRT to de-allocate any per-thread resources that were allocated at runtime. Terminating threads created with the`_beginthread` routines using `ExitThread` or `TerminateThread` will cause these resources to be leaked. The leaks are so small that they could go unnoticed for some time, but will certainly cause progressively severe problems for long running programs. The only difference between`_endthread` and`_endthreadex` is that`_endthreadex` accepts a thread exit code as the `retval` argument, while`_endthread` simply uses `0` as the exit code.

The first method of terminating a thread described earlier—returning from the thread start routine—internally calls `ExitThread` (via_endthreadex) at the base of the stack, passing the routine's return value as the `dwExitCode` argument. Exiting a thread can only occur synchronously on a thread; in other words, some other thread can't exit a separate thread "from the outside." This means that `ExitThread` is safer, though it can lead to issues like lock orphaning and memory leaks because the thread's stack is not before exiting.

The `TerminateThread` function, on the other hand, is extremely dangerous and should almost never be used. The only possible situations in which you should consider using it are those where you are entirely in control of what code the target thread is executing. Terminating a thread this way does not free the user-mode stack and does not deliver `DllMain` notifications. Calling it synchronously on a thread is very similar to `ExitThread`, with these two differences aside. But calling it asynchronously can cause problems. The target thread could be holding on to locks that, after termination, will remain in the acquired state. For example, the thread might be in the process of allocating memory, which often requires a lock. Once terminated, no other thread would be able to subsequently allocate memory, leading to deadlocks. Similarly, the target could be modifying critical system state that could become corrupt when interrupted part way through. If you are considering using `TerminateThread`, you should follow it soon with a call to terminate the process as well.

In all cases, using higher-level synchronization mechanisms to shut down threads is always preferred. This typically requires some combination of state and cooperation among threads to periodically check for shutdown requests and voluntarily return back to the thread start routine when a request has been made. `ExitThread` and `TerminateThread` often seem like "short-cuts" to achieve this, while avoiding the need to perform this kind of higher-level orchestration; there's certainly less tricky cooperation code to write because many important issues are hidden. Generally speaking, this should be considered a sloppy coding practice, viewed with great suspicion, and regarded as likely to lead to many bugs.

Managed code should never explicitly terminate managed threads using these mechanisms. Instead, synchronization should be used to orchestrate

exit or, in some specific scenarios, thread aborts can be used instead (see below). P/Invoking to `ExitThread` or `TerminateThread` will lead to unpredictable and unwanted behavior for much the same reason that calling `ExitThread` instead of `_endthreadex` can cause problems: that is, the CLR has state to clean up and bookkeeping to perform whenever a thread terminates.

Method 4: Thread Aborts (Managed Code Only)

Managed threads can be aborted. When a thread is aborted, the runtime tears it down by introducing an exception at the thread's current instruction pointer, versus stopping the thread in its tracks a la the Win32 `ExitThread` function. Using an exception such as this allows finally blocks to execute as the thread unwinds, ensuring that important resources are cleaned up appropriately. Moreover, the runtime is aware of certain regions of code that are performing uninterruptible operations, such as manipulating important system-wide state, and will delay introducing the aborting exception until a safe point has been reached.

Thread aborts can be introduced synchronously and asynchronously, just like `TerminateThread`. When an asynchronous abort is triggered, an instance of `System.Threading.ThreadAbortException` is constructed and thrown in the aborted thread, just as if the thread itself threw the exception. Synchronous aborts, on the other hand, are fairly straightforward: the thread itself just throws the exception. As described earlier, unhandled thread abort exceptions only terminate the thread on which the exception was raised, and do not cause the process to exit (unless that was the last nonbackground thread).

To initiate a thread abort, the `Thread` class offers an explicit `Abort` API.

```
public void Abort();
public void Abort(object stateInfo);
```

When aborting another thread asynchronously, the call to `Abort` blocks until the thread abort has been processed. Note that when the call unblocks, it does not mean that the thread has been aborted yet. In fact, the thread may suppress the abort, so there is no guarantee that the thread will exit. You should use other synchronization techniques (such as the `Join` API) if you must wait for the thread to complete. If the overload, which accepts the

`stateInfo` parameter, is used, the object is accessible via the `ThreadAbortException`'s `ExceptionState` property, allowing one to communicate the reason for the thread abort.

`ThreadAbortExceptions` thrown during a thread abort are special. They cannot be swallowed by catch blocks on the thread's callstack. The stack will be unwound as usual, but if a catch block tries to swallow the exception, the CLR reraises it once the catch block has finished running. An abort can be reset mid-flight with the `Thread.ResetAbort` API, which will allow exceptions to be caught and the thread to remain alive.

```
public static void ResetAbort();
```

The following code snippet illustrates this behavior.

```
try
{
    try
    {
        Thread.CurrentThread.Abort();
    }
    catch (ThreadAbortException)
    {
        // Try to swallow it.
    }   // CLR automatically reraises the exception here.
}
catch (ThreadAbortException)
{
    Thread.ResetAbort();
    // Try to swallow it again.
}   // The in-flight abort was reset, so it is not reraised again.
```

A single callstack may be executing code in multiple AppDomains at once. Should a `ThreadAbortException` cross an AppDomain boundary on a callstack, say from AppDomain B to A, it will be morphed into an `AppDomainUnloadedException`. Unlike thread abort exceptions, this exception type can be caught and swallowed by code running in A.

Delay-Abort Regions. As mentioned earlier, the runtime only initiates an asynchronous thread abort when the target thread is not actively running critical code: these are called **delay-abort regions.** Each of the following is considered to be a delay-abort region by the CLR: invocation of a catch or

finally block, code within a constrained execution region (CER), running native code on a managed thread, or invocation of a class or module constructor. When a thread is in such a region and is asynchronously aborted, the thread is simply marked with a flag (reflected in its state bitmask by `ThreadState.AbortRequested`), and the thread subsequently initiates the abort as soon as it exits the region, that is, when it reaches a safe point (taking into consideration that such regions may be nested). The determination of whether a thread is in a delay-abort region is made by the CLR suspending the target thread, inspecting its current instruction pointer, and so on.

Thread Abort Dangers. There are two situations in which thread aborts are always safe.

- The main purpose of thread aborts is to tear down threads during CLR AppDomain unloads. When an unload occurs—either because a host has initiated one or because the program has called the `AppDomain.Unload` function—any thread that has a callstack in an AppDomain is asynchronously aborted. As the abort exceptions reach the boundary of the AppDomain, the thread abort is reset and the exception turns into an `AppDomainUnloadedException`, which, as we've noted, can then be caught and handled. This is safe because nearly all .NET Framework code assumes that an asynchronous thread abort means the AppDomain is being unloaded and takes extra precautions to avoid leaking process-wide state.
- Synchronous thread aborts are safe, provided that callers expect an exception to be thrown from the method. Because the thread being aborted controls precisely when aborts happen, it's the responsibility of that code to ensure they happen when program state is consistent. A synchronous abort is effectively the same as throwing any kind of exception, with the notable difference that it cannot be caught and swallowed. It's possible that some code will check the type of the exception in-flight and avoid cleaning up state so that AppDomain unloads are not held up, but these cases should be rare.

All other uses of thread aborts are questionable at best. While a great deal of the .NET Framework goes to great lengths to ensure resources are not leaked and deadlocks do not occur (see Further Reading, Duffy, Atomicity and Asynchronous Exception Failures), the majority of the libraries are not written this way. Note that hosts can also initiate a so-called rude thread abort, which does not run finally blocks and will interrupt the execution of catch and finally clauses. This capability is used only by some hosts and not the unhosted CLR itself and, therefore, is inaccessible to managed code. A detailed discussion of this is outside the scope of this book.

While thread aborts are theoretically safer than other thread termination mechanisms, they can still occur at inopportune times, leading to instability and corruption if used without care. While the runtime knows about critical system state modifications, it knows nothing about application state and, therefore, aborts are not problem free. In fact, you should rarely (if ever) use one. But the runtime and its hosts are able to make use of them with great care, usually because possible state corruption can be contained appropriately.

As a simple illustration of what can go wrong when aborts occur at unexpected and inopportune places, let's look at an example that leads to a resource leak.

```
void UseSomeBigResource()
{
    IntPtr hBigResource = /* S0 */ Allocate();
    try
    {
        // Do something...
    }
    finally
    {
        Free(hBigResource);
    }
}
```

In this example, a thread abort could be triggered after the call to `Allocate` but before the assignment to the `hBigResource` local variable, at S0. An asynchronous thread abort here will lead to memory leakage (because the memory is not GC managed). Even if we were assigning the

result of `Allocate` to a member variable on a type that had a finalizer, to catch the case where the try/finally didn't execute the resource would leak because we never executed the assignment. If instead of allocating memory we were acquiring a mutually exclusive lock, for example, then an abort could lead to deadlock for threads that subsequently tried to acquire the orphaned lock. There are certainly ways to ensure reliable acquisition and release of resources (see Further Reading, Toub; Grunkemeyer), including using delay-abort regions with great care, but given that many of them are new to the CLR 2.0, most code that has been written remains vulnerable to such issues.

Method 5: Process Exit

The final method of terminating a thread is to exit the process without shutting down all of its threads. When it happens, it usually occurs in one of the following ways.

- Win32 offers `ExitProcess` and `TerminateProcess` APIs, which mirror the `ExitThread` and `TerminateThread` APIs reviewed earlier. When `ExitProcess` is called, `ExitThread` is called on all threads in the process, ensuring that DLL thread and process detach notifications are sent to DLLs loaded in the process. Threads are not unwound, so any destructors or finally blocks that are live on callstacks on these threads are not run. `TerminateProcess`, on the other hand, is effectively like calling `TerminateThread` on each thread and also skips the step of sending process detach notifications to loaded DLLs. Because these notifications are skipped, DLLs are not given a chance to free or restore machine-wide state.
- C programs can call either the `exit/_exit or abort` CRT library functions, which are similar to `ExitProcess` and `TerminateProcess`, respectively. Each contains additional logic, however. For example, exit invokes any routines registered with the CRT `atexit/_onexit` functions, and `abort` displays a dialog box indicating that the process has terminated abnormally.
- Managed code may call `Environment.Exit`, which triggers a clean shutdown of all threads in the process. The CLR will suspend all

threads, and then it will finalize any finalizable objects in the process. After this, it exits threads without running finally blocks. The CLR will actually create a so-called "shutdown watchdog thread" that monitors the shutdown process to ensure it doesn't hang. As we'll see in Chapter 6, Data and Control Synchronization, there are circumstances in which managed threads may hang during shutdown due to locks. If, after 2 seconds, the shutdown has not finished, the watchdog thread will take over and rudely shut down the process.

- Any managed code may also call `Environment.FailFast`. This is similar to calling `Exit`, except that it is meant for abnormal and unexpected situations where no managed code must run during the shutdown. This means that finalizers are not run, and AppDomain events are not called, and also an entry is made in the Windows Event Log to indicate failure.

The behavior explained above during shutdown in managed code always occurs. In fact, threads need to be terminated prematurely more frequently than you might think. That's because a managed process exits when all nonbackground threads exit, and it is actually quite common to have many background threads (e.g., in the CLR's thread pool).

Shutting down a process without cleanly exiting the application can lead to problems, particularly if you're using `TerminateThread` or `FailFast`. These APIs are best used to respond to critical situations in which continuing execution poses more risk to the stability of the system and integrity of data than shutting down abruptly and possibly missing some important application-specific cleanup activities. For example, if a thread is in the middle of writing data to disk, it will be stopped midway, possibly corrupting data. Even if a thread has finished writing, data may not be flushed until a certain point in the future, and shutting down skips finally blocks, etc., which may result in buffers not being flushed. There are many things that can go wrong, and they depend on subtle timings and interactions, so a clean shutdown should always be preferred over all of the methods described in this section.

DllMain

We've referenced DLL_THREAD_ATTACH and DLL_THREAD_DETACH notifications at various points above. Now let's see how you register to receive such notifications. Each native DLL may specify a DllMain entry point function in which code to respond to various interesting process events may be placed. The signature of the DllMain function is:

```
BOOL WINAPI DllMain(
    HINSTANCE hInstDLL,
    DWORD fdwReason,
    LPVOID lpReserved
);
```

Defining a DLL entry point is optional. The OS will call the entry point for all DLLs that have defined entry points, as they are loaded into the process, when one of four events occurs. The event is indicated by the value of the fdwReason argument supplied by the OS:

- DLL_PROCESS_ATTACH: This is called when a DLL is first loaded into a process. For libraries statically linked into an EXE, this will occur at process load time, while for dynamically loaded DLLs, it will occur when LoadLibrary is invoked. This event may be used to perform initialization of data structures that the DLL will need during execution. If the lpReserved argument is NULL, it indicates the DLL has been loaded dynamically, while non-NULL indicates it has been loaded statically.
- DLL_PROCESS_DETACH: This is called when the DLL is unloaded from the process, either because the process is exiting or, for dynamically loaded libraries, when the FreeLibrary function has been called. The process detach notification handling code is ordinarily symmetric with respect to the process attach; in other words, it typically is meant to free any data structures or resources that were allocated during the initial DLL load. If lpReserved is NULL, it indicates the DLL is being dynamically unloaded with FreeLibrary, while non-NULL indicates the process is terminating.
- DLL_THREAD_ATTACH: Each time the process creates a new thread, this notification will be made. Any thread specific data structures may

then be allocated. Note that when the initial process attach notification is sent there is not an accompanying thread attach notification, neither will there be notifications for existing threads in the process when a DLL is dynamically loaded after threads were created.

- `DLL_THREAD_DETACH`: When a thread exits the system, the OS invokes the `DllMain` for all loaded DLLs and sends a detach notification from the thread that is exiting. This is the DLL's opportunity to free any data structures or resources allocated inside of the thread attach routine.

There is no equivalent to `DllMain` in managed code. Instead, there is an `AppDomain.ProcessExit` event that the CLR calls during process shutdown. If you are writing a C++/CLI assembly, or interoperating with an existing native DLL, however, you will be delivered `DllMain` notifications as normal.

The `DllMain` function is one of few places that program code is invoked while the OS holds the loader lock. The loader lock is a critical region used by the OS to protect access to loadtime state and automatically acquires it in several places: when a process is shutting down, when a DLL is being loaded, when a DLL is being unloaded, and inside various loader related APIs. It's a lock just like any other, and so it is subject to deadlock. This makes it particularly dangerous to write code in the `DllMain` routine. You must not trigger another DLL load or unload, and certainly should never synchronize with another thread that might hold a lock and then need to acquire the loader lock. It's easy to write deadlock prone code in your `DllMain` without even knowing it. Techniques like lock leveling (see Chapter 11, Concurrency Hazards, for details) can avoid deadlock, but generally speaking, it's better to avoid all synchronization in your `DllMain` altogether. See Further Reading, MSDN, Best Practices for Creating DLLs, for some additional best practices for DLL entry point code.

Prior to C++/CLI in Visual Studio 2005, it was impossible to create a C++ mixed mode native/managed DLL that contained a `DllMain` without it being deadlock prone. The reasons are numerous (see Further Reading, Brumme), but the basic problem is that it's impossible to run managed code without acquiring locks and possibly synchronizing with other threads (due to GC), which effectively guarantees that deadlocks are always

possible. If you're still writing code in 1.0 or 1.1, workarounds are possible (see Further Reading, Currie). As of Visual C++ 2005, however, managed code is not called automatically inside of `DllMain` and thus it's possible to write safe deadlock free entry points, provided you do not call into managed code explicitly. See Further Reading, MSDN, Visual C++: Initialization of Mixed Assemblies for details.

There is a hidden cost to defining `DllMain` routines. Every time a thread is created or destroyed, the OS must enumerate all loaded DLLs and invoke their `DllMain` functions with an attach or detach notification, respectively. Win32 offers an API to suppress notifications for a particular DLL, which can avoid this overhead when the calls are unnecessary.

```
BOOL WINAPI DisableThreadLibraryCalls(HMODULE hModule);
```

Using this API to suppress DLL notifications can provide sizeable performance improvements, particularly for programs that load many DLLs and/or create and destroy threads with regularity. But use it with caution. If a third party DLL has defined a `DllMain` function, it's probably for a reason; suppressing calls into it is apt to cause unpredictable behavior.

Thread Local Storage

Programs can store information inside thread local storage (TLS), which permits each thread to maintain some private data that isn't shared among other threads but that is globally accessible to any code running on that thread. This enables one part of the program to place data into a known location so another part can subsequently access and/or modify it. Static variables in C++ and C#, for example, refer to memory that is shared among all threads in the process. Accessing this shared state must be done with care, as we've established in previous chapters. It's often more attractive to isolate data so that synchronization isn't necessary or because the specific details of your problem allow or require information to be thread specific.

That's where TLS comes into the picture. With TLS, each thread in the system is allocated a separate region of memory to represent the same logical variable. Native and managed code both offer TLS support, with very similar programming interfaces, but the details of each are rather different. We'll review both, in that order.

Win32 TLS

There are two TLS modes for native code: dynamic and static. **Dynamic TLS** can be used in any situation, including static and dynamic link libraries, and executables. **Static TLS** is supported by the C++ compiler and may only be used for statically linked code but has the advantage of greater efficiency when accessing TLS information. Code can freely intermix the two in the same program and process without problems.

Dynamic TLS. In order to use native TLS to store and retrieve information, you must first allocate a TLS slot for each separate piece of data. Allocating a slot simply retrieves a new index and removes it from the list of available indices in the process. This slot index is a numeric `DWORD` value that is used to set or retrieve a `LPVOID` value stored in a per thread, per slot location managed by the OS. In fact, this value is just an index into an array of `LPVOID` entries that each thread has allocated at thread instantiation time.

Reserving a new index is done with the `TlsAlloc` API.

```
DWORD WINAPI TlsAlloc();
```

All TLS slots are 0 initialized when a thread is created, so all slots will initially contain the value `NULL`. The index itself should be treated as an opaque value, much like a `HANDLE`. Each thread in the process uses this same index value to access the same TLS slot, meaning that the value is typically shared in some static or global variable that all threads can access.

If `TlsAlloc` returns `TLS_OUT_OF_INDEXES`, the allocation of the TLS slot failed. The per thread array of TLS slots is limited in number (64 in Windows NT, 95; 80 in Windows 98; and 1,088 in Windows 2000 and beyond, according to MSDN and empirical results). If too many components in a process are fighting to create large numbers of slots, this error can result. In practice, this seldom arises, but the error condition needs to be handled.

Once a TLS slot has been allocated, the `TlsSetValue` and `TlsGetValue` functions can be used to set and retrieve data from the slots, respectively.

```
BOOL WINAPI TlsSetValue(DWORD dwTlsIndex, LPVOID lpTlsValue);
LPVOID WINAPI TlsGetValue(DWORD dwTlsIndex);
```

Note that the TLS slot `dwTlsIndex` isn't validated at all, other than ensuring it falls within the range of available slots mentioned above

(i.e., so that an out-of-bounds array access doesn't result). This means that, due to programming error, you can accidentally index into a garbage slot and the OS will permit you to do so, leading to unexpected results. In the case where you provide a `dwTlsIndex` value outside of the legal range (e.g., less than 0 or greater than 1,087 on Windows 2000), `TlsSetValue` returns `FALSE` and `TlsGetValue` returns `NULL`. `GetLastError` in both cases will return `ERROR_INVALID_PARAMETER` (87). Note that `NULL` is a legal value to store inside a slot, which can be easily confused with an error condition; `TlsGetValue` indicates the lack of error by setting the last error to `ERROR_SUCCESS`.

Last, you must free a TLS slot when it's no longer in use. If this step is forgotten, other components trying to allocate new slots will be unable to re-use the slot, which is effectively a resource leak and can result in an increase in `TLS_OUT_OF_INDEXES` errors. Freeing a slot is done with the `TlsFree` function.

```
BOOL WINAPI TlsFree(DWORD dwTlsIndex);
```

This function returns `FALSE` if the slot specified by `dwTlsIndex` is invalid, and `TRUE` otherwise. Note that freeing a TLS slot zeroes out the slot memory and simply makes the index available for subsequent calls to `TlsAlloc`. If the `LPVOID` value stored in the slot is a pointer to some block of memory, the memory must be explicitly freed before freeing the index. As soon as the TLS slot is free, the index is no longer safe to use—the slot can be handed out immediately to any other threads attempting to allocate slots concurrently, even before the call to `TlsAlloc` returns, in fact.

It's common to use `DllMain` to perform much of the aforementioned TLS management functions, at least when you're writing a DLL. For example, you can call `TlsAlloc` inside `DLL_PROCESS_ATTACH`, initialize the slot's contents for each thread inside `DLL_THREAD_ATTACH`, free the slot's contents during `DLL_THREAD_DETACH`, and call `TlsFree` inside of `DLL_PROCESS_DETACH`. For instance:

```
#include <windows.h>

DWORD g_dwMyTlsIndex; // Keep index in global or static variable.

BOOL WINAPI DllMain(HINSTANCE hinstDLL,
                    DWORD fdwReason, LPVOID lpvReserved)
```

```
{
    switch (fdwReason)
    {
        case DLL_PROCESS_ATTACH:
            // Allocate a TLS slot.
            if ((g_dwMyTlsIndex = TlsAlloc()) == TLS_OUT_OF_INDEXES)
            {
                ; // Handle the error ...
            }
            break;

        case DLL_PROCESS_DETACH:
            // Free the TLS slot.
            TlsFree(g_dwMyTlsIndex);
            break;

        case DLL_THREAD_ATTACH:
            // Allocate the thread-local data.
            TlsSetValue(g_dwMyTlsIndex, new int[1024]);
            break;

        case DLL_THREAD_DETACH:
            // Free the thread local data.
            int * data = reinterpret_cast<int *>(
                TlsGetValue(g_dwMyTlsIndex));
            delete [] data;
            break;
    }
}
```

Recall from earlier that there are some cases in which thread attach and detach notifications may be missed. If a DLL is loaded dynamically, for example, threads may exist prior to the load, in which case there will not be `DLL_THREAD_ATTACH` notifications for them. For that reason, you will usually need to write your code to check the TLS value to see if it has been initialized and, if not, do so lazily. And as noted earlier, sometimes `DLL_THREAD_DETACH` notifications will be skipped. There is little within reason you can do here, and so killing threads in a manner that skips detach notifications when TLS is involved often leads to leaks. This is yet another reason to avoid APIs like `TerminateThread`.

Static TLS. Instead of writing all of the boilerplate to `TlsAlloc`, `TlsFree`, and manage the per-thread data for each TLS slot, you can use the C++ `__declspec(thread)` modifier to turn a static or global variable into a TLS

variable. To do this, instead of writing the code above to `TlsAlloc` and `TlsFree` a slot in `DllMain`, you can simply write:

```
__declspec(thread) int * g_dwMyTlsIndex;
```

You will still need to initialize and free the array itself, however, on a per thread basis. You can do this inside your own `DllMain` thread attach and detach notification code.

When you use `__declspec(thread)`, the compiler will perform all of the necessary TLS management during its own custom `DllMain` initialization and produces more efficient code when reading from and writing to TLS. Static TLS is substantially faster than dynamic TLS because the compiler has enough information to emit code during compilation that accesses slot addresses with a handful of instructions versus having to make one or more function calls to obtain the address, as with dynamic TLS. The compiler knows the three pieces of information it needs to create code that calculates a TLS slot's address: the TEB address (which it finds in a register), the slot index (known statically), and the offset inside the TEB at which the TLS array begins (constant per architecture). From there, it's a simple matter of some pointer arithmetic to access the data inside a TLS slot.

There are limitations around when you can use static TLS, however. You can only use it from within a program or a DLL that will only be linked statically. In other words, it cannot be used reliably when loaded dynamically via `LoadLibrary`. If you try, you will encounter sporadic access violations when trying to access the TLS data.

Managed Code TLS

Similar to native code, there are two modes of TLS access for managed code. But unlike native code, neither has strict limitations about which kind can be used in any particular program. A single program can, in fact, use a combination of both without worry that they will interact poorly with one another.

Thread Statics. The `ThreadStaticAttribute` type is a custom attribute that can be applied to any static field. (While neither the compiler nor

runtime will prevent you from placing it on an instance field, doing so has no effect whatsoever.) This has the effect of giving each thread a separate copy of that particular static variable. For example, say we had a class `C` with a static field `s_array` and wanted each thread to have its own copy:

```
class C
{
    [ThreadStatic]
    static int[] s_array;
}
```

Now each thread that accesses `s_array` will have its own copy of the value. This is accomplished by the CLR managing an array of TLS slots hanging off the managed thread object. All references to this field are emitted by the JIT as method calls to a special helper function that knows how to access the thread local data. Managed TLS access is slower than static TLS in native code because there are extra hidden function calls and many more indirections.

All call sites that access the variable must check for lazy initialization. There is no direct equivalent to `DllMain`'s attach and detach notifications that can be used for this purpose. Even if a static field initializer is provided, it will only run the first time the variable is accessed (which only works for the first thread that happens to access it). Detach notifications are unnecessary because data store in TLS variables will be garbage collected once the thread dies. It's a good idea, however, to set TLS variables to `null` when they are no longer necessary, particularly if the thread is expected to remain alive for some time to come.

Dynamic TLS. Thread statics are (by far) the preferred means of TLS in managed code. However, there are some circumstances in which you may need more dynamic in the way that TLS is used. For example, with thread statics, the TLS information you need to store must be decided statically at compile-time, and you are required to arrange for a static field to represent the TLS data. Sometimes you may need per object TLS. Dynamic TLS allows you to create slots in this kind of way, very similar to how dynamic TLS in native code works.

To use dynamic TLS, you first allocate a new slot. Two kinds of slots are available, those accessed by name and unnamed slots accessed via a slot object. These are allocated with the `AllocateNamedDataSlot` and `AllocateDataSlot` static methods on the `Thread` class.

```
public static LocalDataStoreSlot AllocateNamedDataSlot(string name);
public static LocalDataStoreSlot AllocateDataSlot();
```

When specifying a named slot, the name supplied must be unique, or else an `ArgumentException` will be thrown. In both cases, a `LocalDataStoreSlot` object will be returned. In the case of `AllocateDataSlot`, you must save this object in order to access the slot. If you lose it, you can't access the slot ever again. For named slots, there is a method to look up the slot, though saving it can avoid unnecessary subsequent lookups.

```
public static LocalDataStoreSlot GetNamedDataSlot(string name);
```

`GetNamedDataSlot` will lazily allocate the slot if it hasn't been created already.

Once a slot has been created, you may set and get data using the `SetData` and `GetData` static methods, respectively. Each accepts a `LocalDataStoreSlot` as an argument, and enables you to store and retrieve references to any kind of object.

```
public static object GetData(LocalDataStoreSlot slot);
public static void SetData(LocalDataStoreSlot slot, object data);
```

Last, it is important to free named slots when you no longer need them with the Thread class's `FreeNamedDataSlot` static method.

```
public static void FreeNamedDataSlot(string name);
```

If you fail to free a named slot, it will stay around until the AppDomain or process exits, and data stored under the slot will remain referenced for each thread that has used it (until the thread itself goes away). The `LocalDataStoreSlot` type has a finalizer, which handles cleanup for unnamed slots once you drop all references to instances. However, the `Thread` object itself keeps a reference to all named slots that have been

created, so even if your program drops all references to it, the slot will not be reclaimed as you might imagine.

Where Are We?

This chapter has reviewed a lot of the basic functionality of Windows and CLR threads. Threads are the underpinning of all concurrency on the Windows OS, and so this foundational knowledge is necessary no matter what kind of concurrency you are using. We looked at the lifetime of threads, including how to start and stop them, in addition to some of the most common attributes of threads such as TLS. Subsequent chapters will build on this information.

The next chapter will do just that and will take the discussion of threads to the next level. It is called Advanced Threads for a reason. This chapter intentionally focused more on the basics while the next chapter intentionally focuses on more low-level and internal details.

FURTHER READING

A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques, and Tools,* Second Edition (Addison-Wesley, 2006).

B. Grunkemeyer. Constrained Execution Regions and Other Errata. Weblog article, http://blogs.msdn.com/bclteam/archive/2005/06/14/429181.aspx (2005).

K. Brown *The .NET Developer's Guide to Windows Security* (Addison-Wesley, 2004).

C. Brumme. Startup, Shutdown, and Related Matters. Weblog article, http://blogs.msdn.com/cbrumme/archive/2003/08/20/51504.aspx (2003).

S. Currie. Mixed DLL Loading Problem. MSDN documentation, http://msdn2.microsoft.com/enus/library/Aa290048(VS.71).aspx (2003).

J. Duffy. Atomicity and Asynchronous Exception Failures. Weblog article, http://www.bluebytesoftware.com/blog/2005/03/19/AtomicityAndAsynchronousExceptionFailures.aspx (2005).

J. Duffy. The CLR Commits the Whole Stack. Weblog article, http://www.bluebytesoftware.com/blog/2007/03/10/TheCLRCommitsTheWholeStack.aspx (2007).

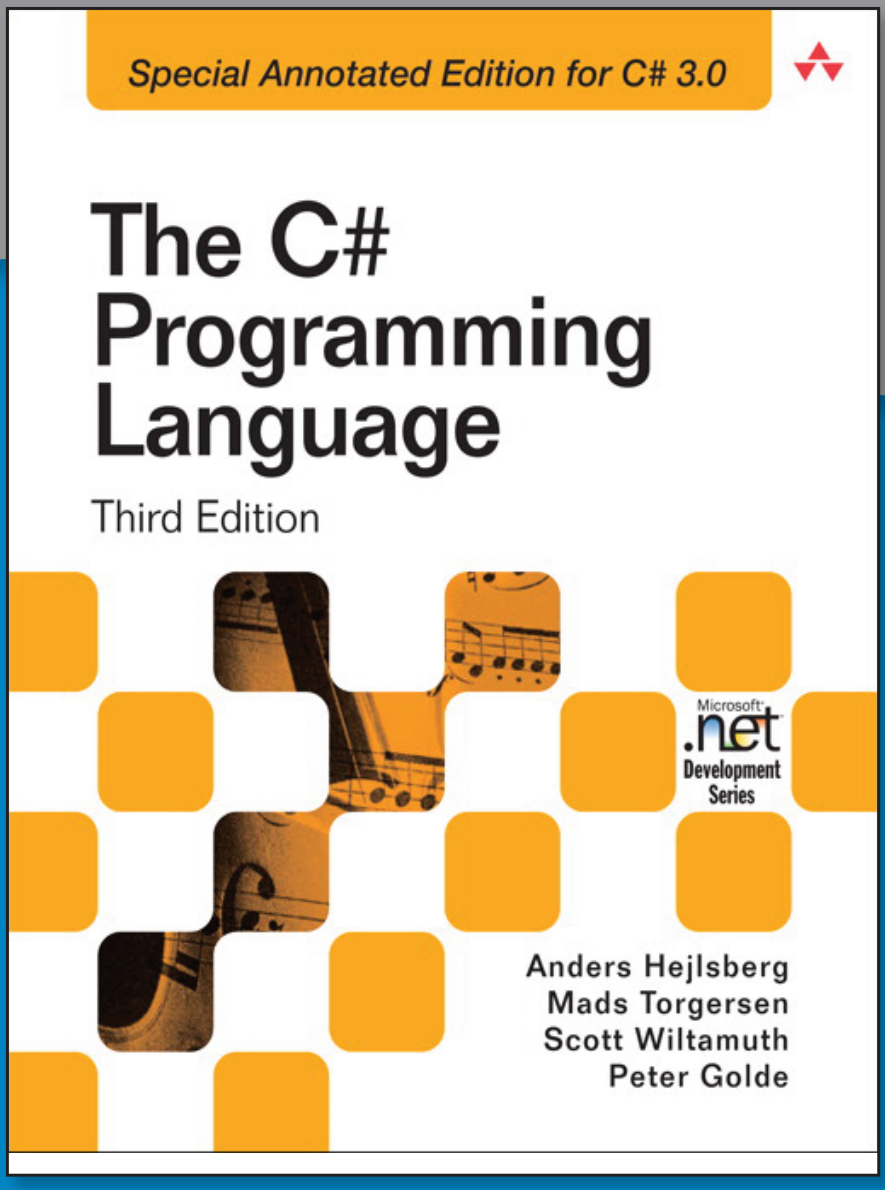
MSDN. Visual C++: Initialization of Mixed Assemblies. MSDN documentation, http://msdn2.microsoft.com/en-us/library/ms173266(VS.80).aspx.

MSDN. Best Practices for Creating DLLs. MSDN documentation, http://www.microsoft.com/whdc/driver/kernel/DLL_bestprac.mspx (2006).

M. Pietrek. A Crash Course on the Depths of Win32™ Structured Exception Handling. *Microsoft Systems Journal*, http://www.microsoft.com/msj/0197/Exception/Exception.aspx (1997).

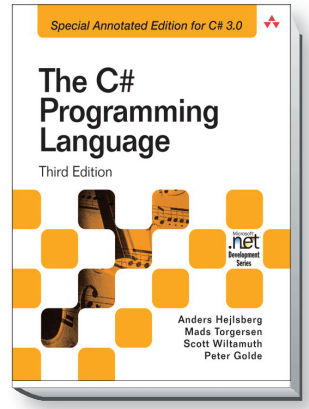
S. Pratschner. *Customizing the Microsoft .NET Framework Common Language Runtime* (MS Press, 2005).

S. Toub. High Availability: Keep Your Code Running with the Reliability Features of the .NET Framework. *MSDN Magazine* (October 2005).



Anders Hejlsberg
Mads Torgersen
Scott Wiltamuth
Peter Golde

BUY ME



The C# Programming Language

Third Edition

The popular C# programming language combines the high productivity of rapid application development languages with the raw power of C and C++. Now, C# 3.0 adds functional programming techniques and LINQ, Language INtegrated Query. The C# Programming Language, Third Edition, is the authoritative and annotated technical reference for C# 3.0.

Written by Anders Hejlsberg, the language's architect, and his colleagues, Mads Torgersen, Scott Wiltamuth, and Peter Golde, this volume has been completely updated and reorganized for C# 3.0. The book provides the complete specification of the language, along with descriptions, reference materials, code samples, and annotations from nine prominent C# gurus.

The many annotations–a new feature in this edition–bring a depth and breadth of understanding rarely found in any programming book. As the main text of the book introduces the concepts of the C# language, cogent annotations explain why they are important, how they are used, how they relate to other languages, and even how they evolved.

This book is the definitive, must-have reference for any developer who wants to understand C#.

About the Authors

Anders Hejlsberg is a programming legend. He is the architect of the C# language and a Microsoft Technical Fellow. He joined Microsoft in 1996, following a 13-year career at Borland, where he was the chief architect of Delphi and Turbo Pascal.

Mads Torgersen is a senior program manager at Microsoft. As the program manager for the C# language, he runs the C# language design meetings and maintains the C# language specification. Prior to joining Microsoft in 2005, Mads was an associate professor at the University of Aarhus, teaching and researching object-oriented programming languages. There, he led the group that designed and implemented generic wildcards for the Java Programming Language.

Scott Wiltamuth is partner program manager for Visual Studio. While at Microsoft, he has worked on a wide range of developer-oriented projects, including Visual Basic, VBScript, JScript, Visual J++, and Visual C#. Scott is one of the designers of the C# language, and holds bachelor's and master's degrees in computer science from Stanford University.

Before leaving Microsoft, **Peter Golde** served as the lead developer of Microsoft's C# compiler. As the primary Microsoft representative on the ECMA committee that standardized C#, he led the implementation of the compiler and worked on the language design.

AVAILABLE
- BOOK: 9780321562999
- SAFARI ONLINE
- EBOOK: 0321592263
- KINDLE: 0321592255

Professional Features

This is the definitive reference to the C# Programming Language, direct from the architect, and updated for the new version 3.0

- New to this edition - insightful, valuable annotations from eleven leading C# programmers, available nowhere else
- Fully updated for the changes to C# 3.0, especially LINQ
- C# has become the most widely used language for Windows development
- Anders Hejlsberg is the creator of C# and a true legend among programmers

Addison Wesley

informit.com/aw

3. Basic Concepts

3.1 Application Start-Up

An assembly that has an ***entry point*** is called an ***application***. When an application is run, a new ***application domain*** is created. Several different instantiations of an application may exist on the same machine at the same time, and each has its own application domain.

An application domain enables application isolation by acting as a container for the application state. An application domain acts as a container and boundary for the types defined in the application and the class libraries it uses. Types loaded into one application domain are distinct from the same types loaded into another application domain, and instances of objects are not directly shared between application domains. For instance, each application domain has its own copy of static variables for these types, and a static constructor for a type is run at most once per application domain. Implementations are free to provide implementation-specific policy or mechanisms for the creation and destruction of application domains.

Application start-up occurs when the execution environment calls a designated method, which is referred to as the application's entry point. This entry point method is always named `Main`, and it can have one of the following signatures:

```
static void Main() {...}

static void Main(string[] args) {...}

static int Main() {...}

static int Main(string[] args) {...}
```

As shown, the entry point may optionally return an `int` value. This return value is used in application termination (§3.2).

The entry point may optionally have one formal parameter. The parameter may have any name, but the type of the parameter must be `string[]`. If the formal parameter is present, the execution environment creates and passes a `string[]` argument containing the command-line arguments that were specified when the application was started. The `string[]` argument is never null, but it may have a length of zero if no command-line arguments were specified.

Because C# supports method overloading, a class or struct may contain multiple definitions of some method, provided each definition has a different signature. However, within a single program, no class or struct may contain more than one method called `Main` whose definition qualifies it to be used as an application entry point. Other overloaded versions of `Main` are permitted, however, provided they have more than one parameter, or their only parameter is something other than type `string[]`.

An application can be made up of multiple classes or structs. It is possible for more than one of these classes or structs to contain a method called `Main` whose definition qualifies it to be used as an application entry point. In such cases, an external mechanism (such as a command-line compiler option) must be used to select one of these `Main` methods as the entry point.

> **ERIC LIPPERT** The "csc" command-line compiler provides the `/main:` switch for this purpose.

In C#, every method must be defined as a member of a class or struct. Ordinarily, the declared accessibility (§3.5.1) of a method is determined by the access modifiers (§10.3.5) specified in its declaration. Similarly, the declared accessibility of a type is determined by the access modifiers specified in its declaration. For a given method of a given type to be callable, both the type and the member must be accessible. However, the application entry point is a special case. Specifically, the execution environment can access the application's entry point regardless of its declared accessibility and regardless of the declared accessibility of its enclosing type declarations.

The application entry point method may not be in a generic class declaration.

In all other respects, entry point methods behave like methods that are not entry points.

3.2 Application Termination

Application termination returns control to the execution environment.

If the return type of the application's ***entry point*** method is `int`, the value returned serves as the application's ***termination status code***. The purpose of this code is to allow communication of success or failure to the execution environment.

If the return type of the entry point method is `void`, reaching the right brace (`}`) that terminates the method, or executing a `return` statement that has no expression, results in a termination status code of `0`.

BILL WAGNER The following rule is an important difference between C# and other managed environments.

Prior to an application's termination, destructors for all of its objects that have not yet been garbage collected are called, unless such cleanup has been suppressed (by a call to the library method `GC.SuppressFinalize`, for example).

3.3 Declarations

Declarations in a C# program define the constituent elements of the program. C# programs are organized using namespaces (§9), which can contain type declarations and nested namespace declarations. Type declarations (§9.6) are used to define classes (§10), structs (§10.14), interfaces (§13), enums (§14), and delegates (§15). The kinds of members permitted in a type declaration depend on the form of the type declaration. For instance, class declarations can contain declarations for constants (§10.4), fields (§10.5), methods (§10.6), properties (§10.7), events (§10.8), indexers (§10.9), operators (§10.10), instance constructors (§10.11), static constructors (§10.12), destructors (§10.13), and nested types (§10.3.8).

A declaration defines a name in the ***declaration space*** to which the declaration belongs. Except for overloaded members (§3.6), it is a compile-time error to have two or more declarations that introduce members with the same name in a declaration space. It is never possible for a declaration space to contain different kinds of members with the same name. For example, a declaration space can never contain a field and a method with the same name.

ERIC LIPPERT "Declaration spaces" are frequently confused with "scopes." Although related conceptually, they have quite different purposes. The scope of a named element is the region of the program text in which that element may be referred to by name without additional qualification. By contrast, the declaration space of an element is the region in which no two elements may have the same name (or same signature, for methods).

Several types of declaration spaces are possible:

- Within all source files of a program, *namespace-member-declaration*s with no enclosing *namespace-declaration* are members of a single combined declaration space called the ***global declaration space***.
- Within all source files of a program, *namespace-member-declaration*s within *namespace-declaration*s that have the same fully qualified namespace name are members of a single combined declaration space.

- Each class, struct, or interface declaration creates a new declaration space. Names are introduced into this declaration space through *class-member-declaration*s, *struct-member-declaration*s, *interface-member-declaration*s, or *type-parameter*s. Except for overloaded instance constructor declarations and static constructor declarations, a class or struct cannot contain a member declaration with the same name as the class or struct. A class, struct, or interface permits the declaration of overloaded methods and indexers. Furthermore, a class or struct permits the declaration of overloaded instance constructors and operators. For example, a class, struct, or interface may contain multiple method declarations with the same name, provided these method declarations have different signatures (§3.6). Note that base classes do not contribute to the declaration space of a class, and base interfaces do not contribute to the declaration space of an interface. Thus, a derived class or interface is allowed to declare a member with the same name as an inherited member. Such a member is said to ***hide*** the inherited member.
- Each delegate declaration creates a new declaration space. Names are introduced into this declaration space through formal parameters (*fixed-parameter*s and *parameter-array*s) and *type-parameter*s.
- Each enumeration declaration creates a new declaration space. Names are introduced into this declaration space through *enum-member-declarations*.
- Each method declaration, indexer declaration, operator declaration, instance constructor declaration, and anonymous function creates a new declaration space called a ***local variable declaration space***. Names are introduced into this declaration space through formal parameters (*fixed-parameter*s and *parameter-array*s) and *type-parameter*s. The body of the function member or anonymous function, if any, is considered to be nested within the local variable declaration space. It is an error for a local variable declaration space and a nested local variable declaration space to contain elements with the same name. Thus, within a nested declaration space, it is not possible to declare a local variable or constant with the same name as a local variable or constant in an enclosing declaration space. It is possible for two declaration spaces to contain elements with the same name as long as neither declaration space contains the other.
- Each *block* or *switch-block*, as well as a *for*, *foreach*, and *using* statement, creates a local variable declaration space for local variables and local constants. Names are introduced into this declaration space through *local-variable-declaration*s and *local-constant-declaration*s. Note that blocks that occur as or within the body of a function member or anonymous function are nested within the local variable declaration space declared by those functions for their parameters. Thus, it is an error to have, for example, a method with a local variable and a parameter of the same name.
- Each *block* or *switch-block* creates a separate declaration space for labels. Names are introduced into this declaration space through *labeled-statement*s, and the names are referenced

through *goto-statement*s. The ***label declaration space*** of a block includes any nested blocks. Thus, within a nested block, it is not possible to declare a label with the same name as a label in an enclosing block.

The textual order in which names are declared is generally of no significance. In particular, textual order is not significant for the declaration and use of namespaces, constants, methods, properties, events, indexers, operators, instance constructors, destructors, static constructors, and types. Declaration order is significant in the following ways:

- Declaration order for field declarations and local variable declarations determines the order in which their initializers (if any) are executed.
- Local variables must be defined before they are used (§3.7).
- Declaration order for enum member declarations (§14.3) is significant when *constant-expression* values are omitted.

The declaration space of a namespace is "open-ended," and two namespace declarations with the same fully qualified name contribute to the same declaration space. For example,

```
namespace Megacorp.Data
{
    class Customer
    {
        ...
    }
}

namespace Megacorp.Data
{
    class Order
    {
        ...
    }
}
```

Here the two namespace declarations contribute to the same declaration space—in this case, declaring two classes with the fully qualified names `Megacorp.Data.Customer` and `Megacorp.Data.Order`. Because the two declarations contribute to the same declaration space, a compile-time error would have occurred if each contained a declaration of a class with the same name.

BILL WAGNER Think of namespaces as a tool to manage the logical organization of your code. Assemblies manage the physical organization of your code.

As specified earlier, the declaration space of a block includes any nested blocks. Thus, in the following example, the `F` and `G` methods result in a compile-time error because the name `i` is declared in the outer block and cannot be redeclared in the inner block. The `H` and `I` methods are valid, however, because the two `i`'s are declared in separate non-nested blocks.

```
class A
{
    void F() {
        int i = 0;
        if (true) {
            int i = 1;
        }
    }

    void G() {
        if (true) {
            int i = 0;
        }
        int i = 1;
    }

    void H() {
        if (true) {
            int i = 0;
        }
        if (true) {
            int i = 1;
        }
    }

    void I() {
        for (int i = 0; i < 10; i++)
            H();
        for (int i = 0; i < 10; i++)
            H();
    }
}
```


3.4 Members

Namespaces and types have ***members***. The members of an entity are generally available through the use of a qualified name that starts with a reference to the entity, followed by a "`.`" token, followed by the name of the member.

Members of a type are either declared in the type declaration or ***inherited*** from the base class of the type. When a type inherits from a base class, all members of the base class, except instance constructors, destructors, and static constructors, become members of the derived type. The declared accessibility of a base class member does not control whether

the member is inherited—inheritance extends to any member that isn't an instance constructor, static constructor, or destructor. However, an inherited member may not be accessible in a derived type, either because of its declared accessibility (§3.5.1) or because it is hidden by a declaration in the type itself (§3.7.1.2).

3.4.1 Namespace Members

Namespaces and types that have no enclosing namespace are members of the ***global namespace***. This corresponds directly to the names declared in the global declaration space.

Namespaces and types declared within a namespace are members of that namespace. This corresponds directly to the names declared in the declaration space of the namespace.

Namespaces have no access restrictions. It is not possible to declare private, protected, or internal namespaces, and namespace names are always publicly accessible.

3.4.2 Struct Members

The members of a struct are the members declared in the struct and the members inherited from the struct's direct base class `System.ValueType` and the indirect base class `object`.

The members of a simple type correspond directly to the members of the struct type aliased by the simple type:

- The members of `sbyte` are the members of the `System.SByte` struct.
- The members of `byte` are the members of the `System.Byte` struct.
- The members of `short` are the members of the `System.Int16` struct.
- The members of `ushort` are the members of the `System.UInt16` struct.
- The members of `int` are the members of the `System.Int32` struct.
- The members of `uint` are the members of the `System.UInt32` struct.
- The members of `long` are the members of the `System.Int64` struct.
- The members of `ulong` are the members of the `System.UInt64` struct.
- The members of `char` are the members of the `System.Char` struct.
- The members of `float` are the members of the `System.Single` struct.
- The members of `double` are the members of the `System.Double` struct.
- The members of `decimal` are the members of the `System.Decimal` struct.
- The members of `bool` are the members of the `System.Boolean` struct.

3.4.3 Enumeration Members

The members of an enumeration are the constants declared in the enumeration and the members inherited from the enumeration's direct base class `System.Enum` and the indirect base classes `System.ValueType` and `object`.

3.4.4 Class Members

The members of a class are the members declared in the class and the members inherited from the base class (except for class `object`, which has no base class). The members inherited from the base class include the constants, fields, methods, properties, events, indexers, operators, and types of the base class, but not the instance constructors, destructors, and static constructors of the base class. Base class members are inherited without regard to their accessibility.

A class declaration may contain declarations of constants, fields, methods, properties, events, indexers, operators, instance constructors, destructors, static constructors, and types.

The members of `object` and `string` correspond directly to the members of the class types they alias:

- The members of `object` are the members of the `System.Object` class.
- The members of `string` are the members of the `System.String` class.

3.4.5 Interface Members

The members of an interface are the members declared in the interface and in all base interfaces of the interface. The members in class `object` are not, strictly speaking, members of any interface (§13.2). However, the members in class `object` are available via member lookup in any interface type (§7.3).

3.4.6 Array Members

The members of an array are the members inherited from class `System.Array`.

3.4.7 Delegate Members

The members of a delegate are the members inherited from class `System.Delegate`.

VLADIMIR RESHETNIKOV In the Microsoft implementation of C#, the members of a delegate also include the instance methods `Invoke`, `BeginInvoke`, `EndInvoke`, and the members inherited from class `System.MulticastDelegate`.

3.5 Member Access

Declarations of members allow control over member access. The accessibility of a member is established by the declared accessibility (§3.5.1) of the member combined with the accessibility of the immediately containing type, if any.

When access to a particular member is allowed, the member is said to be ***accessible***. Conversely, when access to a particular member is disallowed, the member is said to be ***inaccessible***. Access to a member is permitted when the textual location in which the access takes place is included in the accessibility domain (§3.5.2) of the member.

3.5.1 Declared Accessibility

The ***declared accessibility*** of a member can be one of the following:

- Public, which is selected by including a `public` modifier in the member declaration. The intuitive meaning of `public` is "access not limited."
- Protected, which is selected by including a `protected` modifier in the member declaration. The intuitive meaning of `protected` is "access limited to the containing class or types derived from the containing class."
- Internal, which is selected by including an `internal` modifier in the member declaration. The intuitive meaning of `internal` is "access limited to this program."
- Protected internal (meaning protected or internal), which is selected by including both a `protected` modifier and an `internal` modifier in the member declaration. The intuitive meaning of `protected internal` is "access limited to this program or types derived from the containing class."
- Private, which is selected by including a `private` modifier in the member declaration. The intuitive meaning of `private` is "access limited to the containing type."

> **JESSE LIBERTY** There is always a default accessibility, and it is always good programming practice to declare the accessibility explicitly. This makes for code that is easier to read and far easier to maintain.

Depending on the context in which a member declaration takes place, only certain types of declared accessibility are permitted. Furthermore, when a member declaration does not include any access modifiers, the context in which the declaration takes place determines the default declared accessibility.

- Namespaces implicitly have `public` declared accessibility. No access modifiers are allowed on namespace declarations.

- Types declared in compilation units or namespaces can have `public` or `internal` declared accessibility and default to `internal` declared accessibility.
- Class members can have any of the five kinds of declared accessibility and default to `private` declared accessibility. (Note that a type declared as a member of a class can have any of the five kinds of declared accessibility, whereas a type declared as a member of a namespace can have only `public` or `internal` declared accessibility.)

VLADIMIR RESHETNIKOV If a sealed class declares a `protected` or `protected internal` member, a warning is issued. If a static class declares a `protected` or `protected internal` member, a compile-time error occurs (CS1057).

- Struct members can have `public`, `internal`, or `private` declared accessibility and default to `private` declared accessibility because structs are implicitly sealed. Struct members introduced in a struct (that is, not inherited by that struct) cannot have `protected` or `protected internal` declared accessibility. (Note that a type declared as a member of a struct can have `public`, `internal`, or `private` declared accessibility, whereas a type declared as a member of a namespace can have only `public` or `internal` declared accessibility.)
- Interface members implicitly have `public` declared accessibility. No access modifiers are allowed on interface member declarations.
- Enumeration members implicitly have `public` declared accessibility. No access modifiers are allowed on enumeration member declarations.

JOSEPH ALBAHARI The rationale behind these rules is that the default declared accessibility for any construct is the minimum accessibility that it requires to be useful. Minimizing accessibility is positive in the sense that it promotes encapsulation.

CHRIS SELLS Except for operator precedence, which I believe should be made explicit if any doubt ever arises, I'm a big fan of less code. When I see unnecessary use of `private` or `internal`, it looks like noise to me. For example,

```
internal class Foo { private int x; public int X { get { return x; } } }
```

should really be written as

```
class Foo { int x; public int X { get { return x; } } }
```

ERIC LIPPERT There is a difference between the "declared" accessibility and the actual effective accessibility. For example, a method declared as `public` on a class declared as `internal` is, for most practical purposes, an internal method.

A good way to think about this issue is to recognize that a public class member is public only to the entities that have access to the class.

3.5.2 Accessibility Domains

The ***accessibility domain*** of a member consists of the (possibly disjoint) sections of program text in which access to the member is permitted. For purposes of defining the accessibility domain of a member, a member is said to be ***top-level*** if it is not declared within a type, and a member is said to be ***nested*** if it is declared within another type. Furthermore, the ***program text*** of a program is defined as all program text contained in all source files of the program, and the program text of a type is defined as all program text contained between the opening and closing "`{`" and "`}`" tokens in the *class-body*, *struct-body*, *interface-body*, or *enum-body* of the type (including, possibly, types that are nested within the type).

The accessibility domain of a predefined type (such as `object`, `int`, or `double`) is unlimited.

The accessibility domain of a top-level unbound type `T` (§4.4.3) that is declared in a program `P` is defined as follows:

- If the declared accessibility of `T` is `public`, the accessibility domain of `T` is the program text of `P` and any program that references `P`.
- If the declared accessibility of `T` is `internal`, the accessibility domain of `T` is the program text of `P`.

From these definitions, it follows that the accessibility domain of a top-level unbound type is always at least the program text of the program in which that type is declared.

The accessibility domain for a constructed type $T<A_1, \ldots, A_N>$ is the intersection of the accessibility domain of the unbound generic type `T` and the accessibility domains of the type arguments $A_1, \ldots, A_N$.

The accessibility domain of a nested member `M` declared in a type `T` within a program `P` is defined as follows (noting that `M` itself may possibly be a type):

- If the declared accessibility of M is public, the accessibility domain of M is the accessibility domain of T.
- If the declared accessibility of M is protected internal, let D be the union of the program text of P and the program text of any type derived from T, which is declared outside P. The accessibility domain of M is the intersection of the accessibility domain of T with D.
- If the declared accessibility of M is protected, let D be the union of the program text of T and the program text of any type derived from T. The accessibility domain of M is the intersection of the accessibility domain of T with D.
- If the declared accessibility of M is internal, the accessibility domain of M is the intersection of the accessibility domain of T with the program text of P.
- If the declared accessibility of M is private, the accessibility domain of M is the program text of T.

From these definitions, it follows that the accessibility domain of a nested member is always at least the program text of the type in which the member is declared. Furthermore, it follows that the accessibility domain of a member is never more inclusive than the accessibility domain of the type in which the member is declared.

In intuitive terms, when a type or member M is accessed, the following steps are evaluated to ensure that the access is permitted:

- First, if M is declared within a type (as opposed to a compilation unit or a namespace), a compile-time error occurs if that type is not accessible.
- Then, if M is public, the access is permitted.
- Otherwise, if M is protected internal, the access is permitted if it occurs within the program in which M is declared, or if it occurs within a class derived from the class in which M is declared and takes place through the derived class type (§3.5.3).
- Otherwise, if M is protected, the access is permitted if it occurs within the class in which M is declared, or if it occurs within a class derived from the class in which M is declared and takes place through the derived class type (§3.5.3).
- Otherwise, if M is internal, the access is permitted if it occurs within the program in which M is declared.
- Otherwise, if M is private, the access is permitted if it occurs within the type in which M is declared.
- Otherwise, the type or member is inaccessible, and a compile-time error occurs.

In the example

```
public class A
{
    public static int X;
    internal static int Y;
    private static int Z;
}

internal class B
{
    public static int X;
    internal static int Y;
    private static int Z;

    public class C
    {
        public static int X;
        internal static int Y;
        private static int Z;
    }

    private class D
    {
        public static int X;
        internal static int Y;
        private static int Z;
    }
}
```

the classes and members have the following accessibility domains:

- The accessibility domain of `A` and `A.X` is unlimited.
- The accessibility domain of `A.Y`, `B`, `B.X`, `B.Y`, `B.C`, `B.C.X`, and `B.C.Y` is the program text of the containing program.
- The accessibility domain of `A.Z` is the program text of `A`.
- The accessibility domain of `B.Z` and `B.D` is the program text of `B`, including the program text of `B.C` and `B.D`.
- The accessibility domain of `B.C.Z` is the program text of `B.C`.
- The accessibility domain of `B.D.X` and `B.D.Y` is the program text of `B`, including the program text of `B.C` and `B.D`.
- The accessibility domain of `B.D.Z` is the program text of `B.D`.

As this example illustrates, the accessibility domain of a member is never larger than that of a containing type. For example, even though all `X` members have `public` declared accessibility, all members except `A.X` have accessibility domains that are constrained by a containing type.

JOSEPH ALBAHARI Declaring a public member within an `internal` type might seem pointless, given that the member's visibility will be capped at `internal`. It can make sense, however, if the public member modifier is interpreted as meaning "having the same visibility as the containing type."

A good question to ask, in deciding whether to declare a member of an internal type `public` or `internal`, is this: "If the type was later promoted to `public`, would I want this member to become `public`, too?" If the answer is yes, one could argue for declaring the member as `public` from the outset.

As described in §3.4, all members of a base class, except for instance constructors, destructors, and static constructors, are inherited by derived types—even private members of a base class. However, the accessibility domain of a private member includes only the program text of the type in which the member is declared. In the example

```
class A
{
    int x;

    static void F(B b) {
        b.x = 1;        // Okay
    }
}

class B: A
{
    static void F(B b) {
        b.x = 1;        // Error, x not accessible
    }
}
```

the `B` class inherits the private member `x` from the `A` class. Because the member is private, it is only accessible within the *class-body* of `A`. Thus, the access to `b.x` succeeds in the `A.F` method, but fails in the `B.F` method.

BILL WAGNER Notice that the inaccessible methods also obviate the need for the new modifier on `B.F()`.

3.5.3 Protected Access for Instance Members

When a `protected` instance member is accessed outside the program text of the class in which it is declared, and when a `protected internal` instance member is accessed outside the program text of the program in which it is declared, the access must take place within

a class declaration that derives from the class in which it is declared. Furthermore, the access is required to take place *through* an instance of that derived class type or a class type constructed from it. This restriction prevents one derived class from accessing protected members of other derived classes, even when the members are inherited from the same base class.

> **ERIC LIPPERT** For instance, suppose you have a base class `Animal` and two derived classes, `Mammal` and `Reptile`, and `Animal` has a protected method `Feed()`. Then the `Mammal` code can call `Feed()` on a `Mammal` or any subclass of `Mammal` (`Tiger`, say).
>
> `Mammal` code cannot call `Feed()` on an expression of type `Reptile` because there is no inheritance relationship between `Mammal` and `Reptile`.
>
> Furthermore, because an expression of type `Animal` might actually be a `Reptile` at runtime, `Mammal` code also cannot call `Feed()` on an expression of type `Animal`.

Let `B` be a base class that declares a protected instance member `M`, and let `D` be a class that derives from `B`. Within the *class-body* of `D`, access to `M` can take one of the following forms:

- An unqualified *type-name* or *primary-expression* of the form `M`
- A *primary-expression* of the form `E.M`, provided the type of `E` is `T` or a class derived from `T`, where `T` is the class type `D`, or a class type constructed from `D`
- A *primary-expression* of the form `base.M`

In addition to these forms of access, a derived class can access a protected instance constructor of a base class in a *constructor-initializer* (§10.11.1).

In the example

```
public class A
{
    protected int x;

    static void F(A a, B b) {
        a.x = 1;        // Okay
        b.x = 1;        // Okay
    }
}

public class B: A
{
    static void F(A a, B b) {
        a.x = 1;        // Error, must access through instance of B
        b.x = 1;        // Okay
    }
}
```

within A, it is possible to access x through instances of both A and B, because in either case the access takes place *through* an instance of A or a class derived from A. However, within B, it is not possible to access x through an instance of A, because A does not derive from B.

In the example

```
class C<T>
{
    protected T x;
}

class D<T>: C<T>
{
    static void F() {
        D<T> dt = new D<T>();
        D<int> di = new D<int>();
        D<string> ds = new D<string>();
        dt.x = default(T);
        di.x = 123;
        ds.x = "test";
    }
}
```

the three assignments to x are permitted because all of them take place through instances of class types constructed from the generic type.

CHRIS SELLS To minimize the surface area of a class or namespace, I recommend keeping things private/internal until they prove to be necessary in a wider scope. Refactoring is your friend here.

BILL WAGNER These rules exist to allow for the separate evolution of components in different assemblies. You should not incorporate these rules into your regular design patterns.

3.5.4 Accessibility Constraints

Several constructs in the C# language require a type to be ***at least as accessible as*** a member or another type. A type T is said to be at least as accessible as a member or type M if the accessibility domain of T is a superset of the accessibility domain of M. In other words, T is at least as accessible as M if T is accessible in all contexts in which M is accessible.

VLADIMIR RESHETNIKOV For the purposes of this paragraph, only accessibility modifiers are considered. For instance, if a protected member is declared in a `public sealed` class, the fact that this class cannot have descendants (and, therefore, the accessibility domain of this member does not include any descendants) is not considered.

The following accessibility constraints exist:

- The direct base class of a class type must be at least as accessible as the class type itself.
- The explicit base interfaces of an interface type must be at least as accessible as the interface type itself.
- The return type and parameter types of a delegate type must be at least as accessible as the delegate type itself.
- The type of a constant must be at least as accessible as the constant itself.
- The type of a field must be at least as accessible as the field itself.
- The return type and parameter types of a method must be at least as accessible as the method itself.
- The type of a property must be at least as accessible as the property itself.
- The type of an event must be at least as accessible as the event itself.
- The type and parameter types of an indexer must be at least as accessible as the indexer itself.
- The return type and parameter types of an operator must be at least as accessible as the operator itself.
- The parameter types of an instance constructor must be at least as accessible as the instance constructor itself.

In the example

```
class A {...}
public class B: A {...}
```

the `B` class results in a compile-time error because `A` is not at least as accessible as `B`.

Likewise, in the example

```
class A {...}
public class B
{
```

```
    A F() {...}
    internal A G() {...}
    public A H() {...}
}
```

the `H` method in `B` results in a compile-time error because the return type `A` is not at least as accessible as the method.

3.6 Signatures and Overloading

Methods, instance constructors, indexers, and operators are characterized by their ***signatures***:

- The signature of a method consists of the name of the method, the number of type parameters and the type and kind (value, reference, or output) of each of its formal parameters, considered in order from left to right. For these purposes, any type parameter of the method that occurs in the type of a formal parameter is identified not by its name, but by its ordinal position in the type argument list of the method. The signature of a method specifically does not include the return type, the `params` modifier that may be specified for the rightmost parameter, nor the optional type parameter constraints.
- The signature of an instance constructor consists of the type and kind (value, reference, or output) of each of its formal parameters, considered in order from left to right. The signature of an instance constructor specifically does not include the `params` modifier that may be specified for the rightmost parameter.
- The signature of an indexer consists of the type of each of its formal parameters, considered in order from left to right. The signature of an indexer specifically does not include the element type, nor does it include the `params` modifier that may be specified for the rightmost parameter.
- The signature of an operator consists of the name of the operator and the type of each of its formal parameters, considered in order from left to right. The signature of an operator specifically does not include the result type.

Signatures are the enabling mechanism for ***overloading*** of members in classes, structs, and interfaces:

- Overloading of methods permits a class, struct, or interface to declare multiple methods with the same name, provided their signatures are unique within that class, struct, or interface.

- Overloading of instance constructors permits a class or struct to declare multiple instance constructors, provided their signatures are unique within that class or struct.
- Overloading of indexers permits a class, struct, or interface to declare multiple indexers, provided their signatures are unique within that class, struct, or interface.
- Overloading of operators permits a class or struct to declare multiple operators with the same name, provided their signatures are unique within that class or struct.

Although `out` and `ref` parameter modifiers are considered part of a signature, members declared in a single type cannot differ in signature solely by `ref` and `out`. A compile-time error occurs if two members are declared in the same type with signatures that would be the same if all parameters in both methods with `out` modifiers were changed to `ref` modifiers. For other purposes of signature matching (e.g., hiding or overriding), `ref` and `out` are considered part of the signature and do not match each other. (This restriction is intended to allow C# programs to be easily translated to run on the Common Language Infrastructure [CLI], which does not provide a way to define methods that differ solely in terms of `ref` and `out`.)

The following example shows a set of overloaded method declarations along with their signatures.

```
interface ITest
{
    void F();                       // F()
    void F(int x);                  // F(int)
    void F(ref int x);              // F(ref int)
    void F(out int x);              // F(out int)       error
    void F(int x, int y);           // F(int, int)
    int F(string s);                // F(string)
    int F(int x);                   // F(int)           error
    void F(string[] a);             // F(string[])
    void F(params string[] a);      // F(string[])      error
}
```

Note that any `ref` and `out` parameter modifiers (§10.6.1) are part of a signature. Thus, `F(int)` and `F(ref int)` are unique signatures. However, `F(ref int)` and `F(out int)` cannot be declared within the same interface because their signatures differ solely by `ref` and `out`. Also, note that the return type and the `params` modifier are not part of a signature, so it is not possible to overload the declarations solely based on return type or on the inclusion or exclusion of the `params` modifier. As such, the declarations of the methods `F(int)` and `F(params string[])` identified above result in a compile-time error.

3.7 Scopes

The ***scope*** of a name is the region of program text within which it is possible to refer to the entity declared by the name without qualification of the name. Scopes can be ***nested***, and an inner scope may redeclare the meaning of a name from an outer scope (this does not, however, remove the restriction imposed by §3.3 that within a nested block it is not possible to declare a local variable with the same name as a local variable in an enclosing block). The name from the outer scope is then said to be ***hidden*** in the region of program text covered by the inner scope, and access to the outer name is possible only by qualifying the name.

- The scope of a namespace member declared by a *namespace-member-declaration* (§9.5) with no enclosing *namespace-declaration* is the entire program text.
- The scope of a namespace member declared by a *namespace-member-declaration* within a *namespace-declaration* whose fully qualified name is `N` is the *namespace-body* of every *namespace-declaration* whose fully qualified name is `N` or starts with `N`, followed by a period.
- The scope of a name defined by an *extern-alias-directive* extends over the *using-directives*, *global-attributes*, and *namespace-member-declarations* of its immediately containing compilation unit or namespace body. An *extern-alias-directive* does not contribute any new members to the underlying declaration space. In other words, an *extern-alias-directive* is not transitive, but rather affects only the compilation unit or namespace body in which it occurs.
- The scope of a name defined or imported by a *using-directive* (§9.4) extends over the *namespace-member-declarations* of the *compilation-unit* or *namespace-body* in which the *using-directive* occurs. A *using-directive* may make zero or more namespace or type names available within a particular *compilation-unit* or *namespace-body*, but it does not contribute any new members to the underlying declaration space. In other words, a *using-directive* is not transitive, but rather affects only the *compilation-unit* or *namespace-body* in which it occurs.
- The scope of a type parameter declared by a *type-parameter-list* on a *class-declaration* (§10.1) is the *class-base*, *type-parameter-constraints-clauses*, and *class-body* of that *class-declaration*.
- The scope of a type parameter declared by a *type-parameter-list* on a *struct-declaration* (§11.1) is the *struct-interfaces*, *type-parameter-constraints-clauses*, and *struct-body* of that *struct-declaration*.
- The scope of a type parameter declared by a *type-parameter-list* on an *interface-declaration* (§13.1) is the *interface-base*, *type-parameter-constraints-clauses*, and *interface-body* of that *interface-declaration*.

- The scope of a type parameter declared by a *type-parameter-list* on a *delegate-declaration* (§15.1) is the *return-type*, *formal-parameter-list*, and *type-parameter-constraints-clauses* of that *delegate-declaration*.
- The scope of a member declared by a *class-member-declaration* (§10.1.6) is the *class-body* in which the declaration occurs. In addition, the scope of a class member extends to the *class-body* of those derived classes included in the accessibility domain (§3.5.2) of the member.
- The scope of a member declared by a *struct-member-declaration* (§11.2) is the *struct-body* in which the declaration occurs.
- The scope of a member declared by an *enum-member-declaration* (§14.3) is the *enum-body* in which the declaration occurs.
- The scope of a parameter declared in a *method-declaration* (§10.6) is the *method-body* of that *method-declaration*.
- The scope of a parameter declared in an *indexer-declaration* (§10.9) is the *accessor-declarations* of that *indexer-declaration*.
- The scope of a parameter declared in an *operator-declaration* (§10.10) is the *block* of that *operator-declaration*.
- The scope of a parameter declared in a *constructor-declaration* (§10.11) is the *constructor-initializer* and *block* of that *constructor-declaration*.
- The scope of a parameter declared in a *lambda-expression* (§7.14) is the *lambda-expression-body* of that *lambda-expression*.
- The scope of a parameter declared in an *anonymous-method-expression* (§7.14) is the *block* of that *anonymous-method-expression*.
- The scope of a label declared in a *labeled-statement* (§8.4) is the *block* in which the declaration occurs.
- The scope of a local variable declared in a *local-variable-declaration* (§8.5.1) is the block in which the declaration occurs.
- The scope of a local variable declared in a *switch-block* of a `switch` statement (§8.7.2) is the *switch-block*.
- The scope of a local variable declared in a *for-initializer* of a `for` statement (§8.8.3) is the *for-initializer*, the *for-condition*, the *for-iterator*, and the contained *statement* of the `for` statement.
- The scope of a local constant declared in a *local-constant-declaration* (§8.5.2) is the block in which the declaration occurs. It is a compile-time error to refer to a local constant in a textual position that precedes its *constant-declarator*.
- The scope of a variable declared as part of a *foreach-statement*, *using-statement*, *lock-statement*, or *query-expression* is determined by the expansion of the given construct.

Within the scope of a namespace, class, struct, or enumeration member, it is possible to refer to the member in a textual position that precedes the declaration of the member. For example, in

```
class A
{
    void F() {
        i = 1;
    }

    int i = 0;
}
```

it is valid for `F` to refer to `i` before it is declared.

Within the scope of a local variable, it is a compile-time error to refer to the local variable in a textual position that precedes the *local-variable-declarator* of the local variable. For example,

```
class A
{
    int i = 0;

    void F() {
        i = 1;                  // Error, use precedes declaration
        int i;
        i = 2;
    }

    void G() {
        int j = (j = 1);        // Valid
    }

    void H() {
        int a = 1, b = ++a;     // Valid
    }
}
```

In the preceding `F` method, the first assignment to `i` specifically does not refer to the field declared in the outer scope. Rather, it refers to the local variable and results in a compile-time error because it textually precedes the declaration of the variable. In the `G` method, the use of `j` in the initializer for the declaration of `j` is valid because the use does not precede the *local-variable-declarator*. In the `H` method, a subsequent *local-variable-declarator* correctly refers to a local variable declared in an earlier *local-variable-declarator* within the same *local-variable-declaration*.

The scoping rules for local variables are designed to guarantee that the meaning of a name used in an expression context is always the same within a block. If the scope of a local variable were to extend only from its declaration to the end of the block, then in the preceding

example, the first assignment would assign to the instance variable and the second assignment would assign to the local variable, possibly leading to compile-time errors if the statements of the block were later rearranged.

The meaning of a name within a block may differ based on the context in which the name is used. In the example

```
using System;

class A {}

class Test
{
    static void Main() {
        string A = "hello, world";
        string s = A;                   // Expression context

        Type t = typeof(A);             // Type context

        Console.WriteLine(s);           // Writes "hello, world"
        Console.WriteLine(t);           // Writes "A"
    }
}
```

the name `A` is used in an expression context to refer to the local variable `A` and in a type context to refer to the class `A`.

3.7.1 Name Hiding

The scope of an entity typically encompasses more program text than the declaration space of the entity. In particular, the scope of an entity may include declarations that introduce new declaration spaces containing entities of the same name. Such declarations cause the original entity to become ***hidden***. Conversely, an entity is said to be ***visible*** when it is not hidden.

Name hiding occurs when scopes overlap through nesting and when scopes overlap through inheritance. The characteristics of the two types of hiding are described in the following sections.

3.7.1.1 *Hiding through Nesting*

Name hiding through nesting can occur as a result of nesting namespaces or types within namespaces, as a result of nesting types within classes or structs, and as a result of parameter and local variable declarations.

In the example

```
class A
{
    int i = 0;
```

```
    void F() {
        int i = 1;
    }

    void G() {
        i = 1;
    }
}
```

within the `F` method, the instance variable `i` is hidden by the local variable `i`, but within the `G` method, `i` still refers to the instance variable.

When a name in an inner scope hides a name in an outer scope, it hides all overloaded occurrences of that name. In the example

```
class Outer
{
    static void F(int i) {}

    static void F(string s) {}

    class Inner
    {
        void G() {
            F(1);               // Invokes Outer.Inner.F
            F("Hello");         // Error
        }

        static void F(long l) {}
    }
}
```

the call `F(1)` invokes the `F` declared in `Inner` because all outer occurrences of `F` are hidden by the inner declaration. For the same reason, the call `F("Hello")` results in a compile-time error.

VLADIMIR RESHETNIKOV If a nested scope contains a member with the same name as a member from an outer scope, then the member from the outer scope is not always hidden due to the rule "If the member is *invoked*, all *non-invocable* members are removed from the set" (see §7.3):

```
class A {
  static void Foo() { }
  class B {
    const int Foo = 1;
    void Bar() {
      Foo(); // Okay
    }
  }
}
```

3.7.1.2 *Hiding through Inheritance*

Name hiding through inheritance occurs when classes or structs redeclare names that were inherited from base classes. This type of name hiding takes one of the following forms:

- A constant, field, property, event, or type introduced in a class or struct hides all base class members with the same name.
- A method introduced in a class or struct hides all nonmethod base class members with the same name, and all base class methods with the same signature (method name and parameter count, modifiers, and types).
- An indexer introduced in a class or struct hides all base class indexers with the same signature (parameter count and types).

The rules governing operator declarations (§10.10) make it impossible for a derived class to declare an operator with the same signature as an operator in a base class. Thus, operators never hide one another.

Unlike hiding a name from an outer scope, hiding an accessible name from an inherited scope causes a warning to be reported. In the example

```
class Base
{
    public void F() {}
}

class Derived: Base
{
    public void F() {}        // Warning, hiding an inherited name
}
```

the declaration of `F` in `Derived` causes a warning to be reported. Hiding an inherited name is specifically not an error, because that would preclude separate evolution of base classes. For example, the preceding situation might have come about because a later version of `Base` introduced an `F` method that wasn't present in an earlier version of the class. If this situation had been an error, then *any* change made to a base class in a separately versioned class library could potentially cause derived classes to become invalid.

The warning caused by hiding an inherited name can be eliminated through use of the `new` modifier:

```
class Base
{
    public void F() {}
}

class Derived: Base
{
    new public void F() {}
}
```

The `new` modifier indicates that the `F` in `Derived` is "new," and that it is, indeed, intended to hide the inherited member.

A declaration of a new member hides an inherited member only within the scope of the new member.

```
class Base
{
    public static void F() {}
}

class Derived: Base
{
    new private static void F() {}    // Hides Base.F in Derived only
}

class MoreDerived: Derived
{
    static void G() { F(); }          // Invokes Base.F
}
```

In the preceding example, the declaration of `F` in `Derived` hides the `F` that was inherited from `Base`. Because the new `F` in `Derived` has private access, however, its scope does not extend to `MoreDerived`. Thus, the call `F()` in `MoreDerived.G` is valid and will invoke `Base.F`.

CHRIS SELLS If you find yourself using `new` to hide an instance method on the base class, you're almost always going to be disappointed, if for no other reason than a caller can simply cast to the base class to get to the "hidden" method. For example,

```
class Base { public void F() {} }
class Derived : Base { new public void F() {} }
Derived d = new Derived();
((Base)d).F(); // Base.F not so hidden as you'd like...
```

You'll be much happier if you pick a new name for the method in the derived class instead.

3.8 Namespace and Type Names

Several contexts in a C# program require a *namespace-name* or a *type-name* to be specified.

namespace-name:
    namespace-or-type-name

type-name:
    namespace-or-type-name

namespace-or-type-name:
identifier *type-argument-list*$_{opt}$
namespace-or-type-name **.** *identifier* *type-argument-list*$_{opt}$
qualified-alias-member

A *namespace-name* is a *namespace-or-type-name* that refers to a namespace. Following resolution as described later in this section, the *namespace-or-type-name* of a *namespace-name* must refer to a namespace; otherwise, a compile-time error occurs. No type arguments (§4.4.1) can be present in a *namespace-name* (only types can have type arguments).

A *type-name* is a *namespace-or-type-name* that refers to a type. Following resolution as described later in this section, the *namespace-or-type-name* of a *type-name* must refer to a type; otherwise, a compile-time error occurs.

If the *namespace-or-type-name* is a qualified-alias-member, its meaning is as described in §9.7. Otherwise, a *namespace-or-type-name* has one of four forms:

- `I`
- `I<`A_1`, ..., `A_K`>`
- `N.I`
- `N.I<`A_1`, ..., `A_K`>`

where `I` is a single identifier, `N` is a *namespace-or-type-name*, and `<`A_1`, ..., `A_K`>` is an optional *type-argument-list*. When no *type-argument-list* is specified, consider `K` to be zero.

The meaning of a *namespace-or-type-name* is determined as follows:

- If the *namespace-or-type-name* is of the form `I` or of the form `I<`A_1`, ..., `A_K`>`:
  - If `K` is zero and the *namespace-or-type-name* appears within a generic method declaration (§10.6) and if that declaration includes a type parameter (§10.1.3) with name `I`, then the *namespace-or-type-name* refers to that type parameter.
  - Otherwise, if the *namespace-or-type-name* appears within a type declaration, then for each instance type `T` (§10.3.1), starting with the instance type of that type declaration and continuing with the instance type of each enclosing class or struct declaration (if any):
    - If `K` is zero and the declaration of `T` includes a type parameter with name `I`, then the *namespace-or-type-name* refers to that type parameter.
    - Otherwise, if the *namespace-or-type-name* appears within the body of the type declaration, and `T` or any of its base types contain a nested accessible type having name `I` and `K` type parameters, then the *namespace-or-type-name* refers to that type

constructed with the given type arguments. If there is more than one such type, the type declared within the more derived type is selected. Note that nontype members (constants, fields, methods, properties, indexers, operators, instance constructors, destructors, and static constructors) and type members with a different number of type parameters are ignored when determining the meaning of the *namespace-or-type-name*.

- If the previous steps were unsuccessful, then, for each namespace `N`, starting with the namespace in which the *namespace-or-type-name* occurs, continuing with each enclosing namespace (if any), and ending with the global namespace, the following steps are evaluated until an entity is located:

  - If `K` is zero and `I` is the name of a namespace in `N`, then
    - If the location where the *namespace-or-type-name* occurs is enclosed by a namespace declaration for `N` and the namespace declaration contains an *extern-alias-directive* or *using-alias-directive* that associates the name `I` with a namespace or type, then the *namespace-or-type-name* is ambiguous and a compile-time error occurs.
    - Otherwise, the *namespace-or-type-name* refers to the namespace named `I` in `N`.
  - Otherwise, if `N` contains an accessible type having name `I` and `K` type parameters, then
    - If `K` is zero and the location where the *namespace-or-type-name* occurs is enclosed by a namespace declaration for `N` and the namespace declaration contains an *extern-alias-directive* or *using-alias-directive* that associates the name `I` with a namespace or type, then the *namespace-or-type-name* is ambiguous and a compile-time error occurs.
    - Otherwise, the *namespace-or-type-name* refers to the type constructed with the given type arguments.
  - Otherwise, if the location where the *namespace-or-type-name* occurs is enclosed by a namespace declaration for `N`:
    - If `K` is zero and the namespace declaration contains an *extern-alias-directive* or *using-alias-directive* that associates the name `I` with an imported namespace or type, then the *namespace-or-type-name* refers to that namespace or type.
    - Otherwise, if the namespaces imported by the *using-namespace-directive*s of the namespace declaration contain exactly one type having name `I` and `K` type parameters, then the *namespace-or-type-name* refers to that type constructed with the given type arguments.

  - Otherwise, if the namespaces imported by the *using-namespace-directive*s of the namespace declaration contain more than one type having name `I` and `K` type parameters, then the *namespace-or-type-name* is ambiguous and an error occurs.
  - Otherwise, the *namespace-or-type-name* is undefined and a compile-time error occurs.

- Otherwise, the *namespace-or-type-name* is of the form `N.I` or of the form `N.I<`A_1`, …, `A_K`>`. `N` is first resolved as a *namespace-or-type-name*. If the resolution of `N` is not successful, a compile-time error occurs. Otherwise, `N.I` or `N.I<`A_1`, …, `A_K`>` is resolved as follows:
  - If `K` is zero and `N` refers to a namespace and `N` contains a nested namespace with name `I`, then the *namespace-or-type-name* refers to that nested namespace.
  - Otherwise, if `N` refers to a namespace and `N` contains an accessible type having name `I` and `K` type parameters, then the *namespace-or-type-name* refers to that type constructed with the given type arguments.
  - Otherwise, if `N` refers to a (possibly constructed) class or struct type and `N` or any of its base classes contain a nested accessible type having name `I` and `K` type parameters, then the *namespace-or-type-name* refers to that type constructed with the given type arguments. If there is more than one such type, the type declared within the more derived type is selected. Note that if the meaning of `N.I` is being determined as part of resolving the base class specification of `N` then the direct base class of `N` is considered to be object (§10.1.4.1).
  - Otherwise, `N.I` is an invalid *namespace-or-type-name*, and a compile-time error occurs.

A *namespace-or-type-name* is permitted to reference a static class (§10.1.1.3) only if

- The *namespace-or-type-name* is the `T` in a *namespace-or-type-name* of the form `T.I`, or
- The *namespace-or-type-name* is the `T` in a *typeof-expression* (§7.5.11) of the form `typeof(T)`.

3.8.1 Fully Qualified Names

Every namespace and type has a ***fully qualified name***, which uniquely identifies the namespace or type amongst all others. The fully qualified name of a namespace or type `N` is determined as follows:

- If `N` is a member of the global namespace, its fully qualified name is `N`.
- Otherwise, its fully qualified name is `S.N`, where `S` is the fully qualified name of the namespace or type in which `N` is declared.

In other words, the fully qualified name of `N` is the complete hierarchical path of identifiers that lead to `N`, starting from the global namespace. Because every member of a namespace or type must have a unique name, it follows that the fully qualified name of a namespace or type is always unique.

The following example shows several namespace and type declarations along with their associated fully qualified names.

```
class A {}                // A

namespace X               // X
{
    class B               // X.B
    {
        class C {}        // X.B.C
    }

    namespace Y           // X.Y
    {
        class D {}        // X.Y.D
    }
}

namespace X.Y             // X.Y
{
    class E {}            // X.Y.E
}
```


JOSEPH ALBAHARI If a fully qualified name conflicts with a partially qualified or unqualified name (a nested accessible type, for instance), the latter wins. Prefixing the name with `global::` forces the fully qualified name to win (§9.7). There's little chance of such a collision in human-written code, but with machine-written code, the odds are greater. For this reason, some code generators in design tools and IDEs emit the `global::` prefix before all fully qualified type names to eliminate any possibility of conflict.

3.9 Automatic Memory Management

C# employs automatic memory management, which frees developers from manually allocating and freeing the memory occupied by objects. Automatic memory management policies are implemented by a ***garbage collector***. The memory management life cycle of an object is as follows:

1. When the object is created, memory is allocated for it, the constructor is run, and the object is considered live.

2. If the object, or any part of it, cannot be accessed by any possible continuation of execution, other than the running of destructors, the object is considered no longer in use, and it becomes eligible for destruction. The C# compiler and the garbage collector may choose to analyze code to determine which references to an object may be used in the future. For instance, if a local variable that is in scope is the only existing reference to an object, but that local variable is never referred to in any possible continuation of execution from the current execution point in the procedure, the garbage collector may (but is not required to) treat the object as no longer in use.
3. Once the object is eligible for destruction, at some unspecified later time the destructor (§10.13) (if any) for the object is run. Unless overridden by explicit calls, the destructor for the object is run only once.
4. Once the destructor for an object is run, if that object, or any part of it, cannot be accessed by any possible continuation of execution, including the running of destructors, the object is considered inaccessible and the object becomes eligible for collection.
5. At some time after the object becomes eligible for collection, the garbage collector frees the memory associated with that object.

The garbage collector maintains information about object usage. It uses this information to make memory management decisions, such as where in memory to locate a newly created object, when to relocate an object, and when an object is no longer in use or inaccessible.

Like other languages that assume the existence of a garbage collector, C# is designed so that the garbage collector may implement a wide range of memory management policies. For instance, C# does not require that destructors be run or that objects be collected as soon as they are eligible, or that destructors be run in any particular order, or on any particular thread.

The behavior of the garbage collector can be controlled, to some degree, via static methods on the class `System.GC`. This class can be used to request a collection to occur, destructors to be run (or not run), and so forth.

ERIC LIPPERT Using these static methods to control the behavior of the garbage collector is almost never a good idea. In production code, odds are good that the garbage collector knows more about when would be a good time to do a collection than your program does.

Explicit tweaking of the garbage collector behavior at runtime should typically be limited to purposes such as forcing a collection for testing purposes.

Because the garbage collector is allowed wide latitude in deciding when to collect objects and run destructors, a conforming implementation may produce output that differs from that shown by the following code. The program

```
using System;

class A
{
    ~A() {
        Console.WriteLine("Destruct instance of A");
    }
}

class B
{
    object Ref;

    public B(object o) {
        Ref = o;
    }

    ~B() {
        Console.WriteLine("Destruct instance of B");
    }
}

class Test
{
    static void Main() {
        B b = new B(new A());
        b = null;
        GC.Collect();
        GC.WaitForPendingFinalizers();
    }
}
```

creates an instance of class `A` and an instance of class `B`. These objects become eligible for garbage collection when the variable `b` is assigned the value `null`, because after this time it is impossible for any user-written code to access them. The output could be either

```
Destruct instance of A
Destruct instance of B
```

or

```
Destruct instance of B
Destruct instance of A
```

because the language imposes no constraints on the order in which objects are garbage collected.

In subtle cases, the distinction between "eligible for destruction" and "eligible for collection" can be important. For example,

```
using System;

class A
{
    ~A() {
        Console.WriteLine("Destruct instance of A");
    }

    public void F() {
        Console.WriteLine("A.F");
        Test.RefA = this;
    }
}

class B
{
    public A Ref;

    ~B() {
        Console.WriteLine("Destruct instance of B");
        Ref.F();
    }
}

class Test
{
    public static A RefA;
    public static B RefB;

    static void Main() {
        RefB = new B();
        RefA = new A();
        RefB.Ref = RefA;
        RefB = null;
        RefA = null;

        // A and B now eligible for destruction
        GC.Collect();
        GC.WaitForPendingFinalizers();

        // B now eligible for collection, but A is not
        if (RefA != null)
            Console.WriteLine("RefA is not null");
    }
}
```

In this program, if the garbage collector chooses to run the destructor of `A` before the destructor of `B`, then the output of this program might be

```
Destruct instance of A
Destruct instance of B
A.F
RefA is not null
```

Although the instance of `A` was not in use and `A`'s destructor was run, it is still possible for methods of `A` (in this case, `F`) to be called from another destructor. Also, running of a destructor may cause an object to become usable from the mainline program again. In this case, the running of `B`'s destructor caused an instance of `A` that was previously not in use to become accessible from the live reference `Test.RefA`. After the call to `WaitForPendingFinalizers`, the instance of `B` is eligible for collection, but the instance of `A` is not, because of the reference `Test.RefA`.

To avoid confusion and unexpected behavior, it is generally a good idea for destructors to perform cleanup only on data stored in their object's own fields, and not to perform any actions on referenced objects or static fields.

ERIC LIPPERT It is an even better idea for the destructor to clean up only their fields that contain data representing unmanaged objects, such as operating system handles. Because you do not know which thread the destructor will run on or when it will run, it is particularly important that the destructor have as few side effects as possible.

The calls to `Console.WriteLine` in the example are obviously in violation of this good advice to do only cleanup and not perform other actions. This code is intended solely as a pedagogic aid. Real production code destructors should never attempt to do anything that has a complex side effect such as console output.

An alternative to using destructors is to let a class implement the `System.IDisposable` interface. This strategy allows the client of the object to determine when to release the resources of the object, typically by accessing the object as a resource in a `using` statement (§8.13).

BRAD ABRAMS Nine times out of ten, using `GC.Collect()` is a mistake. It is often an indication of a poor design that is being bandaided together. The garbage collector is a finely tuned instrument, like a Porsche. Just as you would not play bumper tag with a new Porsche, so you should generally avoid interfering with the garbage collector's algorithms. The garbage collector is designed to unobtrusively step in at the right time and collect the most important unused memory. Kicking it with the call `GC.Collect()` can throw off the balance and tuning. Before resorting to this solution, take a few minutes to figure why it is required. Have you disposed of all your instances? Have you dropped references where you could? Have you used weak references in the right places?

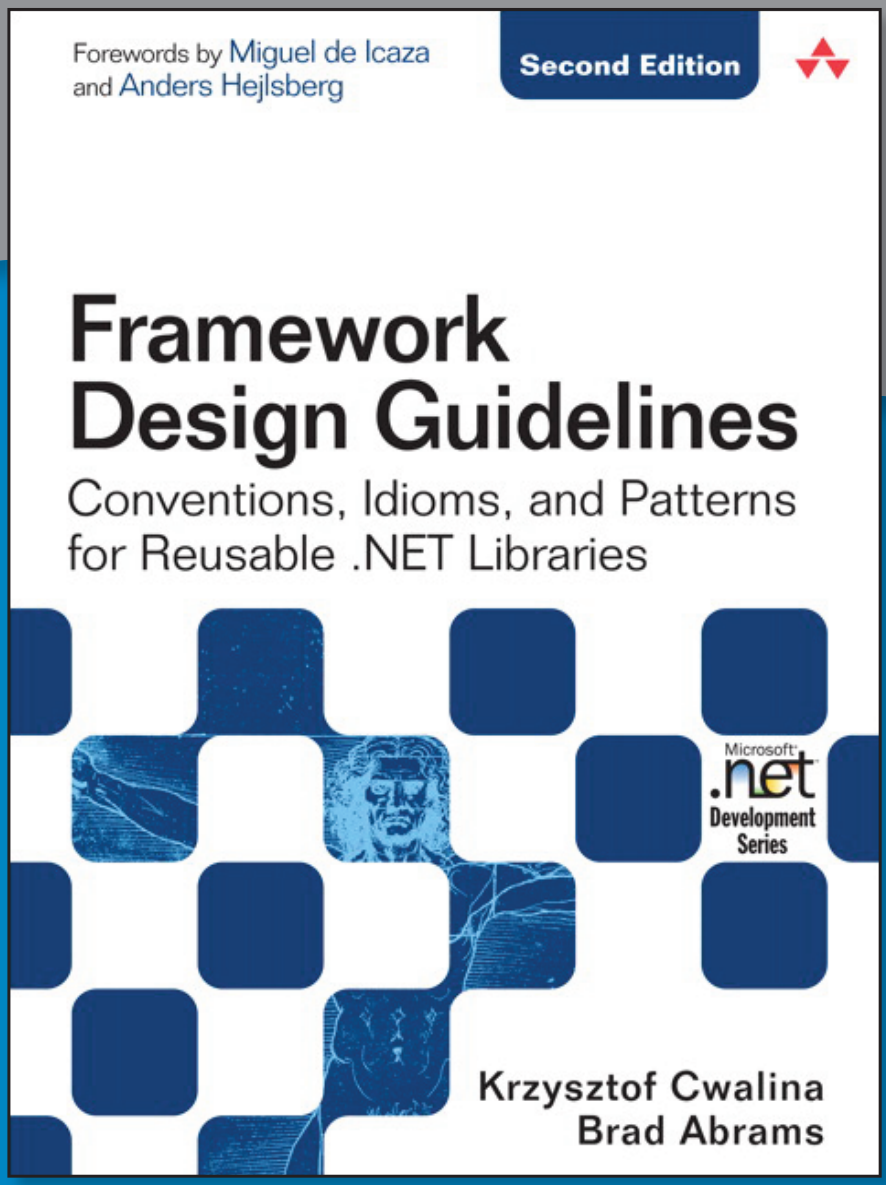
KRZYSZTOF CWALINA Some people attribute the performance problems of modern VM-based systems to their use of garbage collection. The fact is that modern garbage collectors are so efficient that there really isn't much software left that would have problems with the garbage collector's performance in itself. The biggest performance culprit is over-engineering of applications and, sadly, many framework libraries.

BILL WAGNER These notes point out how few of your regular assumptions are valid in the context of a destructor. Member variables may have already executed their destructors. They are called on a different thread, so thread local storage may not be valid. They are called by the system, so your application won't see errors reported by destructors using exceptions. It's hard to over-emphasize how defensively you need to write destructors. Luckily, they are needed only rarely.

3.10 Execution Order

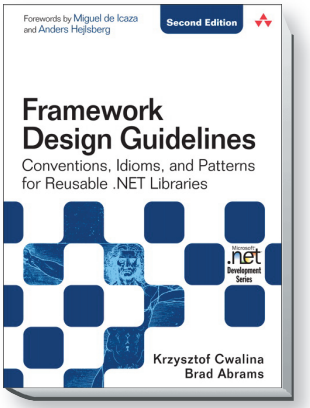
Execution of a C# program proceeds such that the side effects of each executing thread are preserved at critical execution points. A ***side effect*** is defined as a read or write of a volatile field, a write to a nonvolatile variable, a write to an external resource, and the throwing of an exception. The critical execution points at which the order of these side effects must be preserved are references to volatile fields (§10.5.3), `lock` statements (§8.12), and thread creation and termination. The execution environment is free to change the order of execution of a C# program, subject to the following constraints:

- Data dependence is preserved within a thread of execution. That is, the value of each variable is computed as if all statements in the thread were executed in original program order.
- Initialization ordering rules are preserved (§10.5.4 and §10.5.5).
- The ordering of side effects is preserved with respect to volatile reads and writes (§10.5.3). Additionally, the execution environment need not evaluate part of an expression if it can deduce that the expression's value is not used and that no needed side effects are produced (including any caused by calling a method or accessing a volatile field). When program execution is interrupted by an asynchronous event (such as an exception thrown by another thread), it is not guaranteed that the observable side effects will be visible in the original program order.



Krzysztof Cwalina
Brad Abrams

BUY ME



Framework Design Guidelines

Conventions, Idioms, and Patterns for Reusable .NET Libraries

Framework Design Guidelines, Second Edition, teaches developers the best practices for designing reusable libraries for the Microsoft .NET Framework. Expanded and updated for .NET 3.5, this new edition focuses on the design issues that directly affect the programmability of a class library, specifically its publicly accessible APIs.

This book can improve the work of any .NET developer producing code that other developers will use. It includes copious annotations to the guidelines by thirty-five prominent architects and practitioners of the .NET Framework, providing a lively discussion of the reasons for the guidelines as well as examples of when to break those guidelines.

Microsoft architects Krzysztof Cwalina and Brad Abrams teach framework design from the top down. From their significant combined experience and deep insight, you will learn

- The general philosophy and fundamental principles of framework design
- Naming guidelines for the various parts of a framework
- Guidelines for the design and extending of types and members of types
- Issues affecting–and guidelines for ensuring–extensibility
- How (and how not) to design exceptions
- Guidelines for–and examples of–common framework design patterns

Guidelines in this book are presented in four major forms: Do, Consider, Avoid, and Do not. These directives help focus attention on practices that should always be used, those that should generally be used, those that should rarely be used, and those that should never be used. Every guideline includes a discussion of its applicability, and most include a code example to help illuminate the dialogue.

Framework Design Guidelines, Second Edition, is the only definitive source of best practices for managed code API development, direct from the architects themselves.

A companion DVD includes the Designing .NET Class Libraries video series, instructional presentations by the authors on design guidelines for developing classes and components that extend the .NET Framework. A sample API specification and other useful resources and tools are also included.

AVAILABLE

- BOOK: 9780321545619
- SAFARI ONLINE
- EBOOK: 0321605012
- KINDLE: 0321605004

About the Authors

Krzysztof Cwalina is a program manager on the .NET Framework team at Microsoft. He was a founding member of the .NET Framework team and throughout his career has designed many .NET Framework APIs and framework development tools, such as FxCop. He is currently leading a companywide effort to develop, promote, and apply framework design and architectural guidelines to the .NET Framework. He is also leading the team responsible for delivering core .NET Framework APIs. Krzysztof graduated with a B.S. and an M.S. in computer science from the University of Iowa. You can find his blog at http://blogs.msdn.com/kcwalina.

Brad Abrams was a founding member of the Common Language Runtime and .NET Framework teams at Microsoft Corporation. He has been designing parts of the .NET Framework since 1998 and is currently Group Program Manager of the .NET Framework team. Brad started his framework design career building the Base Class Library (BCL) that ships as a core part of the .NET Framework. Brad was also the lead editor on the Common Language Specification (CLS), the .NET Framework Design Guidelines, and the libraries in the ECMA\ISO CLI Standard. Brad has authored and coauthored multiple publications, including Programming in the .NET Environment and .NET Framework Standard Library Annotated Reference, Volumes 1 and 2. Brad graduated from North Carolina State University with a B.S. in computer science. You can find his most recent musings on his blog at http://blogs.msdn.com/BradA.

Addison Wesley

informit.com/aw

3
Naming Guidelines

Following a consistent set of naming conventions in the development of a framework can be a major contribution to the framework's usability. It allows the framework to be used by many developers on widely separated projects. Beyond consistency of form, names of framework elements must be easily understood and must convey the function of each element.

The goal of this chapter is to provide a consistent set of naming conventions that results in names that make immediate sense to developers.

Most of the naming guidelines are simply conventions that have no technical rationale. However, following these naming guidelines will ensure that the names are understandable and consistent.

Although adopting these naming conventions as general code development guidelines would result in more consistent naming throughout your code, you are required only to apply them to APIs that are publicly exposed (public or protected types and members, and explicitly implemented interfaces).

KRZYSZTOF CWALINA The team that develops the .NET Framework Base Class Library spends an enormous amount of time on naming and considers it to be a crucial part of framework development.

This chapter describes general naming guidelines, including how to use capitalization, mechanics, and certain specific terms. It also provides specific guidelines for naming namespaces, types, members, parameters, assemblies, and resources.

3.1 Capitalization Conventions

Because the CLR supports many languages that might or might not be case sensitive, case alone should not be used to differentiate names. However, the importance of case in enhancing the readability of names cannot be overemphasized. The guidelines in this chapter lay out a simple method for using case that, when applied consistently, make identifiers for types, members, and parameters easy to read.

3.1.1 Capitalization Rules for Identifiers

To differentiate words in an identifier, capitalize the first letter of each word in the identifier. Do not use underscores to differentiate words, or for that matter, anywhere in identifiers. There are two appropriate ways to capitalize identifiers, depending on the use of the identifier:

- PascalCasing
- camelCasing

BRAD ABRAMS In the initial design of the Framework, we had hundreds of hours of debate about the naming style. To facilitate these debates, we coined a number of terms. With Anders Hejlsberg, the original designer of Turbo Pascal, and a key member of the design team, it is no wonder that we chose the term PascalCasing for the casing style popularized by the Pascal programming language. We were somewhat cute in using the term camelCasing for the casing style that looks something like the hump on a camel. We used the term SCREAMING_CAPS to indicate an all-uppercase style. Luckily, this style (and name) did not survive in the final guideline.

The PascalCasing convention, used for all identifiers except parameter names, capitalizes the first character of each word (including acronyms over two letters in length), as shown in the following examples:

```
PropertyDescriptor
HtmlTag
```

A special case is made for two-letter acronyms in which both letters are capitalized, as shown in the following identifier:

```
IOStream
```

The camelCasing convention, used only for parameter names, capitalizes the first character of each word except the first word, as shown in the following examples. As the example also shows, two-letter acronyms that begin a camel-cased identifier are both lowercase.

```
propertyDescriptor
ioStream
htmlTag
```

The following are two basic capitalization guidelines for identifiers:

✓ **DO** use PascalCasing for namespace, type, and member names consisting of multiple words.

For example, use `TextColor` rather than `Textcolor` or `Text_color`. Single words, such as `Button`, simply have initial capitals. Compound words that are always written as a single word, like endpoint, are treated as single words and have initial capitals only. More information on compound words is given in section 3.1.3.

✓ **DO** use camelCasing for parameter names.

Table 3-1 describes the capitalization rules for different types of identifiers.

BRAD ABRAMS An early version of this table included a convention for instance field names. We later adopted the guideline that you should almost never use publicly exposed instance fields and should use properties instead. Thus, the guideline for publicly exposed instance fields was no longer needed. For the record, the convention was camelCasing.

TABLE 3-1: Capitalization Rules for Different Types of Identifiers

Identifier	Casing	Example
Namespace	Pascal	`namespace System.Security { ... }`
Type	Pascal	`public class StreamReader { ... }`
Interface	Pascal	`public interface IEnumerable { ... }`

Method	Pascal	`public class Object {`
		`  public virtual string ToString();`
		`}`

Property	Pascal	`public class String {`
		`  public int Length { get; }`
		`}`

Event	Pascal	`public class Process {`
		`  public event EventHandler Exited;`
		`}`

Field	Pascal	`public class MessageQueue {`
		`  public static readonly TimeSpan InfiniteTimeout;`
		`}`
		`public struct UInt32 {`
		`  public const Min = 0;`
		`}`

Enum value	Pascal	`public enum FileMode {`
		`  Append,`
		`  ...`
		`}`

Parameter	Camel	`public class Convert {`
		`  public static int ToInt32(string value);`
		`}`

3.1.2 Capitalizing Acronyms

In general, it is important to avoid using acronyms in identifier names unless they are in common usage and are immediately understandable to anyone who might use the framework. For example, HTML, XML, and IO are all well understood, but less well-known acronyms should definitely be avoided.

KRZYSZTOF CWALINA Acronyms are distinct from abbreviations, which should never be used in identifiers. An acronym is a word made from the initial letters of a phrase, whereas an abbreviation simply shortens a word.

By definition, an acronym must be at least two characters. Acronyms of three or more characters follow the guidelines of any other word. Only the first letter is capitalized, unless it is the first word in a camel-cased parameter name, which is all lowercase.

As mentioned in the preceding section, two-character acronyms (e.g., IO) are treated differently, primarily to avoid confusion. Both characters should be capitalized unless the two-character acronym is the first word in a camel-cased parameter name, in which case both characters are lowercase. The following examples illustrate all of these cases:

```
public void StartIO(Stream ioStream, bool closeIOStream);
public void ProcessHtmlTag(string htmlTag)
```

✓ **DO** capitalize both characters of two-character acronyms, except the first word of a camel-cased identifier.

```
System.IO
public void StartIO(Stream ioStream)
```

✓ **DO** capitalize only the first character of acronyms with three or more characters, except the first word of a camel-cased identifier.

```
System.Xml
public void ProcessHtmlTag(string htmlTag)
```

✓ **DO NOT** capitalize any of the characters of any acronyms, whatever their length, at the beginning of a camel-cased identifier.

BRAD ABRAMS In my time working on the .NET Framework, I have heard every possible excuse for violating these naming guidelines. Many teams feel that they have some special reason to use case differently in their identifiers than in the rest of the Framework. These excuses include consistency with other platforms (MFC, HTML, etc.), avoiding geopolitical issues

(casing of some country names), honoring the dead (abbreviation names that came up with some crypto algorithm), and the list goes on and on. For the most part, our customers have seen the places in which we have diverged from these guidelines (for even the best excuse) as warts in the Framework. The only time I think it really makes sense to violate these guidelines is when using a trademark as an identifier. However, I suggest not using trademarks, because they tend to change faster than APIs do.

BRAD ABRAMS Here is an example of putting these naming guidelines to the test. We have the class shown here in the Framework today. It successfully follows the guidelines for casing and uses `Argb` rather than `ARGB`. But we have actually gotten bug reports along the lines of "How do you convert a color from an ARGB value—all I see are methods to convert 'from argument b.'?"

```
public struct Color  {
...
   public static Color FromArgb(int alpha, Color baseColor);
   public static Color FromArgb(int alpha, int red, int green, int blue);
   public static Color FromArgb(int argb);
   public static Color FromArgb(int red, int green, int blue);
...
}
```

In retrospect, should this have been a place where we violated the guidelines and used `FromARGB`? I do not think so. It turns out that this is a case of overabbreviation. RGB is a well-recognized acronym for red-green-blue values. An ARGB value is a relatively uncommon abbreviation that includes the alpha channel. It would have been clearer to name these `AlphaRgb` and would have been more consistent in naming with the rest of the Framework.

```
public struct Color  {
...
   public static Color FromAlphaRgb(int alpha, Color baseColor);
   public static Color FromAlphaRgb(int alpha, int red, int green, int blue);
   public static Color FromAlphaRgb(int argb);
   public static Color FromAlphaRgb(int red, int green, int blue);
...
}
```

3.1.3 Capitalizing Compound Words and Common Terms

Most compound terms are treated as single words for purposes of capitalization.

✗ **DO NOT** capitalize each word in so-called closed-form compound words.

These are compound words written as a single word, such as endpoint. For the purpose of casing guidelines, treat a closed-form compound word as a single word. Use a current dictionary to determine if a compound word is written in closed form.

Table 3-2 shows capitalization for some of the most commonly used compound words and common terms.

TABLE 3-2: Capitalization and Spelling for Common Compound Words and Common Terms

Pascal	Camel	Not
BitFlag	bitFlag	Bitflag
Callback	callback	CallBack
Canceled	canceled	Cancelled
DoNot	doNot	Don't
Email	email	EMail
Endpoint	endpoint	EndPoint
FileName	fileName	Filename
Gridline	gridline	GridLine
Hashtable	hashtable	HashTable
Id	id	ID
Indexes	indexes	Indices
LogOff	logOff	LogOut

Continues

TABLE 3.2: Continued

Pascal	Camel	Not
LogOn	logOn	LogIn
Metadata	metadata	MetaData, metaData
Multipanel	multipanel	MultiPanel
Multiview	multiview	MultiView
Namespace	namespace	NameSpace
Ok	ok	OK
Pi	pi	PI
Placeholder	placeholder	PlaceHolder
SignIn	signIn	SignOn
SignOut	signOut	SignOff
UserName	userName	Username
WhiteSpace	whiteSpace	Whitespace
Writable	writable	Writeable

Two other terms that are in common usage are in a category by themselves, because they are common slang abbreviations. The two words *Ok* and *Id* (and they should be cased as shown) are the exceptions to the guideline that no abbreviations should be used in names.

BRAD ABRAMS Table 3-2 presents specific examples found in the development of the .NET Framework. You might find it useful to create your own appendix to this table for compound words and other terms commonly used in your domain.

BRAD ABRAMS One abbreviation commonly used in COM interface names was Ex (for interfaces that were extended versions of previously existing interfaces). This abbreviation should be avoided in reusable libraries. Use instead a meaningful name that describes the new functionality. For example, rather than `IDispatchEx`, consider `IDynamicDispatch`.

3.1.4 Case Sensitivity

Languages that can run on the CLR are not required to support case sensitivity, although some do. Even if your language supports it, other languages that might access your framework do not. Any APIs that are externally accessible, therefore, cannot rely on case alone to distinguish between two names in the same context.

PAUL VICK When it came to the question of case sensitivity, there was no question in the minds of the Visual Basic team that the CLR had to support case insensitivity as well as case sensitivity. Visual Basic has been case insensitive for a very long time, and the shock of trying to move VB developers (including myself) into a case-sensitive world would have made any of the other challenges we faced pale in comparison. Add to that the fact that COM is case insensitive, and the matter seemed pretty clear. The CLR would have to take case insensitivity into account.

JEFFREY RICHTER To be clear, the CLR is actually case sensitive. Some programming languages, like Visual Basic, are case insensitive. When the VB compiler is trying to resolve a method call to a type defined in a case-sensitive language like C#, the compiler (not the CLR) figures out the actual case of the method's name and embeds it in metadata. The CLR knows nothing about this. Now if you are using reflection to bind to a method, the reflection APIs do offer the ability to do case-insensitive lookups. This is the extent to which the CLR supports case insensitivity.

There is really only one guideline for case sensitivity, albeit an important one.

✗ **DO NOT** assume that all programming languages are case sensitive. They are not. Names cannot differ by case alone.

3.2 General Naming Conventions

This section describes general naming conventions that relate to word choice, guidelines on using abbreviations and acronyms, and recommendations on how to avoid using language-specific names.

3.2.1 Word Choice

It is important that names of framework identifiers make sense on first reading. Identifier names should clearly state what each member does and what each type and parameter represents. To this end, it is more important that the name be clear than that it be short. Names should correspond to scenarios, logical or physical parts of the system, and well-known concepts rather than to technologies or architecture.

✓ **DO** choose easily readable identifier names.

For example, a property named `HorizontalAlignment` is more English-readable than `AlignmentHorizontal`.

✓ **DO** favor readability over brevity. The property name `CanScrollHorizontally` is better than `ScrollableX` (an obscure reference to the X-axis).

✗ **DO NOT** use underscores, hyphens, or any other non-alphanumeric characters.

✗ **DO NOT** use Hungarian notation.

> **BRENT RECTOR** It might be useful here to define what Hungarian notation is. It is the convention of prefixing a variable name with some lowercase encoding of its data type. For example, the variable `uiCount` would be an unsigned integer. Another common convention also adds a prefix indicating the scope of the variable in addition to or in place of the type (see Jeffrey's later example of static and member variable-scope prefixes).

One downside of Hungarian notation is that developers frequently change the type of variables during early coding, which requires the name of the variable to also change. Additionally, while commonly used fundamental data types (integers, characters, etc.) had well-recognized and standard prefixes, developers frequently fail to use a meaningful and consistent prefix for their custom data types.

KRZYSZTOF CWALINA There have always been both positive and negative effects of using the Hungarian naming convention, and they still exist today. Positives include better readability (if used correctly). Negatives include cost of maintenance, confusion if maintenance was not done properly, and finally, Hungarian makes the API more cryptic (less approachable) to some developers. In the world of procedural languages (e.g., C) and the separation of the System APIs for advanced developers from framework libraries for a much wider developer group, the positives seemed to be greater than the negatives. Today, with System APIs designed to be approachable to more developers, and with object-oriented languages, the trade-off seems to be pulling in the other direction. OO encapsulation brings variable declaration and usage points closer together, OO style favors short, well-factored methods, and abstractions often make the exact type less important or even meaningless.

JEFFREY RICHTER I'll admit it; I miss Hungarian notation. Although in many editors, like Visual Studio, you can hover the mouse over a variable and the editor pops up the type, this does not work when reading source code in a book chapter or magazine article. Fortunately, in OOP, variables tend have a short scope, so that you only need to scan a few lines to find the definition of a variable. However, this is not true for a type's static and instance fields. Personally, I make all my fields private, and I now prefix my instance fields with "m_" and my static fields with "s_" so that I can easily spot fields in my methods. Luckily, this does not conflict with the guidelines described in this chapter, because they only cover publicly exposed members. This helps me a lot, but I still can't tell what type a variable represents. I rely on my editor's tool tips for this.

ANTHONY MOORE While most coding guidelines in use at Microsoft do not promote the use of "m_" and "s_" described above, I believe that practice is worth consideration. For the most part, the consistency of use of a coding guideline is more important than its details. However, in this case I've seen a few bugs caused by confusion between local variables and member fields that this practice could prevent. Similarly, static fields are a dangerous construct, because for a library you generally need to make them—as well as any instances transitively hanging off them—threadsafe. Making them look different from regular members makes it easier to find thread-safety errors when reviewing code.

✗ **AVOID** using identifiers that conflict with keywords of widely used programming languages.

According to Rule 4 of the Common Language Specification (CLS), all compliant languages must provide a mechanism that allows access to named items that use a keyword of that language as an identifier. C#, for example, uses the @ sign as an escape mechanism in this case. However, it is still a good idea to avoid common keywords because it is much more difficult to use a method with the escape sequence than one without it.

JEFFREY RICHTER When I was porting my *Applied Microsoft .NET Framework Programming* book from C# to Visual Basic, I ran into this situation a lot. For example, the class library has `Delegate`, `Module`, and `Assembly` classes, and Visual Basic uses these same terms for keywords. This problem is exacerbated by the fact that VB is a case-insensitive language. Visual Basic, like C#, has a way to escape the keywords to disambiguate the situation to the compiler (using square brackets), but I was surprised that the VB team selected keywords that conflict with so many class library names.

3.2.2 Using Abbreviations and Acronyms

In general, do not use abbreviations or acronyms in identifiers. As stated earlier, it is more important for names to be readable than it is for them to

be brief. It is equally important not to use abbreviations and acronyms that are not generally understood—that is, do not use anything that the large majority of people who are not experts in a given field would not know the meaning of immediately.

✗ **DO NOT** use abbreviations or contractions as part of identifier names.

For example, use `GetWindow` rather than `GetWin`.

✗ **DO NOT** use any acronyms that are not widely accepted, and even if they are, only when necessary.

For example, UI is used for User Interface and HTML is used for Hyper Text Markup Language. Although many framework designers feel that some recent acronym will soon be widely accepted, it is bad practice to use it in framework identifiers.

For acronym capitalization rules, see section 3.1.2.

BRAD ABRAMS We continually debate about whether a given acronym is well known or not. A good divining rod is what I call the grep test. Simply use some search engine to grep the Web for the acronym. If the first few results returned are indeed the meaning you intend, it is likely that your acronym qualifies as well known; if you don't get those search results, think harder about the name. If you fail the test, don't just spell out the acronym but consider how you can be descriptive in the name.

3.2.3 Avoiding Language-Specific Names

Programming languages that target the CLR often have their own names (aliases) for the so-called primitive types. For example, `int` is a C# alias for `System.Int32`. To ensure that your framework can take full advantage of the cross-language interoperation that is one of the core features of the CLR, it is important to avoid the use of these language-specific type names in identifiers.

JEFFREY RICHTER Personally, I take this a step further and never use the language's alias names. I find that the alias adds nothing of value and introduces enormous confusion. For example, I'm frequently asked what the difference is between `String` and `string` in C#. I've even heard people say that strings (lowercase "S") are allocated on the stack while Strings (uppercase "S") are allocated on the heap. In my book *CLR via C#*, I give several reasons in addition to the one offered here for avoiding the alias names. Another example of a class library/language mismatch is the `NullReferenceException` class, which can be thrown by VB code. But VB uses `Nothing`, not `null`.

✓ **DO** use semantically interesting names rather than language-specific keywords for type names.

For example, `GetLength` is a better name than `GetInt`.

✓ **DO** use a generic CLR type name, rather than a language-specific name, in the rare cases when an identifier has no semantic meaning beyond its type.

For example, a method converting to `System.Int64` should be named `ToInt64`, not `ToLong` (because `System.Int64` is a CLR name for the C#-specific alias `long`). Table 3-3 presents several base data types using the CLR type names (as well as the corresponding type names for C#, Visual Basic, and C++).

TABLE 3-3: CLR Type Names for Language-Specific Type Names

C#	Visual Basic	C++	CLR
`sbyte`	`SByte`	`char`	`SByte`
`byte`	`Byte`	`unsigned char`	`Byte`
`short`	`Short`	`short`	`Int16`
`ushort`	`UInt16`	`unsigned short`	`UInt16`
`int`	`Integer`	`int`	`Int32`

C#	Visual Basic	C++	CLR
uint	UInt32	unsigned int	UInt32
long	Long	__int64	Int64
ulong	UInt64	unsigned __int64	UInt64
float	Single	float	Single
double	Double	double	Double
bool	Boolean	bool	Boolean
char	Char	wchar_t	Char
string	String	String	String
object	Object	Object	Object

✓ **DO** use a common name, such as *value* or *item*, rather than repeating the type name, in the rare cases when an identifier has no semantic meaning and the type of the parameter is not important.

The following is a good example of methods of a class that supports writing a variety of data types into a stream:

```
void Write(double value);
void Write(float value);
void Write(short value);
```

3.2.4 Naming New Versions of Existing APIs

Sometimes a new feature cannot be added to an existing type even though the type's name implies that it is the best place for the new feature. In such a case, a new type needs to be added, which often leaves the framework designer with the difficult task of finding a good new name for the new type. Similarly, an existing member often cannot be extended or overloaded to provide additional functionality, and so a member with a new name needs to be added. The guidelines that follow describe how to choose

names for new types and members that supersede or replace existing types or members.

✓ **DO** use a name similar to the old API when creating new versions of an existing API.

This helps to highlight the relationship between the APIs.

```
class AppDomain {
   [Obsolete("AppDomain.SetCachePath has been deprecated. Please use
AppDomainSetup.CachePath instead.")]
   public void SetCachePath(String path) { ... }
}

class AppDomainSetup {
   public string CachePath { get { ... }; set  { ... }; }
}
```

✓ **DO** prefer adding a suffix rather than a prefix to indicate a new version of an existing API.

> **VANCE MORRISON** We did exactly this when we added a faster (but not completely backward-compatible) version of `ReaderWriterLock`. We called it `ReaderWriterLockSlim`. There was debate whether we should call it `SlimReaderWriterLock` (following the guideline that you write it like you say it in English), but decided the discoverability (and the fact that lexical sorting would put them close to each other) was more important.

This will assist discovery when browsing documentation, or using Intellisense. The old version of the API will be organized close to the new APIs, because most browsers and Intellisense show identifiers in alphabetical order.

✓ **CONSIDER** using a brand new, but meaningful identifier, instead of adding a suffix or a prefix.

✓ **DO** use a numeric suffix to indicate a new version of an existing API, particularly if the existing name of the API is the only name that makes sense (i.e., if it is an industry standard) and if adding any meaningful suffix (or changing the name) is not an appropriate option.

```
// old API
[Obsolete("This type is obsolete. Please use the new version of the same
class, X509Certificate2.")]
public class X509Certificate { ... }
// new API
public class X509Certificate2 { ... }
```

KRZYSZTOF CWALINA I would use numeric suffixes as the very last resort. A much better approach is to use a new name or a meaningful suffix.

The BCL team shipped a new type named `TimeZone2` in one of the early prereleases of the .NET Framework 3.5. The name immediately became the center of a controversy in the blogging community. After a set of lengthy discussions, the team decided to rename the type to `TimeZoneInfo`, which is not a great name but is much better than `TimeZone2`.

It's interesting to note that nobody dislikes X509Certificate2. My interpretation of this fact is that programmers are more willing to accept the ugly numeric suffix on rarely used library types somewhere in the corner of the Framework than on a core type in the System namespace.

✗ **DO NOT** use the "Ex" (or a similar) suffix for an identifier to distinguish it from an earlier version of the same API.

```
[Obsolete("This type is obsolete. ...")]
public class Car  { ... }

// new API
public class CarEx      { ... }    // the wrong way
public class CarNew     { ... }    // the wrong way
public class Car2       { ... }    // the right way
public class Automobile { ... }    // the right way
```

✓ **DO** use the "64" suffix when introducing versions of APIs that operate on a 64-bit integer (a long integer) instead of a 32-bit integer. You only need to take this approach when the existing 32-bit API exists; don't do it for brand new APIs with only a 64-bit version.

For example, various APIs on `System.Diagnostics.Process` return `Int32` values representing memory sizes, such as `PagedMemorySize` or

PeakWorkingSet. To appropriately support these APIs on 64-bit systems, APIs have been added that have the same name but a "64" suffix.

```
public class Process {
   // old APIs
   public int PeakWorkingSet { get; }
   public int PagedMemorySize { get; }
   // ...
   // new APIs
   public long PeakWorkingSet64 { get; }
   public long PagedMemorySize64 { get; }
}
```

KIT GEORGE Note that this guideline applies only to retrofitting APIs that have already shipped. When designing a brand new API, use the most appropriate type and name for the API that will work on all platforms, and avoid using both "32" and "64" suffixes. Consider using overloading.

3.3 Names of Assemblies and DLLs

An assembly is the unit of deployment and identity for managed code programs. Although assemblies can span one or more files, typically an assembly maps one-to-one with a DLL. Therefore, this section describes only DLL naming conventions, which then can be mapped to assembly naming conventions.

JEFFREY RICHTER Multifile assemblies are rarely used, and Visual Studio has no built-in support for them.

Keep in mind that namespaces are distinct from DLL and assembly names. Namespaces represent logical groupings for developers, whereas DLLs and assemblies represent packaging and deployment boundaries. DLLs can contain multiple namespaces for product factoring and other reasons. Because namespace factoring is different than DLL factoring, you should design them independently. For example, if you decide to name your DLL MyCompany.MyTechnology, it does not mean that the DLL has to contain a namespace named MyCompany.MyTechnology, though it can.

JEFFREY RICHTER Programmers are frequently confused by the fact that the CLR does not enforce a relationship between namespaces and assembly file names. For example, `System.IO.FileStream` is in `MSCorLib.dll`, and `System.IO.FileSystemWatcher` is in `System.dll`. As you can see, types in a single namespace can span multiple files. Also notice that the .NET Framework doesn't ship with a `System.IO.dll` file at all.

BRAD ABRAMS We decided early in the design of the CLR to separate the developer view of the platform (namespaces) from the packaging and deployment view of the platform (assemblies). This separation allows each to be optimized independently based on its own criteria. For example, we are free to factor namespaces to group types that are functionally related (e.g., all the I/O stuff is in `System.IO`), but the assemblies can be factored for performance (load time), deployment, servicing, or versioning reasons.

✓ **DO** choose names for your assembly DLLs that suggest large chunks of functionality, such as `System.Data`.

Assembly and DLL names don't have to correspond to namespace names, but it is reasonable to follow the namespace name when naming assemblies. A good rule of thumb is to name the DLL based on the common prefix of the assemblies contained in the assembly. For example, an assembly with two namespaces, `MyCompany.MyTechnology.FirstFeature` and `MyCompany.MyTechnology.SecondFeature`, could be called `MyCompany.MyTechnology.dll`.

✓ **CONSIDER** naming DLLs according to the following pattern:

```
<Company>.<Component>.dll
```

where `<Component>` contains one or more dot-separated clauses. For example:

```
Microsoft.VisualBasic.dll
Microsoft.VisualBasic.Vsa.dll
Fabrikam.Security.dll
Litware.Controls.dll
```

3.4 Names of Namespaces

As with other naming guidelines, the goal when naming namespaces is creating sufficient clarity for the programmer using the framework to immediately know what the content of the namespace is likely to be. The following template specifies the general rule for naming namespaces:

```
<Company>.(<Product>|<Technology>)[.<Feature>][.<Subnamespace>]
```

The following are examples:

```
Microsoft.VisualStudio
Microsoft.VisualStudio.Design
Fabrikam.Math
Litware.Security
```

✓ **DO** prefix namespace names with a company name to prevent namespaces from different companies from having the same name.

For example, the Microsoft Office automation APIs provided by Microsoft should be in the namespace `Microsoft.Office`.

> **BRAD ABRAMS** It is important to use the official name of your company or organization when choosing the first part of your namespace name to avoid possible conflicts. For example, if Microsoft had chosen to use MS as its root namespace, it might have been confusing to developers at other companies that use MS as an abbreviation.

✓ **DO** use a stable, version-independent product name at the second level of a namespace name.

> **BRAD ABRAMS** This means staying away from the latest cool and catchy name that the marketing folks have come up with. It is fine to tweak the branding of a product from release to release, but the namespace name is going to be burned into your client's code forever. Therefore, choose something that is technically sound and not subject to the marketing whims of the day.

✗ **DO NOT** use organizational hierarchies as the basis for names in namespace hierarchies, because group names within corporations tend to be short-lived. Organize the hierarchy of namespaces around groups of related technologies.

> **BRAD ABRAMS** We added a set of controls to ASP.NET late in the ship cycle for V1.0 of the .NET Framework that rendered for mobile devices. Because these controls came from a team in a different division, our immediate reaction was to put them in a different namespace (`System.Web.MobileControls`). Then, after a couple of reorganizations and .NET Framework versions, we realized a better engineering trade-off was to fold that functionality into the existing controls in `System.Web.Controls`. In retrospect, we let internal organizational differences affect the public exposure of the APIs, and we came to regret that later. Avoid this type of mistake in your designs.

✓ **DO** use PascalCasing, and separate namespace components with periods (e.g., `Microsoft.Office.PowerPoint`). If your brand employs nontraditional casing, you should follow the casing defined by your brand, even if it deviates from normal namespace casing.

✓ **CONSIDER** using plural namespace names where appropriate.

For example, use `System.Collections` instead of `System.Collection`. Brand names and acronyms are exceptions to this rule, however. For example, use `System.IO` instead of `System.IOs`.

✗ **DO NOT** use the same name for a namespace and a type in that namespace.

For example, do not use `Debug` as a namespace name and then also provide a class named `Debug` in the same namespace. Several compilers require such types to be fully qualified.

These guidelines cover general namespace naming guidelines, but the next section provides specific guidelines for certain special subnamespaces.

3.4.1 Namespaces and Type Name Conflicts

Namespaces are used to organize types into a logical and easy-to-explore hierarchy. They are also indispensable in resolving type name ambiguities that might arise when importing multiple namespaces. However, that fact should not be used as an excuse to introduce known ambiguities between types in different namespaces that are commonly used together. Developers should not be required to qualify type names in common scenarios.

✗ **DO NOT** introduce generic type names such as `Element`, `Node`, `Log`, and `Message`.

There is a very high probability that doing so will lead to type name conflicts in common scenarios. You should qualify the generic type names (`FormElement`, `XmlNode`, `EventLog`, `SoapMessage`).

There are specific guidelines for avoiding type name conflicts for different categories of namespaces. Namespaces can be divided into the following categories:

- Application model namespaces
- Infrastructure namespaces
- Core namespaces
- Technology namespace groups

3.4.1.1 Application Model Namespaces

Namespaces belonging to a single application model are very often used together, but they are almost never used with namespaces of other application models. For example, the `System.Windows.Forms` namespace is very rarely used together with the `System.Web.UI` namespace. The following is a list of well-known application model namespace groups:

```
System.Windows*
System.Web.UI*
```

✗ **DO NOT** give the same name to types in namespaces within a single application model.

For example, do not add a type named `Page` to the `System.Web.UI.Adapters` namespace, because the `System.Web.UI` namespace already contains a type named `Page`.

3.4.1.2 Infrastructure Namespaces

This group contains namespaces that are rarely imported during development of common applications. For example, `.Design` namespaces are mainly used when developing programming tools. Avoiding conflicts with types in these namespaces is not critical.

```
System.Windows.Forms.Design
*.Design
*.Permissions
```

3.4.1.3 `Core` Namespaces

`Core` namespaces include all `System` namespaces, excluding namespaces of the application models and the Infrastructure namespaces. `Core` namespaces include, among others, `System`, `System.IO`, `System.Xml`, and `System.Net`.

✗ **DO NOT** give types names that would conflict with any type in the `Core` namespaces.

For example, never use `Stream` as a type name. It would conflict with `System.IO.Stream`, a very commonly used type.

By the same token, do not add a type named `EventLog` to the `System.Diagnostics.Events` namespace, because the `System.Diagnostics` namespace already contains a type named `EventLog`.

3.4.1.4 Technology Namespace Groups

This category includes all namespaces with the same first two namespace nodes (`<Company>.<Technology>*`), such as `Microsoft.Build.Utilities`

and `Microsoft.Build.Tasks`. It is important that types belonging to a single technology do not conflict with each other.

✗ **DO NOT** assign type names that would conflict with other types within a single technology.

✗ **DO NOT** introduce type name conflicts between types in technology namespaces and an application model namespace (unless the technology is not intended to be used with the application model).

For example, one would not add a type named `Binding` to the `Microsoft.VisualBasic` namespace because the `System.Windows.Forms` namespace already contains that type name.

3.5 Names of Classes, Structs, and Interfaces

In general, class and struct names should be nouns or noun phrases, because they represent entities of the system. A good rule of thumb is that if you are not able to come up with a noun or a noun phrase name for a class or a struct, you probably should rethink the general design of the type. Interfaces representing roots of a hierarchy (e.g., `IList<T>`) should also use nouns or noun phrases. Interfaces representing capabilities should use adjectives and adjective phrases (e.g., `IComparable<T>`, `IFormattable`).

Another important consideration is that the most easily recognizable names should be used for the most commonly used types, even if the name fits some other less-used type better in the purely technical sense. For example, a type used in mainline scenarios to submit print jobs to print queues should be named `Printer`, rather than `PrintQueue`. Even though technically the type represents a print queue and not the physical device (printer), from the scenario point of view, `Printer` is the ideal name because most people are interested in submitting print jobs and not in other operations related to the physical printer device (e.g., configuring the printer). If you need to provide another type that corresponds, for example, to the physical printer to be used in configuration scenarios, the type could be called `PrinterConfiguration` or `PrinterManager`.

KRZYSZTOF CWALINA I know this goes against the technical precision that is one of the core character traits of most software engineers, but I really do think it's more important to have better names from the point of view of the most common scenario, even if it results in slightly inconsistent or even wrong type names from a purely technical point of view. Advanced users will be able to understand slightly inconsistent naming. Most users are usually not concerned with technicalities and will not even notice the inconsistency, but they *will* appreciate the names guiding them to the most important APIs.

Similarly, names of the most commonly used types should reflect usage scenarios, not inheritance hierarchy. Most users use the leaves of an inheritance hierarchy almost exclusively, and they are rarely concerned with the structure of the hierarchy. Yet API designers often see the inheritance hierarchy as the most important criterion for type name selection. For example, `Stream`, `StreamReader`, `TextReader`, `StringReader`, and `FileStream` all describe the place of each of the types in the inheritance hierarchy quite well, but they obscure the most important information for the majority of users: the type that they need to instantiate to read text from a file.

The naming guidelines that follow apply to general type naming.

✓ **DO** name classes and structs with nouns or noun phrases, using PascalCasing.

This distinguishes type names from methods, which are named with verb phrases.

✓ **DO** name interfaces with adjective phrases, or occasionally with nouns or noun phrases.

Nouns and noun phrases should be used rarely and they might indicate that the type should be an abstract class, and not an interface. See section 4.3 for details about deciding how to choose between abstract classes and interfaces.

✗ **DO NOT** give class names a prefix (e.g., "C").

KRZYSZTOF CWALINA One of the few prefixes used is "I" for interfaces (as in `ICollection`), but that is for historical reasons. In retrospect, I think it would have been better to use regular type names. In a majority of the cases, developers don't care that something is an interface and not an abstract class, for example.

BRAD ABRAMS On the other hand, the "I" prefix on interfaces is a clear recognition of the influence of COM (and Java) on the .NET Framework. COM popularized, even institutionalized, the notation that interfaces begin with "I." Although we discussed diverging from this historic pattern, we decided to carry forward the pattern because so many of our users were already familiar with COM.

JEFFREY RICHTER Personally, I like the "I" prefix, and I wish we had more stuff like this. Little one-character prefixes go a long way toward keeping code terse and yet descriptive. As I said earlier, I use prefixes for my private type fields because I find this very useful.

BRENT RECTOR Note: This is really another application of Hungarian notation (though one without the disadvantages of the notation's use in variable names).

✓ **CONSIDER** ending the name of derived classes with the name of the base class.

This is very readable and explains the relationship clearly. Some examples of this in code are: `ArgumentOutOfRangeException`, which is a kind of `Exception`, and `SerializableAttribute`, which is a kind of `Attribute`. However, it is important to use reasonable judgment in applying this guideline; for example, the `Button` class is a kind of

`Control` event, although `Control` doesn't appear in its name. The following are examples of correctly named classes:

```
public class FileStream : Stream {...}
public class Button : Control {...}
```

✓ **DO** prefix interface names with the letter *I*, to indicate that the type is an interface.

For example, `IComponent` (descriptive noun), `ICustomAttributeProvider` (noun phrase), and `IPersistable` (adjective) are appropriate interface names. As with other type names, avoid abbreviations.

> **JEFFREY RICHTER** There is one interface I'm aware of that doesn't follow this guideline: `System._AppDomain`. It is very disconcerting to me when I see this type used without the uppercase *I*. Please don't make this same mistake in your code.

✓ **DO** ensure that the names differ only by the "I" prefix on the interface name when you are defining a class–interface pair where the class is a standard implementation of the interface.

The following example illustrates this guideline for the interface `IComponent` and its standard implementation, the class `Component`:

```
public interface IComponent { ... }
public class Component : IComponent { ... }
```

> **PHIL HAACK** One place where the Framework violates this convention is the class `HttpSessionState`, which you would suspect implements `IHttpSessionState`, but you'd be wrong, as I was.
>
> This inconsistency nearly bit us when we were developing our `HttpContextBase` abstraction of `HttpContext`, because it seemed we could expose the `Session` property as the `IHttpSessionState` interface, which turned out to not be the case.

3.5.1 Names of Generic Type Parameters

Generics were added to .NET Framework 2.0. The feature introduced a new kind of identifier called *type parameter.* The following guidelines describe naming conventions related to naming such type parameters:

✓ **DO** name generic type parameters with descriptive names unless a single-letter name is completely self-explanatory and a descriptive name would not add value.

```
public interface ISessionChannel<TSession> { ... }
public delegate TOutput Converter<TInput,TOutput>(TInput from);
public class Nullable<T> { ... }
public class List<T> { ... }
```

✓ **CONSIDER** using `T` as the type parameter name for types with one single-letter type parameter.

```
public int IComparer<T> { ... }
public delegate bool Predicate<T>(T item);
public struct Nullable<T> where T:struct { ... }
```

✓ **DO** prefix descriptive type parameter names with `T`.

```
public interface ISessionChannel<TSession> where TSession : ISession{
   TSession Session { get; }
}
```

✓ **CONSIDER** indicating constraints placed on a type parameter in the name of the parameter.

For example, a parameter constrained to `ISession` might be called `TSession`.

3.5.2 Names of Common Types

If you are deriving from or implementing types contained in the .NET Framework, it is important to follow the guidelines in this section.

✓ **DO** follow the guidelines described in Table 3-4 when naming types derived from or implementing certain .NET Framework types.

These suffixing guidelines apply to the whole hierarchy of the specified base type. For example, it is not just types derived directly from `System.Exception` that need the suffixes, but those derived from `Exception` subclasses as well.

These suffixes should be reserved for the named types. Types derived from or implementing other types should not use these suffixes. For example, the following represent incorrect naming:

```
public class ElementStream : Object { ... }
public class WindowsAttribute : Control { ... }
```

TABLE 3-4: Name Rules for Types Derived from or Implementing Certain Core Types

Base Type	Derived/Implementing Type Guideline
`System.Attribute`	✓ **DO** add the suffix "Attribute" to names of custom attribute classes.
`System.Delegate`	✓ **DO** add the suffix "EventHandler" to names of delegates that are used in events. ✓ **DO** add the suffix "Callback" to names of delegates other than those used as event handlers. ✗ **DO NOT** add the suffix "Delegate" to a delegate.
`System.EventArgs`	✓ **DO** add the suffix "EventArgs."
`System.Enum`	✗ **DO NOT** derive from this class; use the keyword supported by your language instead; for example, in C#, use the `enum` keyword. ✗ **DO NOT** add the suffix "Enum" or "Flag."
`System.Exception`	✓ **DO** add the suffix "Exception."
`IDictionary` `IDictionary<TKey,TValue>`	✓ **DO** add the suffix "Dictionary." Note that `IDictionary` is a specific type of collection, but this guideline takes precedence over the more general collections guideline that follows.

Continues

TABLE 3.4: Continued

Base Type	Derived/Implementing Type Guideline

`IEnumerable`
`ICollection`
`IList`
`IEnumerable<T>`
`ICollection<T>`
`IList<T>`
	✓ **DO** add the suffix "Collection."

`System.IO.Stream`	✓ **DO** add the suffix "Stream."
`CodeAccessPermission` `IPermission`	✓ **DO** add the suffix "Permission."

3.5.3 Naming Enumerations

Names of enumeration types (also called enums) in general should follow the standard type-naming rules (PascalCasing, etc.). However, there are additional guidelines that apply specifically to enums.

✓ **DO** use a singular type name for an enumeration unless its values are bit fields.

```
public enum ConsoleColor {
   Black,
   Blue,
   Cyan,
   ...
}
```

✓ **DO** use a plural type name for an enumeration with bit fields as values, also called flags enum.

```
[Flags]
public enum ConsoleModifiers {
   Alt,
   Control,
   Shift
}
```

✗ **DO NOT** use an "Enum" suffix in enum type names.

For example, the following enum is badly named:

```
// Bad naming
public enum ColorEnum {
    ...
}
```

✗ **DO NOT** use "Flag" or "Flags" suffixes in enum type names.

For example, the following enum is badly named:

```
// Bad naming
[Flags]
public enum ColorFlags {
    ...
}
```

✗ **DO NOT** use a prefix on enumeration value names (e.g., "ad" for ADO enums, "rtf" for rich text enums, etc.).

```
public enum ImageMode {
   ImageModeBitmap = 0, // ImageMode prefix is not necessary
   ImageModeGrayscale = 1,
   ImageModeIndexed = 2,
   ImageModeRgb = 3,
}
```

The following naming scheme would be better:

```
public enum ImageMode {
   Bitmap = 0,
   Grayscale = 1,
   Indexed = 2,
   Rgb = 3,
}
```

BRAD ABRAMS Notice that this guideline is the exact opposite of common usage in C++ programming. It is important in C++ to fully qualify each enum member because they can be accessed outside of the scope of the enum name. However, in the managed world, enum members are only accessed through the scope of the enum name.

3.6 Names of Type Members

Types are made of members: methods, properties, events, constructors, and fields. The following sections describe guidelines for naming type members.

3.6.1 Names of Methods

Because methods are the means of taking action, the design guidelines require that method names be verbs or verb phrases. Following this guideline also serves to distinguish method names from property and type names, which are noun or adjective phrases.

> **STEVEN CLARKE** Do your best to name methods according to the task that they enable, not according to some implementation detail. In a usability study on the `System.Xml` APIs, participants were asked to write code that would perform a query over an instance of an `XPathDocument`. To do this, participants needed to call the `CreateXPathNavigator` method from `XPathDocument`. This returns an instance of an `XPathNavigator` that is used to iterate over the document data returned by a query. However, no participants expected or realized that they would have to do this. Instead, they expected to be able to call some method named `Query` or `Select` on the document itself. Such a method could just as easily return an instance of `XPathNavigator` in the same way that `CreateXPathNavigator` does. By tying the name of the method more directly to the task it enables, rather than to the implementation details, it is more likely that developers using your API will be able to find the correct method to accomplish a task.

✓ **DO** give methods names that are verbs or verb phrases.

```
public class String {
   public int CompareTo(...);
   public string[] Split(...);
   public string Trim();
}
```

3.6.2 Names of Properties

Unlike other members, properties should be given noun phrase or adjective names. That is because a property refers to data, and the name of the property reflects that. PascalCasing is always used for property names.

✓ **DO** name properties using a noun, noun phrase, or adjective.

```
public class String {
   public int Length { get; }
}
```

✗ **DO NOT** have properties that match the name of "Get" methods as in the following example:

```
public string TextWriter { get {...} set {...} }
public string GetTextWriter(int value) { ... }
```

This pattern typically indicates that the property should really be a method. See section 5.1.3 for additional information.

✓ **DO** name collection properties with a plural phrase describing the items in the collection instead of using a singular phrase followed by "List" or "Collection."

```
public class ListView {
   // good naming
   public ItemCollection Items { get; }

   // bad naming
   public ItemCollection ItemCollection { get; }
}
```

✓ **DO** name Boolean properties with an affirmative phrase (`CanSeek` instead of `CantSeek`). Optionally, you can also prefix Boolean properties with "Is," "Can," or "Has," but only where it adds value.

For example, `CanRead` is more understandable than `Readable`. However, `Created` is actually more readable than `IsCreated`. Having the prefix is often too verbose and unnecessary, particularly in the face of Intellisense in the code editors. It is just as clear to type `MyObject.Enabled =` and have Intellisense give you the choice of true or false as it is to have `MyObject.IsEnabled =`, and the second one is more verbose.

KRZYSZTOF CWALINA In selecting names for Boolean properties and functions, consider testing out the common uses of the API in an if-statement. Such a usage test will highlight whether the word choices and grammar of the API name (e.g., active versus passive voice, singular versus plural) make sense as English phrases. For example, both of the following

```
if(collection.Contains(item))
if(regularExpression.Matches(text))
```

read more naturally than

```
if(collection.IsContained(item))
if(regularExpression.Match(text))
```

Also, all else being equal, you should prefer the active voice to the passive voice:

```
if(stream.CanSeek) // better than ..
if(stream.IsSeekable)
```

✓ **CONSIDER** giving a property the same name as its type.

For example, the following property correctly gets and sets an enum value named `Color`, so the property is named `Color`:

```
public enum Color {...}
public class Control {
   public Color Color { get {...} set {...} }
}
```

3.6.3 Names of Events

Events always refer to some action, either one that is happening or one that has occurred. Therefore, as with methods, events are named with verbs, and verb tense is used to indicate the time when the event is raised.

✓ **DO** name events with a verb or a verb phrase.

Examples include `Clicked`, `Painting`, `DroppedDown`, and so on.

✓ **DO** give events names with a concept of before and after, using the present and past tenses.

For example, a close event that is raised before a window is closed would be called `Closing`, and one that is raised after the window is closed would be called `Closed`.

✗ **DO NOT** use "Before" or "After" prefixes or postfixes to indicate pre- and post-events. Use present and past tenses as just described.

✓ **DO** name event handlers (delegates used as types of events) with the "EventHandler" suffix, as shown in the following example:

```
public delegate void ClickedEventHandler(object sender, ClickedEventArgs e);
```

Note that you should create custom event handlers very rarely. Instead, most APIs should simply use `EventHandler<T>`. Section 5.4.1 talks about event design in more detail.

> **JASON CLARK** Today, it is the rare case where you would need to define your own "EventHandler" delegate. Instead, you should use the `EventHandler<TEventArgs>` delegate type, where `TEventArgs` is either `EventArgs` or your own `EventArgs` derived class. This reduces type definitions in the system and ensures that your event follows the pattern described in the preceding bullet.

✓ **DO** use two parameters named *sender* and *e* in event handlers.

The sender parameter represents the object that raised the event. The sender parameter is typically of type `object`, even if it is possible to employ a more specific type. The pattern is used consistently across the Framework and is described in more detail in section 5.4.

```
public delegate void <EventName>EventHandler(object sender,
                                              <EventName>EventArgs e);
```

✓ **DO** name event argument classes with the "EventArgs" suffix, as shown in the following example:

```
public class ClickedEventArgs : EventArgs {
  int x;
  int y;
```

```
    public ClickedEventArgs (int x, int y) {
      this.x = x;
      this.y = y;
    }
    public int X { get { return x; } }
    public int Y { get { return y; } }
}
```

3.6.4 Naming Fields

The field-naming guidelines apply to static public and protected fields. Internal and private fields are not covered by guidelines, and public or protected instance fields are not allowed by the member design guidelines, which are described in Chapter 5.

✓ **DO** use PascalCasing in field names.

```
public class String {
   public static readonly string Empty ="";
}
public struct UInt32 {
   public const Min = 0;
}
```

✓ **DO** name fields using a noun, noun phrase, or adjective.

✗ **DO NOT** use a prefix for field names.

For example, do not use "g_" or "s_" to indicate static fields. Publicly accessible fields (the subject of this section) are very similar to properties from the API design point of view; therefore, they should follow the same naming conventions as properties.

BRAD ABRAMS Notice that, as with just about all the guidelines in this book, this guideline is meant to apply only to publicly exposed fields. In this case, it's important that the names be clean and simple so the masses of consumers can easily understand them. As many have noted, there are very good reasons to use some sort of convention for private fields and local variables.

JEFF PROSISE As a matter of personal preference, I typically prefix the names of private fields with an underscore (for example, `_connection`). When I read the code back after a time away, this makes it obvious to me which fields are not intended for public consumption. This convention is used quite a lot in the .NET Framework—for example, in `System.Net.HttpWebRequest` and `System.Web.HttpContext`—but it is not used throughout.

3.7 Naming Parameters

Beyond the obvious reason of readability, it is important to follow the guidelines for parameter names because parameters are displayed in documentation and in the designer when visual design tools provide Intellisense and class browsing functionality.

✓ **DO** use camelCasing in parameter names.

```
public class String {
   public bool Contains(string value);
   public string Remove(int startIndex, int count);
}
```

✓ **DO** use descriptive parameter names.

Parameter names should be descriptive enough to use with their types to determine their meaning in most scenarios.

✓ **CONSIDER** using names based on a parameter's meaning rather than the parameter's type.

Development tools must provide useful information about the type, so the parameter name can be put to better use describing semantics rather than the type. Occasional use of type-based parameter names is entirely appropriate—but it is not ever appropriate under these guidelines to revert to the Hungarian naming convention.

3.7.1 Naming Operator Overload Parameters

This section talks about naming parameters of operator overloads.

✓ **DO** use *left* and *right* for binary operator overload parameter names if there is no meaning to the parameters.

```
public static TimeSpan operator-(DateTimeOffset left,
                                 DateTimeOffset right)
public static bool operator==(DateTimeOffset left,
                              DateTimeOffset right)
```

✓ **DO** use *value* for unary operator overload parameter names if there is no meaning to the parameters.

```
public static BigInteger operator-(BigInteger value);
```

✓ **CONSIDER** meaningful names for operator overload parameters if doing so adds significant value.

```
public static BigInteger Divide(BigInteger dividend,
                                BigInteger divisor);
```

✗ **DO NOT** use abbreviations or numeric indices for operator overload parameter names.

```
// incorrect parameter naming
public static bool operator ==(DateTimeOffset d1,
                               DateTimeOffset d2);
```

3.8 Naming Resources

Because localizable resources can be referenced via certain objects as if they were properties, the naming guidelines for resources are similar to property guidelines.

✓ **DO** use PascalCasing in resource keys.

✓ **DO** provide descriptive rather than short identifiers.

Keep them concise where possible, but do not sacrifice readability for space.

✗ **DO NOT** use language-specific keywords of the main CLR languages.

✓ **DO** use only alphanumeric characters and underscores in naming resources.

✓ **DO** use the following naming convention for exception message resources.

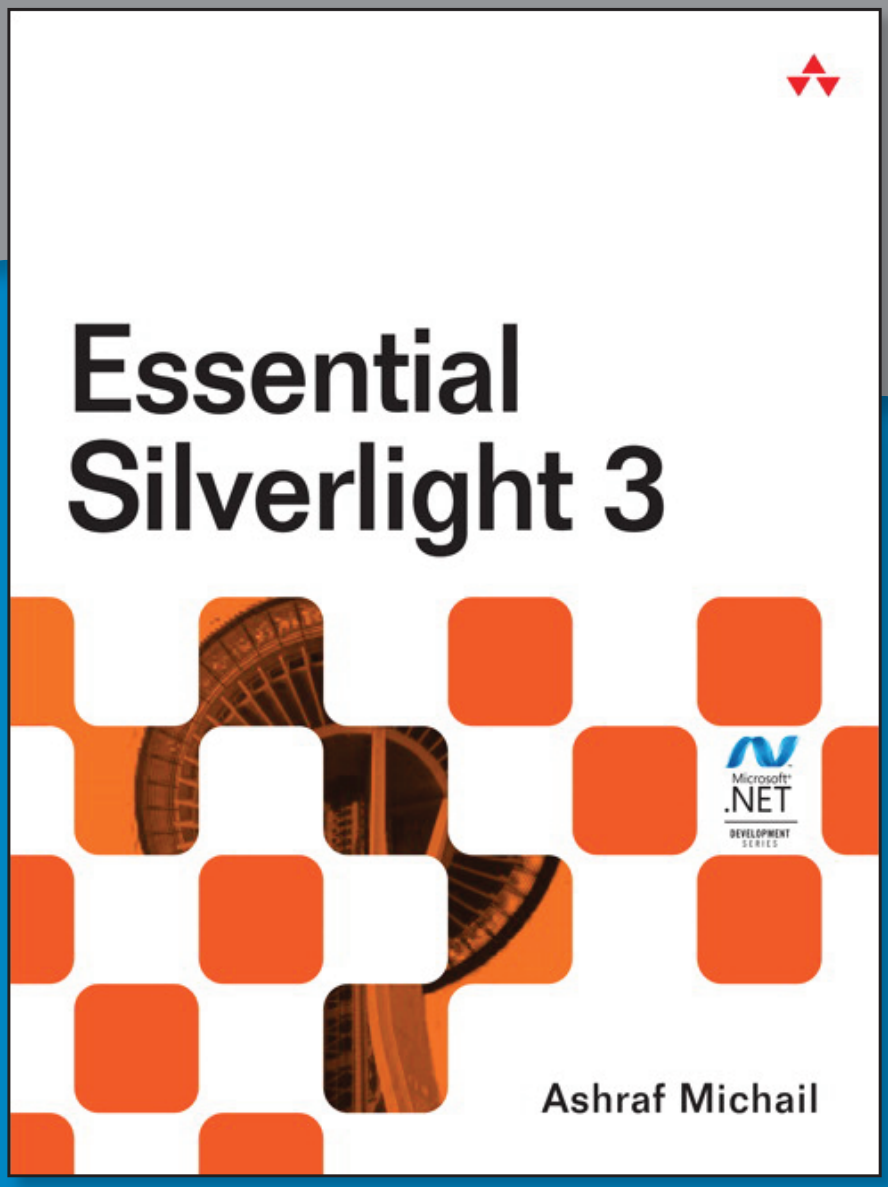
The resource identifier should be the exception type name plus a short identifier of the exception:

```
ArgumentExceptionIllegalCharacters
ArgumentExceptionInvalidName
ArgumentExceptionFileNameIsMalformed
```

SUMMARY

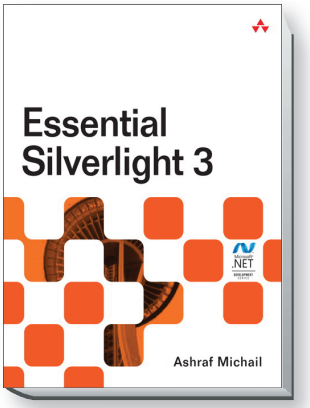
The naming guidelines described in this chapter, if followed, provide a consistent scheme that will make it easy for users of a framework to identify the function of elements of the framework. The guidelines provide naming consistency across frameworks developed by different organizations or companies.

The next chapter provides general guidelines for implementing types.



Ashraf Michail

BUY ME



Essential Silverlight 3

Get under the hood with Silverlight 2: a deep look at the platform's inner workings, by its long-time lead architect

- Crucial, never-before-published information for building Silverlight 2 applications that perform better, are more reliable, and are easier to deploy
- Includes extensive code examples that show apply the author's insights in real applications
- Unparalleled, in-depth coverage of Silverlight graphics, text, audio, video, animation, input, layout, and data binding

About the Author

Ashraf Michail is the only Microsoft architect who has continued to work on Silverlight since the original project began. In 2001, he joined Microsoft's newly forming WPF team, where he built the GPU accelerated graphics engine used to render WPF content and included in the Vista Desktop Window Manager. In 2004, he became a WPF architect focused on improving the end-to-end WPF experience. In 2005, he became an architect on the new Silverlight team, where he is currently working on Silverlight's next release. With nine years of experience delivering web platforms and rendering engines, Michail's deep insights have guided Silverlight's design.

AVAILABLE
- BOOK: 9780321554161
- SAFARI ONLINE
- EBOOK: 0321562283
- KINDLE: 0321562291

Due to Publish in October 2009

Please note that the text in this chapter has not yet been proofread or copy-edited and may contain grammatical errors. The inclusion of this text is to show an example of what is included in this book.

Chapter 3

Graphics

In this chapter, you will learn how to add rich vector graphics and images to your application. You will also learn how to optimize performance and image quality of those graphics elements. In particular, Chapter 3 will discuss:

- The graphics system design principles
- The elements for displaying graphics
- The problems the Silverlight run-time solves under the hood and the problems your application must solve

Graphics Principles

The Silverlight graphics API makes it easy to add vector graphics, bitmap images, and text to your applications. This section describes the graphics API design principles.

Vector Graphics and Bitmap Images

Bitmap images are a common method of adding graphics to an application. However, bitmap images become blurry when scaled up, as shown in Figure 3-1, or aliased when scaled down, as shown in Figure 3-2. Unlike a bitmap image, if you scale a vector graphic, it will remain sharp as shown in Figure 3-3. Both vector graphics and bitmap images are useful in most applications. For example, a user interface control will look better at different sizes and resolutions with vector graphics instead of bitmap images. Bitmap images are useful for displaying content that is not easily expressible in vector form such as digital photographs or visual effects not supported by the run-time.

New in Silverlight 3

There are a number of techniques for scaling down an image to produce a result better than Figure 3-2. However, these techniques can be computationally expensive and slow down your animations. Silverlight 3 adds support for mip-mapping that converts your image to a set of smaller images at various sizes using a better algorithm. This conversion happens as Silverlight is downloading and decoding your images. During an animation, Silverlight 3 will then dynamically select the right resolution to display. This process can increase memory usage, but substantially improves the display quality of scaled down images and 3D transforms applied bitmap images.

Scaling up images in Silverlight 3 still remains blurry as shown in Figure 3-1.

Figure 3-1
Scaling up an image of a circle

Figure 3-2
Scaling down an image of a circle

Figure 3-3
Scaling a vector graphic circle

Retained Mode

There are two types of graphics API: retained mode and immediate mode. A retained mode API automatically responds to changes to a graph of objects. An immediate mode API

requires you to issue all the draw commands to describe a change. For example, to remove the rectangle shown in Figure 3-4 in a retained mode system, simply call a function to remove that element. The retained mode graphics system is responsible for redrawing the background, the triangle beneath, and the ellipse above. To remove the same rectangle shown in Figure 3-4 with an immediate mode API, you need to make three calls to draw the background, the triangle beneath it, and the ellipse above as shown in Figure 3-5.

A retained mode API enables you to:

- Construct a set of graphics elements
- Change properties of the graphics elements
- Build a graph representing the relationship between those elements
- Manipulate the graph structure

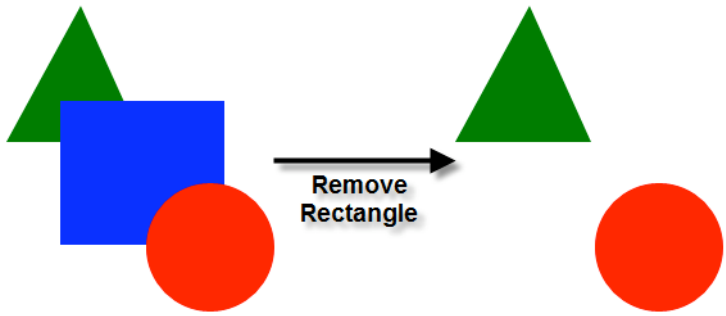


Figure 3-4
Removing a rectangle with a retained mode API

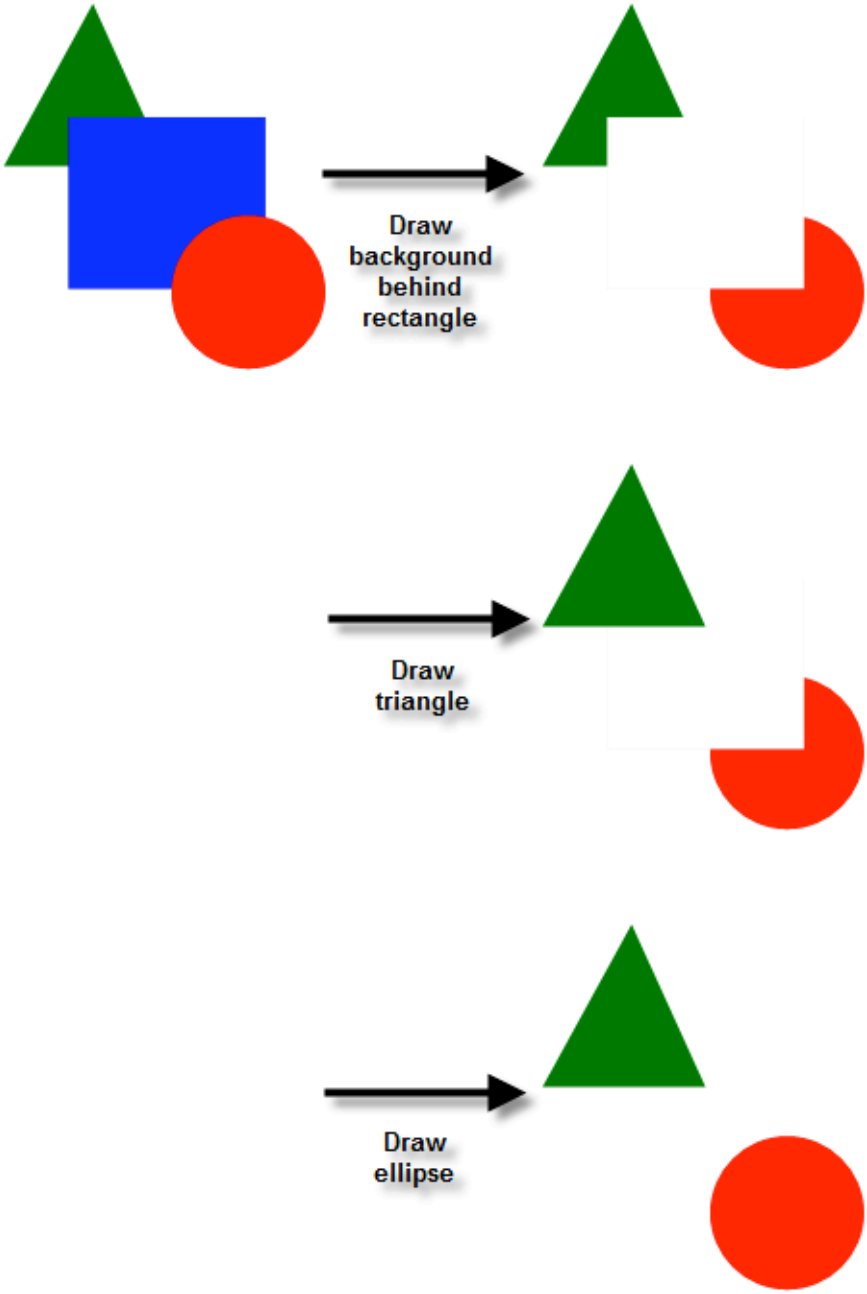


Figure 3-5
Removing a rectangle with an immediate mode API

A retained mode graphics API is easier to use than an immediate mode API and enables the underlying system to provide automatic performance optimizations such as drawing incrementally and avoiding the drawing of occluded shapes. Silverlight provides a retained mode system to optimize for ease of use, animating vector graphics content, and for building applications composed of UI controls.

In Silverlight, you can construct the retained mode graph declaratively with a XAML file:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <!-- triangle -->
    <Path
        Fill="Green"
        Data="F1 M 128,12L 12,224L 224,224"
    />

    <!-- rectangle -->
    <Rectangle
        Fill="Blue"
        Canvas.Left="96"
        Canvas.Top="160"
        Width="256"
        Height="224"
    />

    <!-- circle -->
    <Ellipse
        Fill="Red"
        Canvas.Left="230"
        Canvas.Top="288"
        Width="200"
        Height="200"
    />

</Canvas>
```

Alternatively, you can construct the retained mode graph with code:

```
Canvas canvas = new Canvas();

//
// Make the triangle
//

Path path = new Path();

path.Fill = new SolidColorBrush(Colors.Green);
path.SetValue(Path.DataProperty, "F1 M 128,12L 12,224L 224,224");

canvas.Children.Add(path);


//
// Make the rectangle
//

Rectangle rectangle = new Rectangle();

rectangle.Fill = new SolidColorBrush(Colors.Blue);
rectangle.SetValue(Canvas.LeftProperty, 96);
rectangle.SetValue(Canvas.TopProperty, 160);
rectangle.Width = 256;
```

```
rectangle.Height = 224;

canvas.Children.Add(rectangle);

//
// Make the circle
//

Ellipse ellipse = new Ellipse();

ellipse.Fill = new SolidColorBrush(Colors.Red);
ellipse.SetValue(Canvas.LeftProperty, 230);
ellipse.SetValue(Canvas.TopProperty, 288);
ellipse.Width = 200;
ellipse.Height = 200;

canvas.Children.Add(ellipse);
```

Cross Platform Consistency

An important goal for the Silverlight graphics engine is to enable a developer to write their application once and have it run consistently across a variety of operating systems and browsers. Each operating system has a local graphics library. However, these local operating system graphics libraries differ significantly in feature set, performance, and image quality. To ensure cross-platform consistency and performance, Silverlight includes its own rendering engine.

Tools

Silverlight is capable of loading vector and image content from designer tools and integrating with developer written application code. For vector graphics and animation, you can use Expression Design and Expression Blend to generate XAML content for use with the Silverlight run-time. There are also a variety of free XAML exporters available to generate XAML content including an Adobe Illustrator exporter, an XPS print driver, and several others.

Balancing Image Quality and Speed

In addition to displaying static XAML, Silverlight provides real-time animation at 60 frames per second. However, real-time performance is highly dependent on the application content, the specific hardware configuration of the target machine, the resolution of the target display, the operating system, and the hosting web browser.

When an application does not meet the 60 frame per second goal, the three options the Silverlight team uses to improve performance are:

- Make optimizations to components in the Silverlight run-time
- Lower image quality to achieve better speed

Reducing image quality for speed is the most controversial optimization technique. The Silverlight application must look good. However, it is possible to trade off image quality in a manner that is generally acceptable to the end-user. For example, vector graphics rendering makes some quality sacrifices but still maintains an acceptable visual bar. On the other hand,

end users have a much higher standard for text quality and the Silverlight run-time spends more CPU time rendering text clearly.

Graphics Elements

As previously discussed, the Silverlight run-time can load and draw vector graphics XAML on a variety of platforms. The graphics elements can also be specified programmatically when you use C# or JavaScript. This section will describe the elements you can use to display vector graphics and images in your applications.

Shapes

This section will start the graphics element discussion with the base class for all graphics primitives: the `Shape` element. The `Shape` element provides:

- A `Fill` property to specify how the shape interior is drawn
- A `Stretch` property to indicate how a `Shape` element stretches to a specified `Width` and `Height`
- `Stroke` and `StrokeThickness` properties to specify the pen behavior

The elements that derive from `Shape` will define the shape's geometry. These elements include `Rectangle`, `Ellipse`, `Line`, `Polygon`, `Polyline`, and `Path`.

Rectangle

To draw a rectangle at a specified position, place it in a `Canvas` element and specify its `Canvas.Top`, `Canvas.Left`, `Width`, `Height`, and `Fill` color:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Rectangle
        Fill="LightGray"
        Canvas.Left="50"
        Canvas.Top="50"
        Width="400"
        Height="400"
    />

</Canvas>
```

You can add an outline to the rectangle as shown in Figure 3-6 by setting the `Stroke` property to specify the color and the `StrokeThickness` property to specify the thickness:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Rectangle
        Fill="LightGray"
        Stroke="Black"
        StrokeThickness="10"
```

```
        Canvas.Left="50"
        Canvas.Top="50"
        Width="400"
        Height="400"
    />

</Canvas>
```

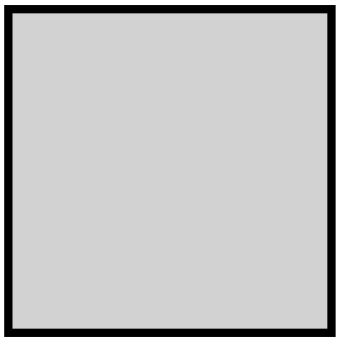


Figure 3-6
Rectangle element with an outline

You can use the `Rectangle` element to draw the rounded rectangles shown in Figure 3-7 by specifying the `RadiusX` and `RadiusY` properties:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Rectangle
        Fill="LightGray"
        Stroke="Black"
        StrokeThickness="10"
        RadiusX="40"
        RadiusY="60"
        Canvas.Left="50"
        Canvas.Top="50"
        Width="400"
        Height="400"
    />

</Canvas>
```

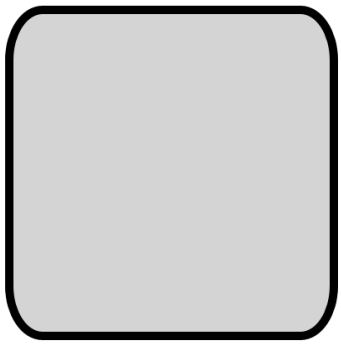


Figure 3-7
Rectangle element with rounded corners

Ellipse

As with the `Rectangle` element, you can position an `Ellipse` element with the same `Canvas.Top`, `Canvas.Left`, `Width`, and `Height` properties. Silverlight will stretch the ellipse to fit the specified bounds as shown in Figure 3-8:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Ellipse
        Fill="LightGray"
        Stroke="Black"
        StrokeThickness="10"
        Canvas.Left="50"
        Canvas.Top="50"
        Width="400"
        Height="400"
    />

</Canvas>
```

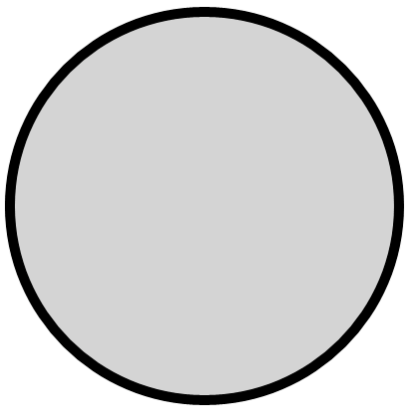


Figure 3-8
Ellipse element

Line

To draw a line, you can use the `Line` element and set its `X1`, `Y1`, `X2`, `Y2` properties. As with all other shapes, you can use the `Stroke` property to specify the fill color, and the `StrokeThickness` property to specify the thickness as shown in Figure 3-9:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <!-- thick diagonal line -->
    <Line
        Stroke="Black"
        StrokeThickness="40"
        X1="60"
        Y1="60"
        X2="400"
        Y2="400"
    />

    <!-- one pixel horizontal line -->
    <Line
        Stroke="Black"
        StrokeThickness="1"
        X1="100"
        Y1="60"
        X2="400"
        Y2="60"
    />

</Canvas>
```

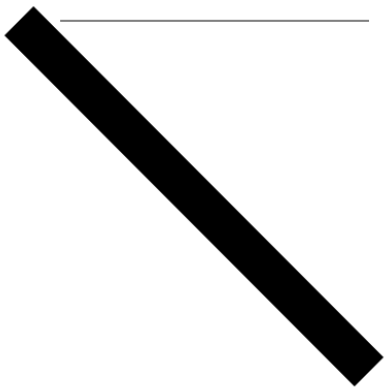


Figure 3-9
Line element

If you look closely at the pixels for the horizontal line shown in Figure 3-10, you will see it has a height of two pixels despite the one pixel `StrokeThickness` specified in the XAML. Furthermore, the line is grey instead of the specified black color. To understand this rendered result, consider the following equivalent `Rectangle` element:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">
```

```
    <!-- one pixel horizontal line drawn as a rectangle -->
    <Rectangle
        Fill="Black"
        Canvas.Left="99.5"
        Canvas.Top="59.5"
        Width="301"
        Height="1"
    />

</Canvas>
```

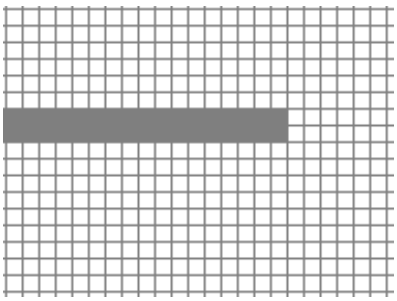


Figure 3-10

Pixels rendered for a Line element

The previous `Rectangle` element has half integer values for its position. The reason for the half pixel coordinates is that the line expands by half `StrokeThickness` in either direction. Since `StrokeThickness` is equal to one, the line adjusts the top and left coordinates by -0.5. Since the rectangle is between two pixels, it gets anti-aliased and occupies two pixels with a lighter color. If you want sharp horizontal and vertical lines, you should draw a rectangle positioned at integer coordinates to get the result shown in Figure 3-11:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <!-- one pixel horizontal line drawn as a rectangle -->
    <Rectangle
        Fill="Black"
        Canvas.Left="99"
        Canvas.Top="59"
        Width="301"
        Height="1"
    />

</Canvas>
```

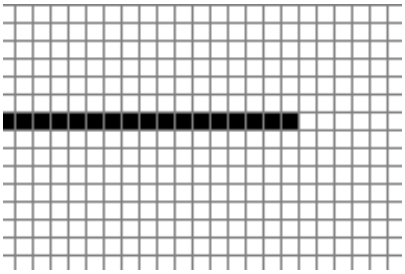


Figure 3-11
Sharp horizontal line drawn with a Rectangle element

Path

The `Path` element extends the `Shape` element by providing a `Data` property that specifies the geometry object. The `Rectangle`, `Ellipse`, and `Line` elements previously discussed are all expressible with the more general `Path` element. For example, instead of specifying a `Rectangle` element, we can specify a `Path` element with a `RectangleGeometry`:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Path
        Fill="Blue"
        Stroke="Black"
        StrokeThickness="10"
    >
        <Path.Data>
            <RectangleGeometry Rect="96,160,256,224"/>
        </Path.Data>
    </Path>

</Canvas>
```

The `Path` element syntax is more verbose than the specialized shapes syntax. However, since Silverlight converts all shapes internally to `Path` elements, if you understand how the `Path` element draws you will understand how all shapes draw.

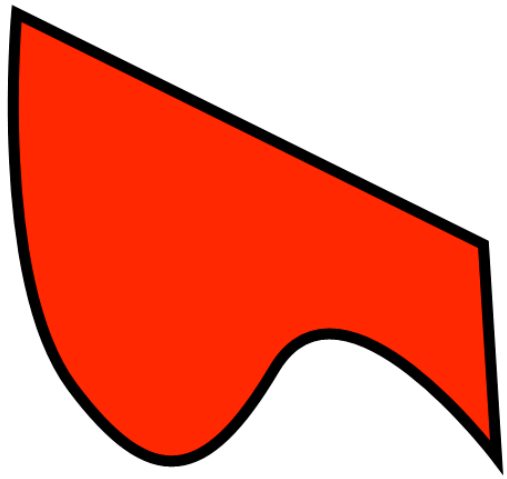


Figure 3-12
Example path

In addition to expressing specialized shapes, the `Path` element can express a geometry consisting of a collection of `Figure` elements. A `Figure` element is a connected set of line segments and Bezier segments. The most common method to specify these figures, line segments, and curves is the path mini-language. For example, to draw the shape in Figure 3-12 you would do the following:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Path
        StrokeThickness="10"
        Stroke="Black"
        Fill="Red"
        Data="M 14,16
              C 14,16 -8,256 64,352
              C 136,448 185,440 247,336
              C 309,233 448,416 448,416
              L 436,224
              Z"
    />

</Canvas>
```

The commands supported by the mini-language include those shown in Table 3-1. Each command is followed by the action it takes.

Table 3-1
Path Mini-language Commands

Command	Action
`M` *x,y*	Move to position x,y
`L` *x,y*	Draw a line from the current position to position *x, y*
`C` *x1,y1, x2,y2, x3, y3*	Draw a cubic Bezier segment with control points consisting of the current position, (*x1,y1*), (*x2,y2*), and (*x3, y3*)
`Q` *x1,y1, x2,y2*	Draw a quadratic Bezier segment with control points consisting

	of the current position, (x*1,y1*), and (*x2,y2*)
`H` *x*	Draw a horizontal line from the current position *x0,y0* to position *x, y0*
`V` *y*	Draw a vertical line from the current position *x0,y0* to position *x0, y*
`Z`	Close a figure
`F0`	Specify EvenOdd fill rule
`F1`	Specify NonZero fill rule

An alternative form of specifying the geometry is to use the expanded XAML syntax:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Path
        StrokeThickness="10"
        Stroke="Black"
        Fill="Red"
    >
        <Path.Data>
            <PathGeometry>
               <PathGeometry.Figures>
                  <PathFigure StartPoint="14, 16" IsClosed="true">
                     <PathFigure.Segments>
                        <BezierSegment
                            Point1="14,16"
                            Point2="-8,256"
                            Point3="64,352"
                        />
                        <BezierSegment
                            Point1="136,448"
                            Point2="185,440"
                            Point3="247,336"
                        />
                        <BezierSegment
                            Point1="309,233"
                            Point2="448,416"
                            Point3="448,416"
                        />
                        <LineSegment
                            Point="436,224"
                        />
                     </PathFigure.Segments>
                  </PathFigure>
               </PathGeometry.Figures>
            </PathGeometry>
        </Path.Data>
    </Path>

</Canvas>
```

Performance Tip

These two forms of specifying a path may render identically, but they differ in performance and flexibility. The mini-language form will parse faster, consume less memory, and draw fastest at run-time. The mini-language is the recommended form for static content. However, it is not possible to bind an animation or change a segment property of a path generated with the mini-language since Silverlight does not create the path, figure, or segment API objects.

One additional concept previously discussed is that of a `Figure` element. Because the `Path` element can have more than one figure, it can create an empty space in the middle of a filled space as shown in Figure 3-13:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Path
        StrokeThickness="10"
        Stroke="Black"
        Fill="LightGray"
        Data="M 50,50 L 50,450 450,450 450,50 50,50 z
              M 100,100 L 100,400 400,400 400,100 100,100 z"
    />

</Canvas>
```

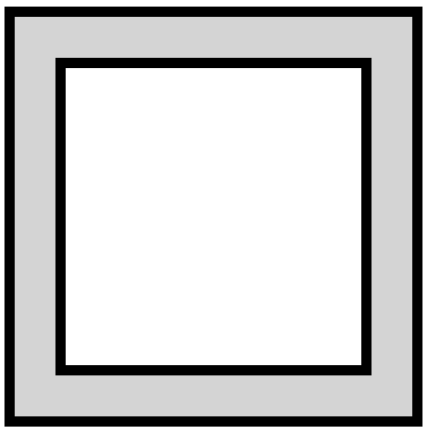


Figure 3-13
Path with an empty space in the center

Fill rules

The previous section explained how to use geometry to specify an outline of an area to fill. However, an outline does not uniquely specify the inside and outside of the shape. For example, the outline in Figure 3-13 could generate any of the rendering results shown in Figure 3-14. What is missing is a method to distinguish the inside of a shape from the outside of the shape.

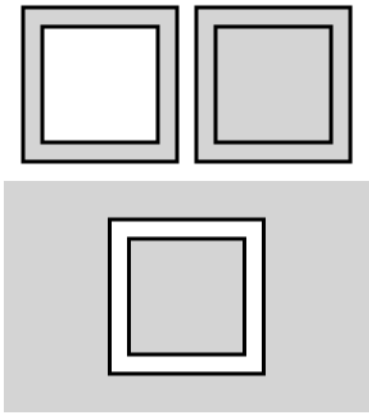


Figure 3-14
Different fills for the same outline

One approach to specifying what is inside a shape is to cast a horizontal line through the shape and count the number of edges crossed from left to right. If the number of edges crossed is even, classify the horizontal line segment as outside the shape. If the number of edges is odd, classify that segment as inside the shape. This fill rule is the `EvenOdd` rule and is the default fill mode for `Path` elements. To specify the fill mode explicitly, you can specify the `FillRule` property on geometry or use `F0` for `EvenOdd` from the path mini-language:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <!-- Path with fill rule F0 = EvenOdd -->

    <Path
        StrokeThickness="10"
        Stroke="Black"
        Fill="LightGray"
        Data="F0
              M 50,50 L 50,450 450,450 450,50 50,50 z
              M 100,100 L 100,400 400,400 400,100 100,100 z"
    />

</Canvas>
```

An alternative rule is the `NonZero` rule, which considers the order points are specified in the input. If an edge goes up in the y direction, assign that edge a +1 winding number. If an edge goes down in the y direction, assign that edge a -1 winding number. The `NonZero` rule defines the interior of the shape to be all those segments where the sum of winding numbers from the left to the current segment is not zero. For example, if you specify the geometry shown in Figure 3-14 with the point order in the following markup, it would result in the winding numbers and filled segments shown in Figure 3-15.

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <!-- Path with fill rule F1 = NonZero -->
```

```
    <Path
        StrokeThickness="10"
        Stroke="Black"
        Fill="LightGray"
        Data="F1
              M 50,50 L 50,450 450,450 450,50 50,50 z
              M 100,100 L 100,400 400,400 400,100 100,100 z"
    />

</Canvas>
```

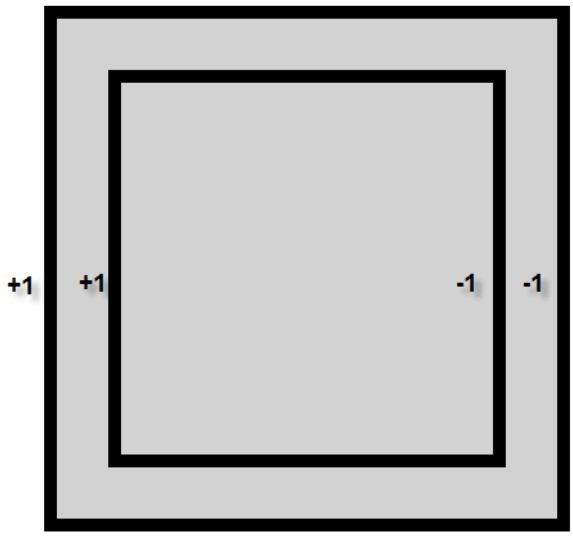


Figure 3-15
Winding mode numbers resulting in a filled center space

If you specify the points in the following order, the shape would render differently as shown in Figure 3-16:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <!-- Path with fill rule F1 = NonZero -->

    <Path
        StrokeThickness="10"
        Stroke="Black"
        Fill="LightGray"
        Data="F1
              M 50,50 L 50,450 450,450 450,50 50,50 z
              M 100,100 L 400,100 400,400 100,400 100,100 z"
    />

</Canvas>
```

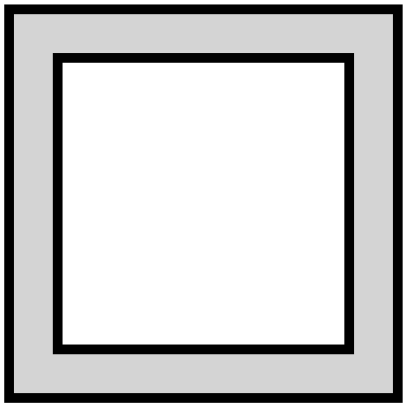


Figure 3-16
Different fill as a result of a different point order

Performance Tip

The `NonZero` rule is more complicated than the `EvenOdd` rule and does render slower. For most vector graphics fills, the `EvenOdd` rule gives the desired result.

Images

In addition to the vector graphics elements previously discussed, the other fundamental graphics primitive is the `Image` element. To display an image, you can use the `Image` element with a reference to a URI to produce the result shown in Figure 3-17:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">
    <Image Source="silverlight.png"/>
</Canvas>
```

The `Source` property can reference any image in JPG or PNG format. However, if the JPG or PNG contains DPI (dots per inch) information, Silverlight ignores this information because it is usually not accurate for display purposes. All references to URI's in XAML are relative to the location of the XAML file. For example, if the XAML file is in a XAP, Silverlight will search for silverlight.png in the XAP. If the XAML file is a resourced in a managed assembly, Silverlight will search for silverlight.png in that same assembly.

If you do not specify the `Width` and `Height` properties of an image, Silverlight draws the image at its natural size, which results in a pixel perfect rendering of the original image data.

Figure 3-17
Image element example

Technical Insight

As shown in the previous example, the `Source` property of an `Image` element can be set to a URI. Many references to the same `Source` URI will cause Silverlight to download the image once and use it multiple times. If you remove all references to `Image` elements for a specific URI, Silverlight removes the image from the memory cache and a future reference will cause Silverlight to initiate another download. Future downloads may be serviced from the browser cache or go over the network if the image is no longer in the browser cache.

Brushes

All of the previous examples filled the `Path` element pixels with a single color. Silverlight also supports filling arbitrary shapes with image brushes, linear gradient brushes, and radial gradient brushes. A brush is a mathematical function that maps an *(x,y)* position to a color. For example, `SolidColorBrush` is a simple function that ignores its position and always outputs the same color. This section will describe the brushes available in Silverlight and include the function used to map from screen position to color.

Solid Color Brush

`SolidColorBrush` returns a single color for all screen positions. When you specify a `Fill` or `Stroke` property value, Silverlight will create a solid color brush for the corresponding shape fill or pen stroke respectively:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Rectangle
        Fill="Blue"
```

```
        Stroke="Black"
        StrokeThickness="10"
        Canvas.Left="96"
        Canvas.Top="160"
        Width="256"
        Height="224"
    />

</Canvas>
```

Alternatively, you can specify a brush with the expanded XAML syntax:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Rectangle
        StrokeThickness="10"
        Canvas.Left="96"
        Canvas.Top="160"
        Width="256"
        Height="224"
    >
        <Rectangle.Fill>
            <SolidColorBrush Color="Blue"/>
        </Rectangle.Fill>
        <Rectangle.Stroke>
            <SolidColorBrush Color="Black"/>
        </Rectangle.Stroke>
    </Rectangle>

</Canvas>
```

The previous examples specified a named color. You can also specify a color explicitly by providing a hex string of the form #aarrggbb which represents a hex alpha channel value, red channel value, green channel value, and blue channel value. For example, opaque green would be #ff 00ff00.

From C#, you can specify a color by creating an instance of the `Color` class:

```
Color green = Color.FromArgb(0xff, 0x0, 0xff, 0x0);
```

The alpha channel specifies the degree of transparency where `0x0` indicates completely transparent, `0xff` indicates an opaque color, and intermediate values indicate partial transparency. A brush with a transparent color will blend its color to the background color using the following formula:

$$Color\_destination = (alpha*Color\_source + (0xff-alpha)*Color\_destination)/256$$

Silverlight will pass colors specified in hex format to the web browser without a color space conversion. Typically, the browser will interpret the color as a sRGB color, i.e., an 8-bit per channel 2.2 implied gamma space color. However, the visible color may vary with

operating systems, web browser, the monitor, and the operating system color profile. An alternative form of specifying colors is the floating point scRGB linear gamma format:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Rectangle
        Fill="sc#1, 1.0, 0.0, 0.0"
        Canvas.Left="96"
        Canvas.Top="160"
        Width="256"
        Height="224"
    />

</Canvas>
```

Silverlight converts scRGB colors to sRGB internally for blending. Consequently, specifying colors in native sRGB is desirable to avoid extra color conversion steps.

Gradient Brushes

Gradient brushes define a set of colors and positions along a virtual gradient line. The function that maps screen position to color will first map the screen position to a point along the gradient line and then interpolate a color based on the two nearest points.

Consider the following use of `LinearGradientBrush`:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Ellipse
        Width="450"
        Height="450"
    >
        <Ellipse.Fill>
            <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
                <LinearGradientBrush.GradientStops>
                    <GradientStop Color="White" Offset="0"/>
                    <GradientStop Color="Black" Offset="1"/>
                </LinearGradientBrush.GradientStops>
            </LinearGradientBrush>
        </Ellipse.Fill>
    </Ellipse>

</Canvas>
```

The linear gradient brush above will produce the fill shown in Figure 3-18. A linear gradient brush will map a screen position to the point on the line closest to that position.

The brush will then interpolate a color based on the two nearest points specified along the line as shown in Figure 3-18.

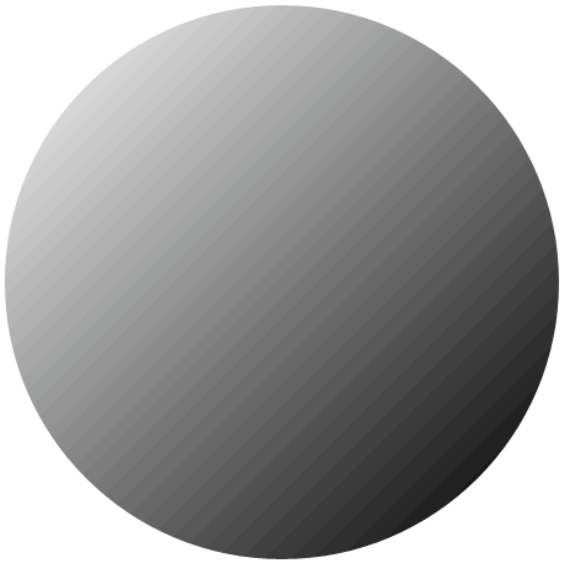


Figure 3-18
Linear gradient brush

Alternatively, you can specify a radial gradient fill using `RadialGradientBrush` that will take the distance from the screen position to the center of the radial gradient and map that distance to the specified gradient line of colors and positions. For example, the XAML below will generate the rendering shown in Figure 3-19:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Ellipse
        Width="450"
        Height="450"
    >
```

```
        <Ellipse.Fill>
            <RadialGradientBrush>
                <RadialGradientBrush.GradientStops>
                    <GradientStop Color="White" Offset="0"/>
                    <GradientStop Color="Black" Offset="1"/>
                </RadialGradientBrush.GradientStops>
            </RadialGradientBrush>
        </Ellipse.Fill>
    </Ellipse>

</Canvas>
```

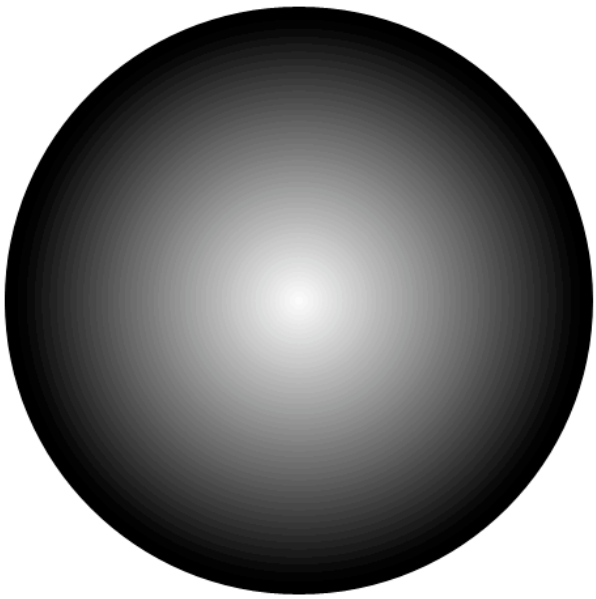


Figure 3-19
Radial gradient brush

Another feature of `RadialGradientBrush` is the capability to move the point that maps to the start of our gradient line. In particular, in our previous example, the center of the radial gradient circle mapped to the start of our gradient line and the radius of the gradient circle mapped to the end of our gradient line. With this pattern, you will always get a uniform ellipse. You can move the center using the `Center` property to get the result shown in Figure 3-20:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Ellipse
        Width="450"
        Height="450"
    >
        <Ellipse.Fill>
            <RadialGradientBrush GradientOrigin="0.25 0.25">
                <RadialGradientBrush.GradientStops>
                    <GradientStop Color="White" Offset="0"/>
                    <GradientStop Color="Black" Offset="1"/>
```

```
                </RadialGradientBrush.GradientStops>
            </RadialGradientBrush>
        </Ellipse.Fill>
    </Ellipse>

</Canvas>
```

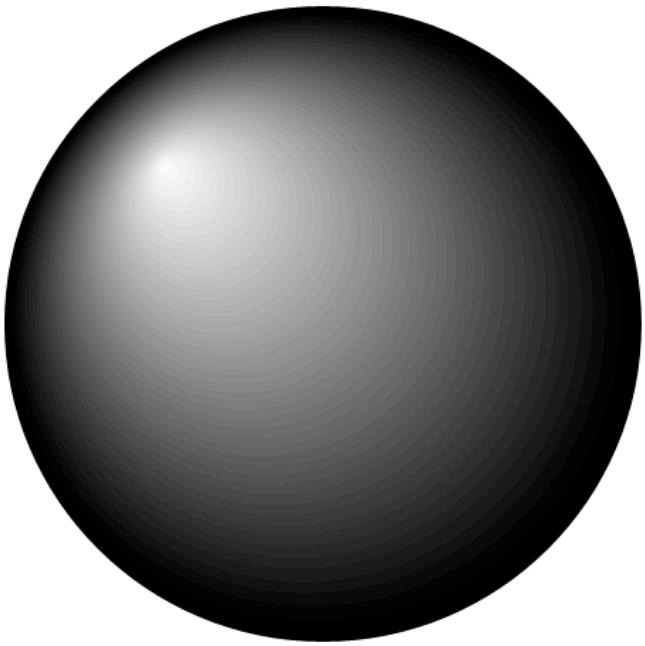


Figure 3-20
Focal gradient brush

One other feature of linear and radial gradients is the capability to specify the behavior when the display position maps to some position outside the range of the gradient line. The `SpreadMethod` property defines that behavior. The `Pad` mode will repeat the closest point when off the line, the `Reflect` mode will mirror to a point on the line, and the `Repeat` mode will simply take the position modulo the length of the line as shown in Figure 3-21.

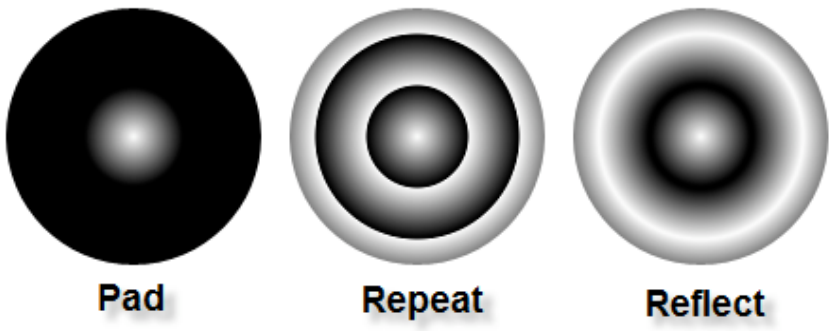


Figure 3-21
SpreadMethod example

Image Brushes

The role of the image brush is to map a screen position to a pixel in the specified image. For example, the following XAML would result in the image brush rendering shown in Figure 3-22:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Ellipse
        Width="450"
        Height="450"
        Stroke="Black"
        StrokeThickness="10"
    >
        <Ellipse.Fill>
            <ImageBrush ImageSource="silverlight.png"/>
        </Ellipse.Fill>
    </Ellipse>

</Canvas>
```

Figure 3-22
ImageBrush example

Strokes

The previous section showed how to use a brush to color the fill of a shape. You can also use a brush to add color to the outline of a shape by setting the stroke properties. For example, the following XAML would generate the output shown in Figure 3-23:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">
    <Ellipse
```

```
        Stroke="Black"
        StrokeThickness="10"
        Canvas.Left="50"
        Canvas.Top="50"
        Width="400"
        Height="400"
    />
</Canvas>
```

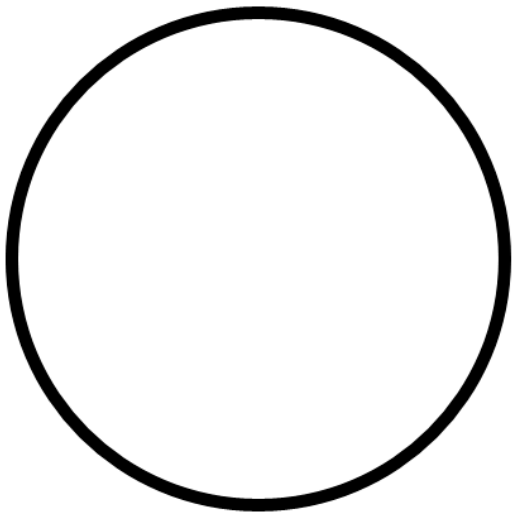


Figure 3-23
Sample stroke applied to an ellipse

Stroke

A stroke transforms geometry to a widened form that describes the shape outline instead of the shape fill. Silverlight will fill the widened geometry with exactly the same rendering rules as the main shape fill. For example, Figure 3-24 shows an example of a widened ellipse.

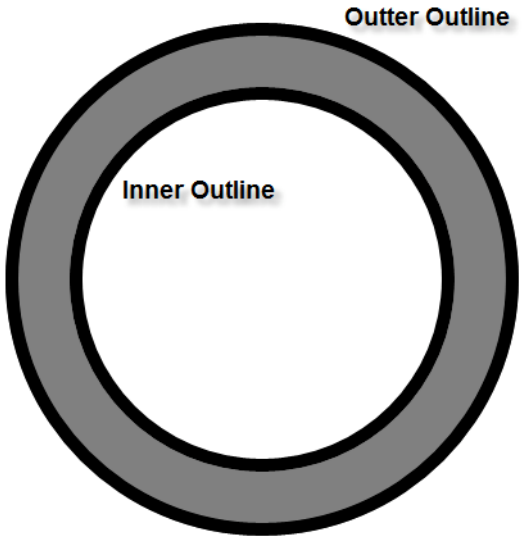


Figure 3-24
The widening process applied to an ellipse

The widening process will expand the original geometry by half the stroke thickness to form an outer outline. The widening process will also shrink the original geometry by half the stroke thickness to form an inner outline. The outer and inner outlines combine to form two figures Silverlight fills to produce the resulting stroke.

Technical Insight

One side effect of the widening process is that local self-intersection can occur. For example, the process of widening a triangle will generate several self-intersections as shown in Figure 3-25. One option is to run a loop removal algorithm to remove these intersections before rasterization. However, by simply filling the new geometry with the `NonZero` fill rule, these self intersections are not visible to the end user.

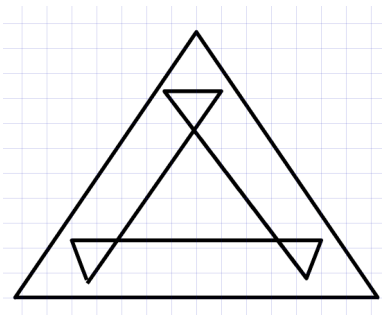


Figure 3-25
The widening process applied to a triangle

Dashes

To add dashes to your strokes, specify an array of distances alternating between the dash filled distance and the gap distance. For example, the simple dash array in the following XAML generates the output shown in Figure 3-26:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">
    <Ellipse
        Stroke="Black"
        StrokeThickness="10"
        StrokeDashArray="5, 4, 2, 4"
        Canvas.Left="50"
        Canvas.Top="50"
        Width="400"
        Height="400"
    />
</Canvas>
```

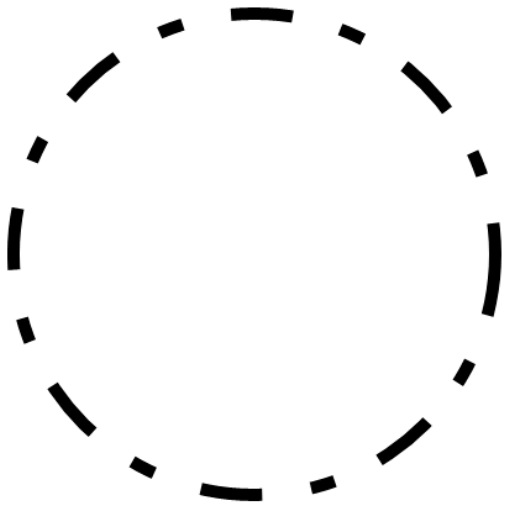


Figure 3-26
StrokeDashArray example of long and short dashes

Technical Insight

Dashes are also a geometry modifier built on top of the stroke geometry modifier. When you specify a `StrokeDashArray`, Silverlight will take the output of the pen and subdivide it into smaller geometries. Large numbers of dashes can result in significant slowdowns in rendering speed and therefore you should use them sparingly.

Canvas

Every example shown so far has had a single root `Canvas` element with a set of `Shape` elements contained within it. In addition to providing a convenient container, the `Canvas` element also enables you to modify the rendering primitives it contains as a group. In particular, the `Canvas` element enables the following:

- Naming groups of elements
- Grouping shapes so that you can add or remove the group with a single operation
- Applying a transform to the group of elements
- Clipping a group of elements
- Apply an opacity or opacity mask effect to a group of elements

Transforms, clipping, and opacity effects are available on both individual shapes and the `Canvas` element.

Performance Tip

For individual shapes, it is faster to express clipping or opacity as a different geometry or a different brush color. For example, draw a `Path` with an `ImageBrush` to achieve the same result as applying a clip to an `Image` element. Similarly, you can add opacity to the brush color alpha channel instead of adding `Opacity` to the shape.

Transforms

A transform enables you to position, rotate, scale, or skew a shape or group of shapes. To transform a group of primitives, you can set the `RenderTransform` on the `Canvas` element as exemplified in the following listing to achieve the result shown in Figure 3-27:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">
    <Canvas.RenderTransform>
        <TransformGroup>
            <ScaleTransform ScaleX="1.5"/>
            <RotateTransform Angle="30"/>
            <TranslateTransform X="100" Y="-10"/>
        </TransformGroup>
    </Canvas.RenderTransform>

    <Ellipse
        Fill="LightGray"
        Stroke="Black"
        StrokeThickness="20"
        Width="200"
        Height="200"
    />
    <Rectangle
        Fill="Gray"
        Stroke="Black"
        StrokeThickness="20"
        Canvas.Left="100"
```

```
        Canvas.Top="100"
        Width="200"
        Height="200"
    />

</Canvas>
```

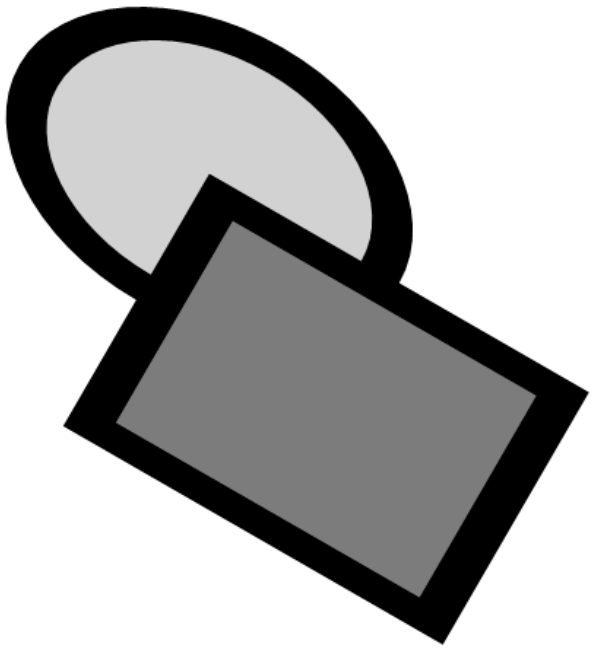


Figure 3-27
RenderTransform example of overlapping a rectangle over an ellipse

As shown in the previous example, you can use a list of `ScaleTransform`, `TranslateTransform`, and `RotateTransform` elements in a `TransformGroup` element. Alternatively, you can specify an explicit matrix with a `MatrixTransform`:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">
    <Canvas.RenderTransform>
        <TransformGroup>
            <MatrixTransform Matrix="
                1.30,   0.75,
               -0.50,   0.87,
              100.00, -10.00"
            />
        </TransformGroup>
    </Canvas.RenderTransform>

    <Ellipse
        Fill="LightGray"
        Stroke="Black"
        StrokeThickness="20"
        Width="200"
```

```
        Height="200"
    />
    <Rectangle
        Fill="Gray"
        Stroke="Black"
        StrokeThickness="20"
        Canvas.Left="100"
        Canvas.Top="100"
        Width="200"
        Height="200"
    />

</Canvas>
```

3D Transforms (New in Silverlight 3)

In Silverlight 3, you can set the `Projection` property to a `PlaneProjection` to rotate a group of elements in 3D as shown in Figure 3-28:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">
    <Canvas.Projection>
        <PlaneProjection RotationY="-60" CenterOfRotationY="50" />
    </Canvas.Projection>

    <Ellipse
        Fill="LightGray"
        Stroke="Black"
        StrokeThickness="20"
        Width="200"
        Height="200"
      Canvas.Top="50"
    />
    <Rectangle
        Fill="Gray"
        Stroke="Black"
        StrokeThickness="20"
        Canvas.Left="100"
        Canvas.Top="100"
        Width="200"
        Height="200"
    />

</Canvas>
```

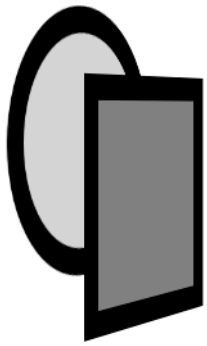


Figure 3-28
3D projection example

Each projection logically has its own camera. To position more than one object relative to the same perspective camera, position them all in the same place and use the `GlobalOffsetX`, `GlobalOffsetY`, and `GlobalOffsetZ` properties to move in the 3D world as shown in Figure 3-29:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

  <Rectangle
    Fill="Gray"
    Stroke="Black"
    StrokeThickness="20"
    Canvas.Left="200"
    Canvas.Top="100"
    Width="200"
    Height="200"
  >
    <Rectangle.Projection>
        <PlaneProjection
           GlobalOffsetX="-200"
           RotationY="-60"
           CenterOfRotationY="50"
        />
    </Rectangle.Projection>
  </Rectangle>

  <Rectangle
    Fill="Gray"
    Stroke="Black"
    StrokeThickness="20"
    Canvas.Left="200"
    Canvas.Top="100"
    Width="200"
    Height="200"
  >
    <Rectangle.Projection>
        <PlaneProjection  GlobalOffsetZ="-150"/>
    </Rectangle.Projection>
  </Rectangle>

  <Rectangle
```

```
    Fill="Gray"
    Stroke="Black"
    StrokeThickness="20"
    Canvas.Left="200"
    Canvas.Top="100"
    Width="200"
    Height="200"
  >
    <Rectangle.Projection>
        <PlaneProjection
           GlobalOffsetX="200"
           RotationY="60"
           CenterOfRotationY="50"
         />
    </Rectangle.Projection>
  </Rectangle>

</Canvas>
```

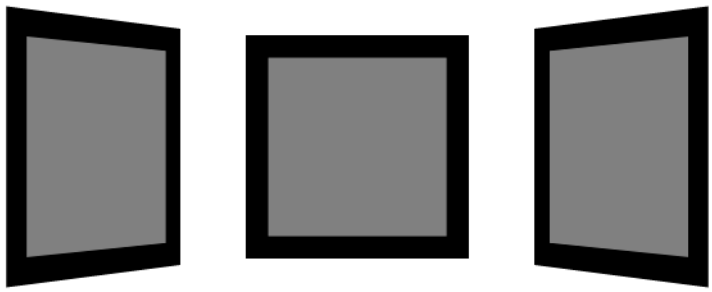


Figure 3-29
Position three rectangles in the same 3D projection camera

The global offset properties apply after the rotation property. You can also use the also a `LocalOffsetX`, `LocalOffsetY`, and `LocalOffsetZ` properties on the `PlaneProjection` object to apply an offset before the rotation.

Clipping

Clipping is the process of restricting the display area to a specified shape. To clip an element, set the `Clip` property as shown in the following listing:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">
    <Canvas.Clip>
        <EllipseGeometry
            Center="100,200"
            RadiusX="150"
            RadiusY="150"
        />
    </Canvas.Clip>

    <Ellipse
        Fill="LightGray"
```

```
        Stroke="Black"
        StrokeThickness="20"
        Width="200"
        Height="200"
    />
    <Rectangle
        Fill="Gray"
        Stroke="Black"
        StrokeThickness="20"
        Canvas.Left="100"
        Canvas.Top="100"
        Width="200"
        Height="200"
    />
</Canvas>
```

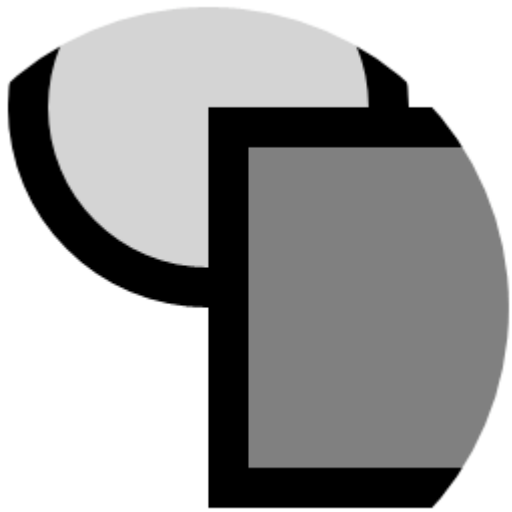


Figure 3-30
Clipping example

Performance Tip

A clipping operation is semantically equivalent to intersecting two geometries. Clipping a group of elements or a single shape does come with a significant performance penalty. You should avoid clipping when possible.

Opacity

Setting opacity on a brush or setting a transparent color on a brush will introduce alpha blending. In particular, if a brush contains a transparent color, the brush will blend its color with the content underneath using the following formula:

Color_destination = alpha*Color_source + (1-alpha)*Color_destination

The other form of opacity is setting the `Opacity` property on a `Canvas`. This operation is not equivalent to changing the opacity of each of the shapes within the `Canvas` element as demonstrated by Figure 3-31.

Performance Tip

Setting `Opacity` on a `Canvas` element will resolve occlusion first and then blend content. This process is significantly more expensive at run-time than blending individual primitives. If possible, you should set opacity on a brush, brush color, or a `Path` element for maximum performance.

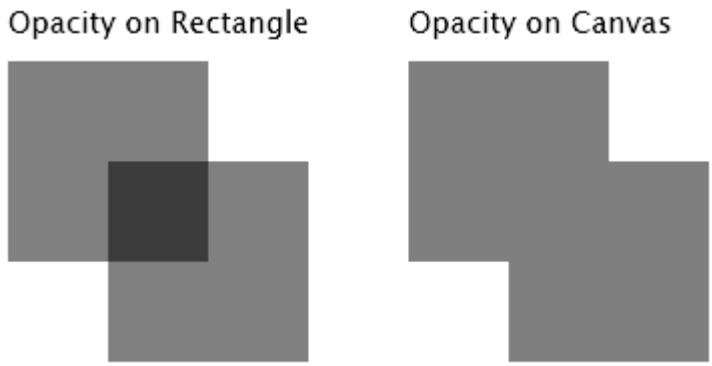


Figure 3-31
Canvas Opacity versus per path Opacity

OpacityMask

The `OpacityMask` property on a `UIElement` provides a mechanism to blend brush per pixel alpha information with the content of a `UIElement`. For example, the following XAML would produce the result shown in Figure 3-32:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">
    <Canvas.OpacityMask>
        <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
            <LinearGradientBrush.GradientStops>
                <GradientStop Color="Transparent" Offset="0"/>
                <GradientStop Color="White" Offset="1"/>
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </Canvas.OpacityMask>

    <Ellipse
        Fill="LightGray"
        Stroke="Black"
        StrokeThickness="20"
        Width="200"
        Height="200"
    />
```

```
    <Rectangle
        Fill="Gray"
        Stroke="Black"
        StrokeThickness="20"
        Canvas.Left="100"
        Canvas.Top="100"
        Width="200"
        Height="200"
    />

</Canvas>
```

Figure 3-32
OpacityMask example

Performance Tip

`OpacityMask` is computationally expensive at run-time. In some cases, it is faster to draw content on top that blends to the background instead of using the `OpacityMask`. For example, you can achieve the effect in Figure 3-32 with the following XAML:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Ellipse
        Fill="LightGray"
        Stroke="Black"
        StrokeThickness="20"
```

```
        Width="200"
        Height="200"
    />
    <Rectangle
        Fill="Gray"
        Stroke="Black"
        StrokeThickness="20"
        Canvas.Left="100"
        Canvas.Top="100"
        Width="200"
        Height="200"
    />

    <!-- simulate opacity mask effect with a rectangle on top -->

    <Rectangle Width="300" Height="300">
        <Rectangle.Fill>
            <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
                <LinearGradientBrush.GradientStops>
                    <GradientStop Color="White" Offset="0"/>
                    <GradientStop Color="Transparent" Offset="1"/>
                </LinearGradientBrush.GradientStops>
            </LinearGradientBrush>
        </Rectangle.Fill>
    </Rectangle>

</Canvas>
```

Under the Hood

Previous sections have discussed the graphics principles and the graphics API elements. This section goes deeper under the covers to describe how Silverlight draws XAML content and displays it in the browser window. Understanding this process will help you understand the Silverlight run-time performance characteristics. Furthermore, you will understand the problems solved by the run-time and the problems your application must solve.

In particular, this section will discuss:

- The draw loop process which takes changes to the graph of objects and draws it to an off screen back buffer
- The rasterization process that converts vector graphics primitives to pixels in an offscreen back buffer
- Performance optimizations such as incremental redraw, occlusion culling, and multi-core
- How the off screen back buffer gets displayed in the browser window

Draw Loop

Silverlight draws at a regular timer interval set by the `MaxFrameRate` property. On each tick of the timer, Silverlight will do the following:

1. Check for any changes to the properties of our graph of `Canvas` and `Shape` elements. If no changes exist, Silverlight does no further work for this timer tick.
2. Perform any pending layout operations. The layout chapter will discuss these layout operations further.
3. Gather rendering changes and prepare to rasterize them.
4. Incrementally rasterize the changes for the current timer tick. The graphics state at the current timer tick is also known as the current **frame**.
5. Notify the browser that a frame (or an incremental delta to an existing frame) is complete for display.

Performance Tip

One property of the draw loop is that nothing draws immediately after you make a change to the element tree. Consequently, profiling tools will not associate the cost of a drawing operation with the function that added those drawing primitives. To tune your performance, you should measure the maximum frame rate of your application during development. In particular, set the `MaxFrameRate` property to some value that is beyond what Silverlight can achieve and turn on the frame rate display as shown in the following JavaScript:

```
function loadHandler(sender, args)
{
   sender.settings.EnableFramerateCounter = true;
   sender.settings.MaxFrameRate = 10000;
}
```

During development, watch for content that drops the frame rate significantly, and consider specifying that content in an alternative form.

Rasterization

After the draw loop has identified which elements need to be redrawn, Silverlight converts those elements to a set of pixels in our off screen back buffer. The previous discussion of shapes described how to specify path outlines and a method of specifying the inside and the outside of the shape. However, the geometry describes an abstract infinite resolution outline of a shape and a screen has a finite number of pixels to color. Rasterization is the process of converting from a path outline to discrete pixels. This section describes how rasterization is accomplished.

The simplest method to convert geometry to pixels is a process called sampling. The sampling process uses a discrete number of sample points to convert from the infinite shape

description to pixels. For example, consider the simple sample pattern consisting of a uniform grid of sample points with one sample point per pixel. If the sample point is contained within the geometry, light up the pixel. If the sample point is not contained within the geometry, do not light the pixel. For example, the circle specified by the following XAML would light the pixels shown in Figure 3-33:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">

    <Ellipse
        Fill="Black"
        Width="15"
        Height="15"
    />

</Canvas>
```

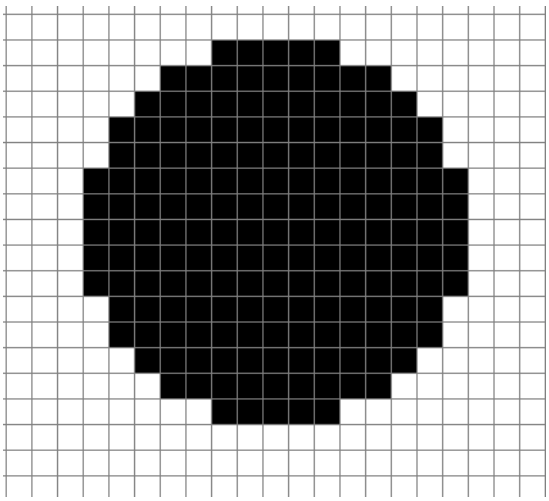


Figure 3-33
Sampling a circle

You may have noticed that the integer coordinates were located at the top left of the pixel and the sample points were in the center of a pixel. This convention enables a symmetrical curved surface specified on integer coordinates to produce a symmetrical rasterization. If the sample points were on integer coordinates instead, the ellipse would lose symmetry as shown in Figure 3-34.

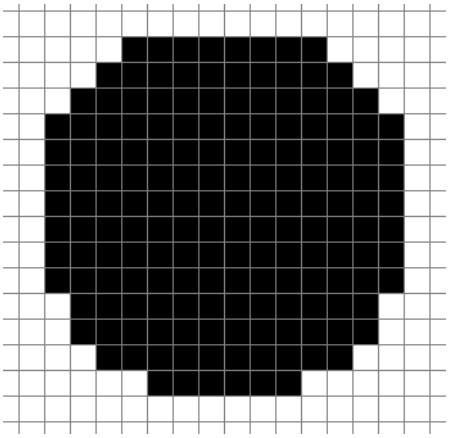


Figure 3-34
Sampling a circle with integer sample point coordinates

The rasterization shown in Figure 3-33 appears to have jagged edges. This jagged appearance is the consequence of aliasing. Aliasing is the loss of information that results from converting from a continuous curve to a discrete set of samples. Anti-aliasing is a term that refers to a technique that attempts to minimize aliasing artifacts.

The Silverlight anti-aliasing technique consists of sampling multiple times per pixel and applying a box filter to produce the final pixel color. Silverlight conceptually samples 64 times per pixel as shown in Figure 3-35. The box filter will average the contribution of all samples within a rectangle bordering the pixel to produce a final pixel color. If some partial number of samples is in the box, Silverlight applies transparency to blend smoothly with what is underneath the geometry as shown in Figure 3-36. This anti-aliasing technique produces a smooth transition from inside the shape to outside the shape along edges.

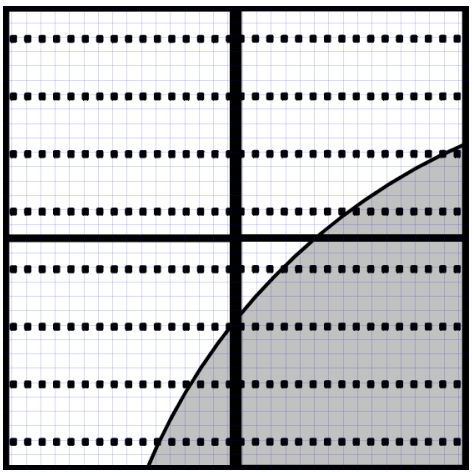


Figure 3-35
Anti-aliasing sampling pattern

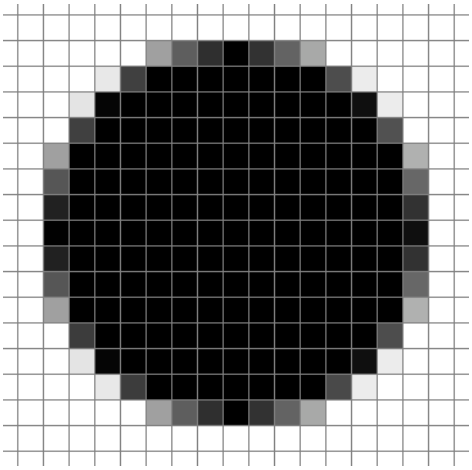


Figure 3-36
Anti-aliased rasterization

Technical Insight

You may be wondering why there are 16 samples per pixel in the x direction and only 4 samples per pixel in the y direction. The reason for picking this sample pattern is that horizontal resolution is critical to being able to render text clearly. Furthermore, horizontal resolution is computationally cheap and vertical resolution is computationally expensive. The 16x4 sampling pattern balances image quality and speed.

Instead of a box pattern, it is also possible to accumulate samples in a circular pattern, weight samples unevenly, or even have a sample pattern that extended far beyond a single pixel in size. In fact, all of these other algorithms generate better image quality than a box filter but typically render more slowly. The Silverlight high-resolution box filter is a choice made to achieve good rendering performance with reasonable image quality.

One artifact of anti-aliasing is a visible seam that sometimes results from drawing two adjacent shapes. For example, the following two rectangles that meet in the middle of a pixel would generate a seam:

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">
    <Rectangle
        Fill="Black"
        Width="100.5"
        Height="100.5"
    />
    <Rectangle
        Fill="Black"
        Canvas.Left="100.5"
        Width="100.5"
        Height="100.5"
```

```
    />
</Canvas>
```

The previous XAML will result in the rasterization shown in Figure 3-37. Notice the gap between the two rectangles. The rectangles joined perfectly in the input XAML, so why is there a seam in the rendered result?

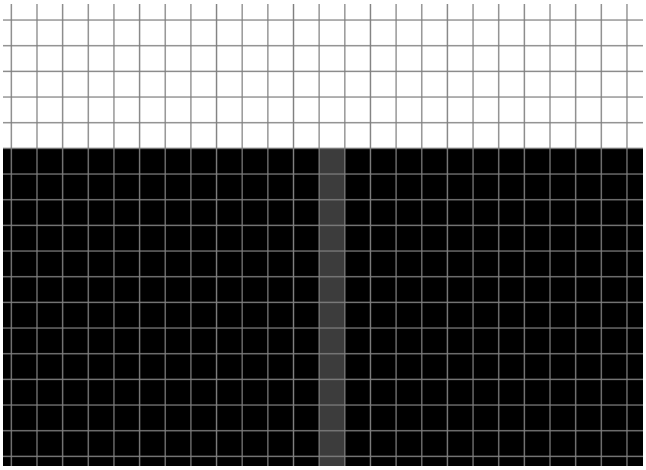


Figure 3-37
Anti-aliasing seam example

These seams are a result of the rasterization rules described in this section. Consider the rasterization process applied to pixel *X* shown in Figure 3-37. Rectangle *A* is covering exactly half the samples for pixel *X*. Silverlight will consequently draw that pixel of Rectangle *A* with 0.5 anti-aliasing alpha. Alpha is a term that refers to the transparency used to blend colors with a formula such as:

$$Color\_destination = alpha*Color\_source + (1 - alpha)*Color\_destination$$

In our example, *alpha=0.5*, *Color_source=Black*, and *Color_destination=White*. Blending along the edge of rectangle *A* will result in a destination color of:

$$0.5*Black + (1 - 0.5)*White = 0.5*White$$

Rectangle *B* also covers half its sample points. Silverlight will also blend pixel *X* of rectangle *B* with *alpha=0.5* to a background color of *0.5*White*. Consequently, the resulting color will be:

$$0.5*Black + (1 - 0.5)*(0.5White) = 0.25\ White.$$

The final pixel color has one quarter of the background color showing through as a visible seam.

Technical Insight

This result is an artifact of sampling each primitive independently. An alternative anti-aliasing mode is full screen anti-aliasing that processes all samples from all shapes simultaneously. However, Silverlight does not

currently use full screen anti-aliasing because it results in slower run-time performance.

To avoid these seams, you should snap edges to pixel boundaries as shown in Figure 3-38. Snapping will also produce a sharper edge between the two shapes. However, pixel snapping only removes seams if you align the shapes edges with the *x*-axis or the *y*-axis. For the rotated edges shown in Figure 3-39, snapping does not remove the artifact. For rotated edges, the common technique to avoid this seam is to overlap the edges so that the background is no longer visible.

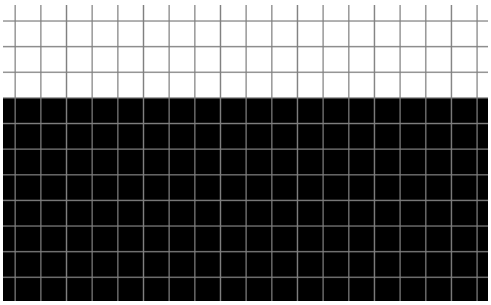


Figure 3-38
Pixel snapped rasterization

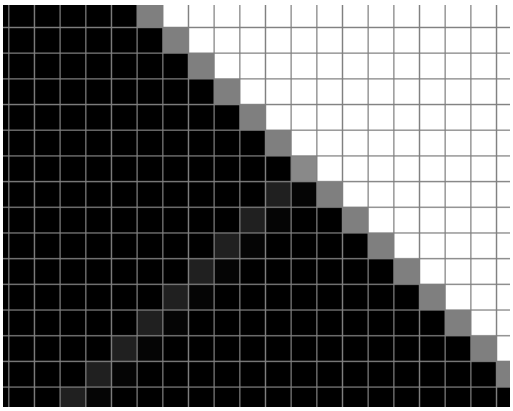


Figure 3-39
Seams with a rotated edge

Bilinear Filtering

The previous section discussed how Silverlight converts an arbitrary geometry to a set of pixels to fill. Silverlight then colors the filled pixels based on the brush specified. This process is straightforward for solid color brushes and gradient brushes. However, with image brushes, Silverlight must map from the destination pixels to the original image data, which may be at a different resolution. This section describes the mapping function used to achieve the image data stretch shown in Figure 3-40.

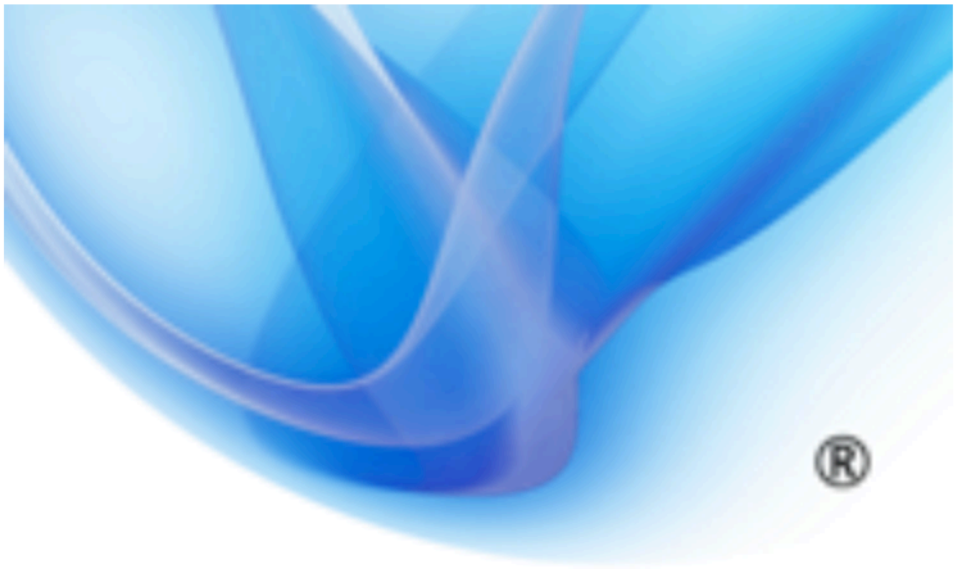


Figure 3-40
Image with bilinear filtering

Nearest neighbor is a simple image scaling function that will transform the destination pixel to an image bitmap position and pick the nearest pixel color. Nearest neighbor sampling generates ugly aliasing artifacts when the image is displayed with a scale or rotation as shown in Figure 3-41. You will notice jagged lines if you look at the picture closely.

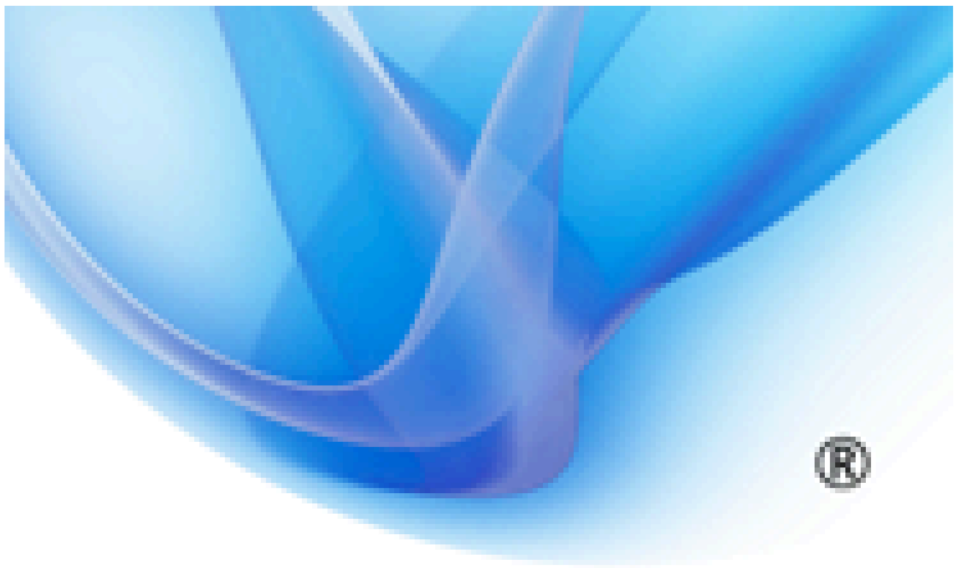


Figure 3-41
Image with nearest neighbor

Silverlight will use nearest neighbor sampling in the special case where the brush image data maps exactly onto centers of pixels. For rotated, scaled, or non-integer translated images, bilinear filtering is used to produce the result shown in Figure 3-40.

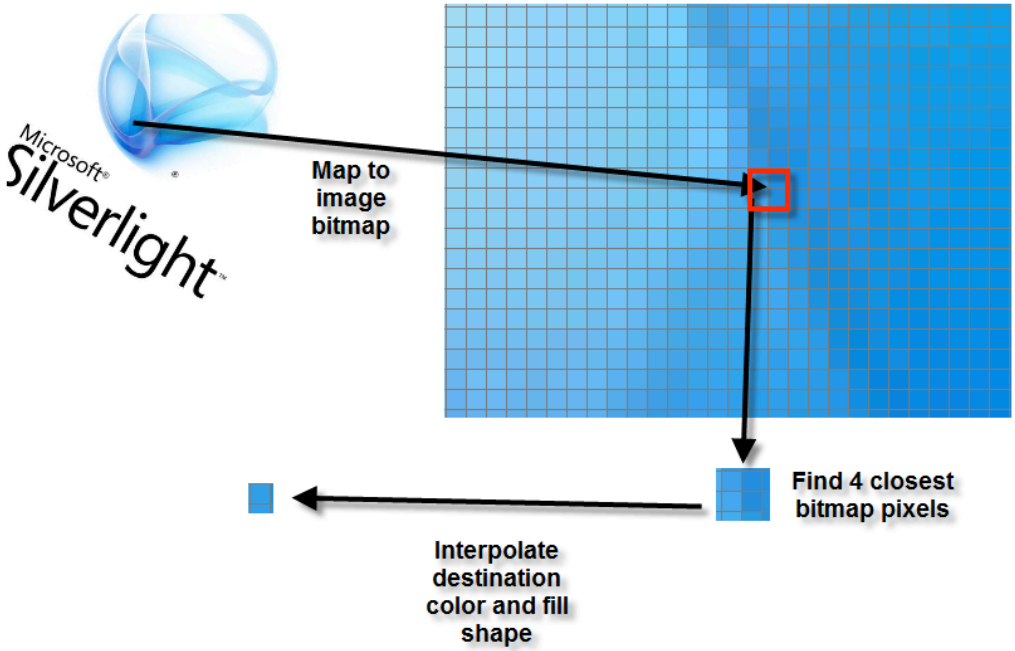


Figure 3-42
The bilinear filtering process

Bilinear filtering maps the screen position to a position *(u,v)* in image space. The bilinear filtering process will then interpolate a color from pixels *(floor(u), floor(v))*, *(floor(u)+1, floor(v))*, *(floor(u), floor(v)+1)*, and *(floor(u)+1, floor(v)+1)*. Figure 3-42 illustrates this process. Bilinear filtering generates good results for scales that are within a factor of two of the original image size. Figure 3-43 demonstrates the results of scaling an image in two sizes within reasonable limits.

Figure 3-43
Image scaling within good limits

With bilinear filtering, if you scale up an image significantly it will become blurry. Conversely, if you scale down an image significantly it will look aliased. Figure 3-44 shows examples of both these artifacts.

Figure 3-44
Image scaling extremes

New in Silverlight 3

There are a number of techniques for scaling down an image to produce a result better than Figure 3-44. However, these techniques can be computationally expensive and slow down your animations. Silverlight 3 adds support for mip-mapping that converts your image to a set of smaller images at various sizes using a better algorithm.

For example, if you have a 128x128 image, Silverlight will also generate copies at 64x64, 32x32, 16x16, 8x8, 4x4, 2x2, and 1x1 resolutions resized with high quality. When displaying the image at a particular scale, Silverlight will choose the closest resolution to the display size or even use multiple sizes at once when displaying in 3D.

This conversion happens as Silverlight is downloading and decoding your images and only adds a little time to the loading time of your application, but does not slow down animation speed.

Incremental Redraw

In addition to drawing static objects for a single frame, Silverlight must constantly redraw objects as they are changing. If an object moves from one position to another, it would be wasteful to redraw all the pixels on the screen. Instead, Silverlight marks the old position as needing a redraw and marks the new position as also needing a redraw. To simplify this marking algorithm, Silverlight uses the bounding box of a shape instead of the tight shape bounds.

For example, suppose the shape shown in the following XAML moves from position 0, 0 to position 100,100. Figure 3-45 shows the area that is redrawn.

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007">
```

```
    <Ellipse
        Fill="Black"
        Width="100"
        Height="100"
    />
</Canvas>
```

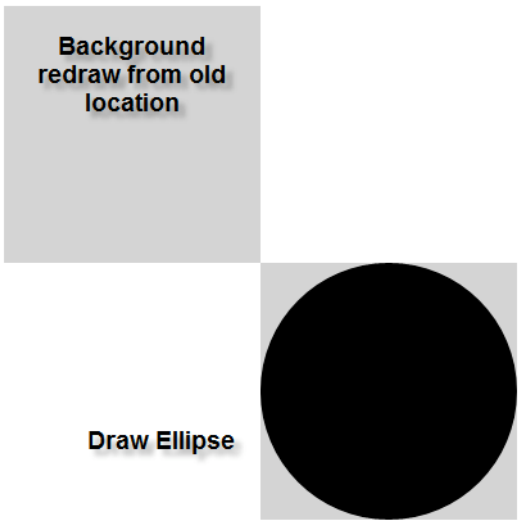


Figure 3-45
Incremental redraw regions

To view a visualization of these incremental redraw regions in an application, use the following JavaScript:

```
function loadHandler(sender, args)
{
   sender.settings.EnableRedrawRegions = true;
}
```

This visualization will blend a transparent color on top of any content drawn and cycles to a different color each frame. Consequently, any content that is flashing represents content that Silverlight is constantly redrawing. Any content that stabilizes on a single color has not changed for several frames.

Occlusion Culling

The most expensive operation in the draw loop is the rasterization process, which writes each of the destination pixels. For example, a full screen animation can consist of processing several hundred million pixels per second. Each of these pixels will apply at least one brush operation. If there are overlapping brushes, the computational requirements can multiply by a factor of 3 to 10.

As the graph of elements gets more complicated, it may no longer render at the desired frame rate. To optimize the rasterization process, Silverlight avoids brush operations for completely occluded brush pixels. For example, if you draw a full screen background and an almost full screen image, Silverlight computes all the image pixels and only those

background pixels not covered by the image itself. For complicated graphs of elements, this optimization can produce a 3-10x speedup.

Performance Tip

Occlusion culling only applies to brush pixel color optimizations. If a complicated geometry is behind a big opaque rectangle, the rasterizer will walk the geometry before it realizes that the pixel operations are not necessary. Consequently, it is still important to remove hidden content from the element tree for maximum performance.

Multi-core Rendering

Silverlight will take advantage of multiple CPU cores to produce faster rendering throughput. In particular, Silverlight will subdivide a frame into a set of horizontal bands and distribute the rasterization of those bands across CPU cores as shown in Figure 3-46. Currently, only the frame rasterization step and media operations run in parallel across CPU cores. Systems such as layout, control templating, application user code, and animation all run on a single thread. Consequently, you can determine if your application is rasterization bound by simply setting your framerate to 10000 frames per second and measuring your CPU usage percentage. If you achieve almost 100% CPU usage on a dual core machine, you are almost entirely rasterization bound. If you achieve 70% CPU usage on a dual core machine (at 10000 frames per second) that means that 30% of the work is not running in parallel.

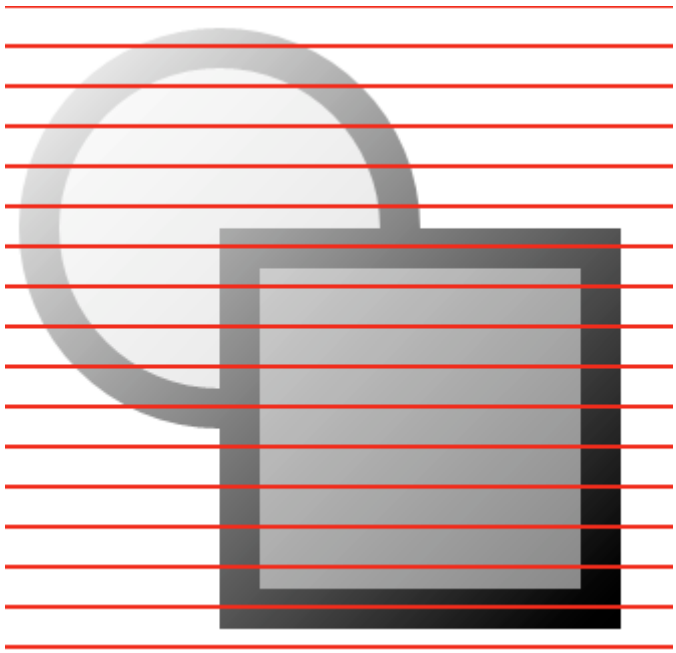


Figure 3-46
Dividing a scene for multi-core rendering

How Content Gets to the Screen

As previously discussed, the draw loop first draws a frame to an off screen back buffer and then it notifies the browser that the frame is ready for display. With `windowless=false` mode, Silverlight content goes to the screen without browser intervention on most operating systems. With `windowless=true`, the browser copies the off screen frame to its display area. This extra step is both slow and can result in visual tearing effects in a number of browsers. The worst mode of operation is when `windowless=true` is specified with a transparent color for the background of the control. The transparent color causes the web browser to redraw the content underneath the control each time any control content has changed.

Performance Tip

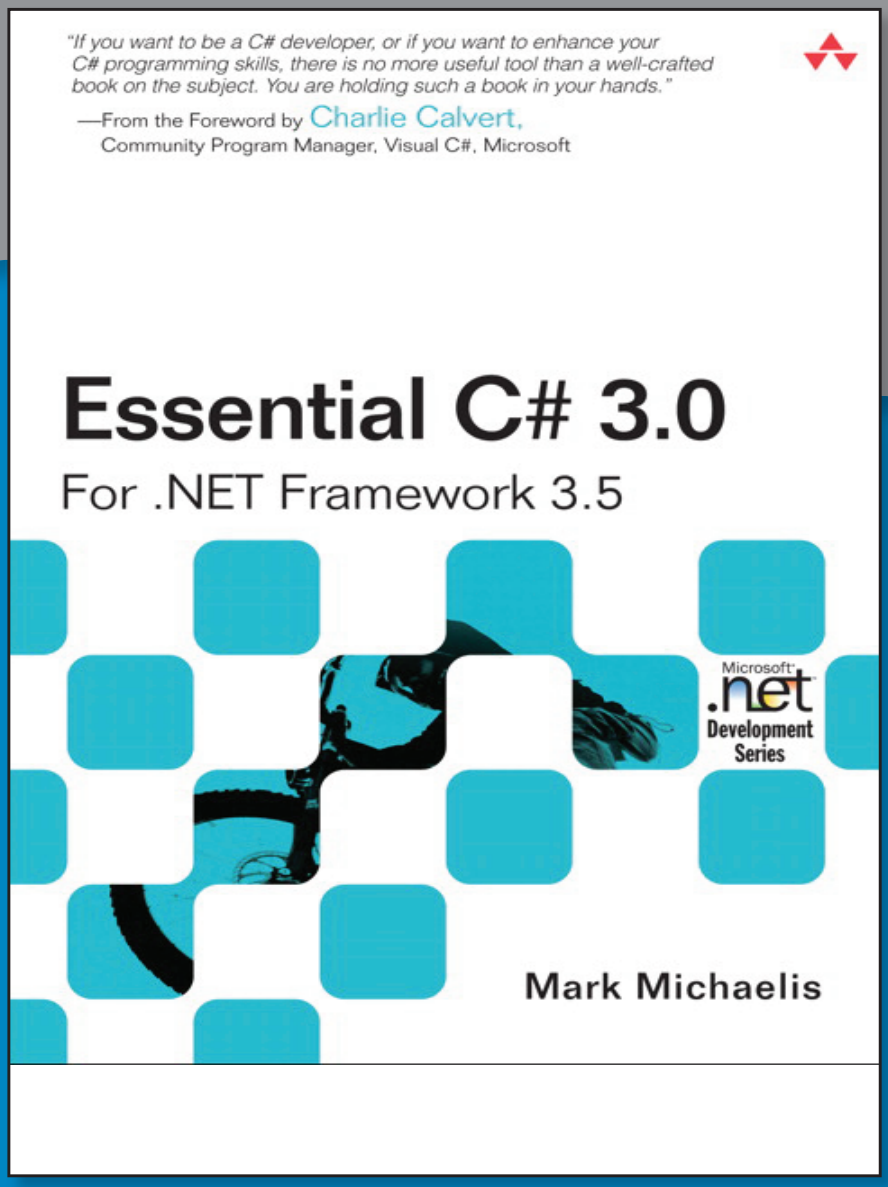
You should avoid using both a transparent background and `windowless=true` if possible.

Where Are We?

This chapter has described:

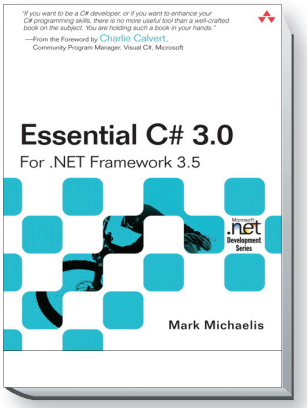
- The graphics system design principles
- The elements for displaying graphics
- The problems the Silverlight run-time solves under the hood and the problems your application must solve

In addition, you have learned a number of important performance optimization techniques for use with your application.



Mark Michaelis

BUY ME



Essential C# 3.0

For .NET Framework 3.5

Essential C# 3.0 is an extremely well-written and well-organized "no-fluff" guide to C# 3.0, which will appeal to programmers at all levels of experience with C#. This fully updated edition dives deep into the new features that are revolutionizing programming, with brand new chapters covering query expressions, lambda expressions, extension methods, collection interface extensions, standard query operators, and LINQ as a whole.

Author Mark Michaelis covers the C# language in depth, and each importantconstruct is illustrated with succinct, relevant code examples. (Complete code examples are available online.) Graphical "mind maps" at the beginning of each chapter show what material is covered and how each topic relates to the whole. Topics intended for beginners and advanced readers are clearly marked.

Following an introduction to C#, readers learn about

- C# primitive data types, value types, reference types, type conversions, and arrays
- Operators and control flow, loops, conditional logic, and sequential programming
- Methods, parameters, exception handling, and structured programming
- Classes, inheritance, structures, interfaces, and object-oriented programming
- Well-formed types, operator overloading, namespaces, and garbage collection
- Generics, collections, custom collections, and iterators
- Delegates and lambda expressions
- Standard query operators and query expressions
- LINQ: language integrated query
- Reflection, attributes, and declarative programming
- Threading, synchronization, and multithreaded patterns
- Interoperability and unsafe code
- The Common Language Infrastructure that underlies C#

Whether you are just starting out as a programmer, are an experienced developer looking to learn C#, or are a seasoned C# programmer interested in learning the new features of C# 3.0, **Essential C# 3.0** gives you just what you need to quickly get up and running writing C# applications.

AVAILABLE
- BOOK: 9780321533920
- SAFARI ONLINE Safari Books Online
- EBOOK: 0321580699
- KINDLE: 0321580680

About the Author

Mark Michaelis is an enterprise software architect at Itron Inc. In addition, Mark recently started intelliTechture, a software engineering and consulting company offering high-end consulting in Microsoft VSTS/TFS, BizTalk, SharePoint, and .NET 3.0. Mark also serves as a chief software architect and trainer for IDesign Inc.

Mark holds a B.A. in philosophy from the University of Illinois and an M.S. in computer science from the Illinois Institute of Technology. Mark was recently recognized as a Microsoft Regional Director. Starting in 1996, he has been a Microsoft MVP for C#, Visual Studio Team System, and the Windows SDK. He serves on several Microsoft software design review teams, including C#, the Connected Systems Division, and VSTS. Mark speaks at many developer conferences and has written numerous articles and books.

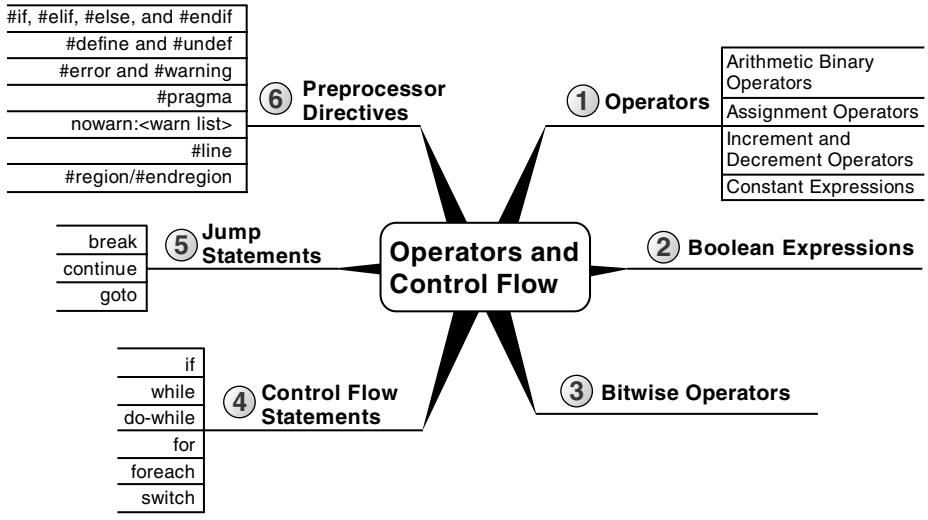
When not bonding with his computer, Mark is busy with his family or training for the Ironman. Mark lives in Spokane, Washington, with his wife Elisabeth, and three children, Benjamin, Hanna, and Abigail.

Addison Wesley

informit.com/aw

3
Operators and Control Flow

IN THIS CHAPTER, you will learn about operators and control flow statements. Operators provide syntax for performing different calculations or actions appropriate for the operands within the calculation. Control flow statements provide the means for conditional logic within a program or looping over a section of code multiple times. After introducing the `if` control flow statement, the chapter looks at the concept of Boolean expressions, which are embedded within many control flow statements, pointing



out that integers will not cast explicitly to `bool` and the advantages of this restriction. The chapter ends with a discussion of the primitive C# "preprocessor" and its accompanying directives.

Operators

Now that you have been introduced to the predefined data types (refer to Chapter 2), you can begin to learn more about how to use these data types in combination with operators to perform calculations. For example, you can make calculations on variables you have declared.

BEGINNER TOPIC

Operators

Operators specify operations within an expression, such as a mathematical expression, to be performed on a set of values, called **operands,** to produce a new value or result. For example, in Listing 3.1 there are two operands, the numbers `4` and `2`, that are combined using the subtraction operator, `-`. You assign the result to the variable `total`.

Listing 3.1: A Simple Operator Example

```
total = 4 - 2;
```

Operators are generally broken down into three categories: unary, binary, and ternary, corresponding to the number of operands 1, 2, and 3, respectively. This section covers some of the most basic unary and binary operators. Ternary operators appear later in the chapter.

Plus and Minus Unary Operators (+, –)

Sometimes you may want to change the sign of a numerical variable. In these cases, the unary minus operator (–) comes in handy. For example, Listing 3.2 changes the total current U.S. debt to a negative value to indicate that it is an amount owed.

Listing 3.2: Specifying Negative Values[1]

```
//National Debt to the Penny
decimal debt = -9202150370120.72M;
```

Using the minus operator is equivalent to multiplying a number by –1.

The unary plus operator (+) has no effect on a value. It is a superfluous addition to the C# language and was included for the sake of symmetry.

Arithmetic Binary Operators (+, –, *, /, %)

Binary operators require two operands in order to process an equation: a left-hand side operand and a right-hand side operand. Binary operators also require that the code assign the resultant value to avoid losing it.

Language Contrast: C++—Operator-Only Statements

Binary operators in C# require an assignment or call; they always return a new result. Neither operand in a binary operator expression can be modified. In contrast, C++ will allow a single statement, such as 4+5, to compile even without an assignment. In C#, call, increment, decrement, and new object expressions are allowed for operator-only statements.

The subtraction example in Listing 3.3 is an example of a binary operator—more specifically, an arithmetic binary operator. The operands appear on each side of the arithmetic operator and then the calculated value is assigned. The other arithmetic binary operators are addition (+), division (/), multiplication (*), and remainder (%; sometimes called the mod operator).

Listing 3.3: Using Binary Operators

```
class Division
{
  static void Main()
  {
      int numerator;
      int denominator;
```

1. As of January 15, 2008.

```
        int quotient;
        int remainder;

        System.Console.Write("Enter the numerator: ");
        numerator = int.Parse(System.Console.ReadLine());

        System.Console.Write("Enter the denominator: ");
        denominator = int.Parse(System.Console.ReadLine());

        quotient = numerator / denominator;
        remainder = numerator % denominator;

        System.Console.WriteLine(
            "{0} / {1} = {2} with remainder {3}",
            numerator, denominator, quotient, remainder);
    }
}
```

Output 3.1 shows the results of Listing 3.3.

OUTPUT 3.1:

```
Enter the numerator: 23
Enter the denominator: 3
23 / 3 = 7 with remainder 2
```

Note the order of associativity when using binary operators. The binary operator order is from left to right. In contrast, the assignment operator order is from right to left. On its own, however, associativity does not specify whether the division will occur before or after the assignment. The order of precedence defines this. The precedence for the operators used so far is as follows:

1) *, /, and %, 2) + and -, and 3) =

Therefore, you can assume that the statement behaves as expected, with the division and remainder operators occurring before the assignment.

If you forget to assign the result of one of these binary operators, you will receive the compile error shown in Output 3.2.

OUTPUT 3.2:

```
... error CS0201: Only assignment, call, increment, decrement,
and new object expressions can be used as a statement
```

BEGINNER TOPIC

Associativity and Order of Precedence

As with mathematics, programming languages support the concept of **associativity.** Associativity refers to how operands are grouped and, therefore, the order in which operators are evaluated. Given a single operator that appears more than once in an expression, associative operators will produce the same result regardless of the order in which they are evaluated. Binary operators such as + and – are associative because the order in which the operators are applied is not significant; `a+b+c` has the same result whether `a+b` is performed first or `b+c` is performed first.

Associativity applies only when all the operators are the same. When different operators appear within a statement, the **order of precedence** for those operators dictates which operators are evaluated first. Order of precedence, for example, indicates that the multiplication operator be evaluated before the plus operator in the expression `a+b*c`.

Using the Plus Operator with Strings

Operators can also work with types that are not numeric. For example, it is possible to use the plus operator to concatenate two or more strings, as shown in Listing 3.4.

LISTING 3.4: USING BINARY OPERATORS WITH NON-NUMERIC TYPES

```
class FortyTwo
{
  static void Main()
  {
      short windSpeed = 42;
      System.Console.WriteLine(
          "The original Tacoma Bridge in Washington\nwas "
          + "brought down by a "
          + windSpeed + " mile/hour wind.");
  }
}
```

Output 3.3 shows the results of Listing 3.4.

Output 3.3:

```
The original Tacoma Bridge in Washington
was brought down by a 42 mile/hour wind.
```

Because sentence structure varies among languages in different cultures, developers should be careful not to use the plus operator with strings that require localization. Composite formatting is preferred (refer to Chapter 1).

Using Characters in Arithmetic Operations

When introducing the `char` type in the last chapter, I mentioned that even though it stores characters and not numbers, the `char` type is an integer type. It can participate in arithmetic operations with other integer types. However, interpretation of the value of the `char` type is not based on the character stored within it, but rather on its underlying value. The digit `3`, for example, contains a Unicode value of `0x33` (hexadecimal), which in base 10 is `51`. The digit `4`, on the other hand, contains a Unicode value of `0x34`, or `52` in base 10. Adding `3` and `4` in Listing 3.5 results in a hexadecimal value of `0x167`, or `103` in base 10, which is equivalent to the letter `g`.

Listing 3.5: Using the Plus Operator with the `char` Data Type

```
int n = '3' + '4';
char c = (char)n;
System.Console.WriteLine(c);  // Writes out g.
```

Output 3.4 shows the results of Listing 3.5.

Output 3.4:

```
g
```

You can use this trait of character types to determine how far two characters are from one another. For example, the letter `f` is three characters away from the letter `c`. You can determine this value by subtracting the letter `c` from the letter `f`, as Listing 3.6 demonstrates.

Listing 3.6: Determining the Character Difference between Two Characters

```
int distance = 'f' - 'c';
System.Console.WriteLine(distance);
```

Output 3.5 shows the results of Listing 3.6.

Output 3.5:

```
3
```

Special Floating-Point Characteristics

The floating-point types, `float` and `double`, have some special characteristics, such as the way they handle precision. This section looks at some specific examples, as well as some unique floating-point type characteristics.

A float, with seven digits of precision, can hold the value 1,234,567 and the value 0.1234567. However, if you add these two floats together, the result will be rounded to 1234567, because the decimal portion of the number is past the seven significant digits that a float can hold. This type of rounding can become significant, especially with repeated calculations or checks for equality (see the upcoming Advanced Topic, Unexpected Inequality with Floating-Point Types).

Note that inaccuracies can occur with a simple assignment, such as `double number = 4.2F`. Since the double can hold a more accurate value than the `float` can store, the C# compiler will actually evaluate this expression to `double number = 4.1999998092651367;`. `4.1999998092651367` is 4.2 as a `float`, but not quite 4.2 when represented as a `double`.

ADVANCED TOPIC

Unexpected Inequality with Floating-Point Types

The inaccuracies of floats can be very disconcerting when comparing values for equality, since they can unexpectedly be unequal. Consider Listing 3.7.

Listing 3.7: Unexpected Inequality Due to Floating-Point Inaccuracies

```
decimal decimalNumber = 4.2M;
double doubleNumber1 = 0.1F * 42F;
```

```
double doubleNumber2 = 0.1D * 42D;
float floatNumber = 0.1F * 42F;

Trace.Assert(decimalNumber != (decimal)doubleNumber1);
// Displays: 4.2 != 4.20000006258488
System.Console.WriteLine(
    "{0} != {1}", decimalNumber, (decimal)doubleNumber1);

Trace.Assert((double)decimalNumber != doubleNumber1);
// Displays: 4.2 != 4.20000006258488
System.Console.WriteLine(
    "{0} != {1}", (double)decimalNumber, doubleNumber1);

Trace.Assert((float)decimalNumber != floatNumber);
// Displays: (float)4.2M != 4.2F
System.Console.WriteLine(
    "(float){0}M != {1}F",
    (float)decimalNumber, floatNumber);

Trace.Assert(doubleNumber1 != (double)floatNumber);
// Displays: 4.20000006258488 != 4.20000028610229
System.Console.WriteLine(
    "{0} != {1}", doubleNumber1, (double)floatNumber);

Trace.Assert(doubleNumber1 != doubleNumber2);
// Displays: 4.20000006258488 != 4.2
System.Console.WriteLine(
    "{0} != {1}", doubleNumber1, doubleNumber2);

Trace.Assert(floatNumber != doubleNumber2);
// Displays: 4.2F != 4.2D
System.Console.WriteLine(
    "{0}F != {1}D", floatNumber, doubleNumber2);

Trace.Assert((double)4.2F != 4.2D);
// Display: 4.19999980926514 != 4.2
System.Console.WriteLine(
    "{0} != {1}", (double)4.2F, 4.2D);

Trace.Assert(4.2F != 4.2D);
// Display: 4.2F != 4.2D
System.Console.WriteLine(
    "{0}F != {1}D", 4.2F, 4.2D);
```

Output 3.6 shows the results of Listing 3.7.

The `Assert()` methods are designed to display a dialog whenever the parameter evaluates for false. However, all of the `Assert()` statements in this code listing will evaluate to true. Therefore, in spite of the apparent

Output 3.6:

```
4.2 != 4.2000000625848B
4.2 != 4.2000000625848B
(float)4.2M != 4.2F
4.2000000625848B != 4.2000002861022D
4.2000000625848B != 4.2
4.2F != 4.2D
4.1999998092651D != 4.2
4.2F != 4.2D
```

equality of the values in the code listing, they are in fact not equivalent due to the inaccuracies of a `float`. Furthermore, there is not some compounding rounding error. The C# compiler performs the calculations instead of the runtime. Even if you simply assign `4.2F` rather than a calculation, the comparisons will remain unequal.

To avoid unexpected results caused by the inaccuracies of floating-point types, developers should avoid using equality conditionals with these types. Rather, equality evaluations should include a tolerance. One easy way to achieve this is to subtract one value (operand) from the other and then evaluate whether the result is less than the maximum tolerance. Even better is to use the decimal type in place of the float type.

You should be aware of some additional unique floating-point characteristics as well. For instance, you would expect that dividing an integer by zero would result in an error, and it does with precision data types such as `int` and `decimal`. `float` and `double`, however, allow for certain special values. Consider Listing 3.8, and its resultant output, Output 3.7.

Listing 3.8: Dividing a Float by Zero, Displaying NaN

```
float n=0f;
// Displays: NaN
System.Console.WriteLine(n / 0);
```

Output 3.7:

```
NaN
```

In mathematics, certain mathematical operations are undefined. In C#, the result of dividing 0F by the value 0 results in "Not a Number," and all

attempts to print the output of such a number will result in `NaN`. Similarly, taking the square root of a negative number (`System.Math.Sqrt(-1)`) will result in `NaN`.

A floating-point number could overflow its bounds as well. For example, the upper bound of a `float` type is 3.4E38. Should the number overflow that bound, the result would be stored as "positive infinity" and the output of printing the number would be `Infinity`. Similarly, the lower bound of a `float` type is –3.4E38, and assigning a value below that bound would result in "negative infinity," which would be represented by the string `-Infinity`. Listing 3.9 produces negative and positive infinity, respectively, and Output 3.8 shows the results.

Listing 3.9: Overflowing the Bounds of a `float`

```
// Displays: -Infinity
System.Console.WriteLine(-1f / 0);
// Displays: Infinity
System.Console.WriteLine(3.402823E+38f * 2f);
```

Output 3.8:

```
-Infinity
Infinity
```

Further examination of the floating-point number reveals that it can contain a value very close to zero, without actually containing zero. If the value exceeds the lower threshold for the `float` or `double` type, then the value of the number can be represented as "negative zero" or "positive zero," depending on whether the number is negative or positive, and is represented in output as `-0` or `0`.

Parenthesis Operator

The parenthesis operator allows you to group operands and operators so that they are evaluated together. This is important because it provides a means of overriding the default order of precedence. For example, the following two expressions evaluate to something completely different:

```
(60 / 10) * 2
60 / (10 * 2)
```

The first expression is equal to 12; the second expression is equal to 3. In both cases, the parentheses affect the final value of the expression.

Sometimes the parenthesis operator does not actually change the result, because the order-of-precedence rules apply appropriately. However, it is often still a good practice to use parentheses to make the code more readable. This expression, for example:

```
fahrenheit = (celsius * 9 / 5) + 32;
```

is easier to interpret confidently at a glance than this one is:

```
fahrenheit = celsius * 9 / 5 + 32;
```

Developers should use parentheses to make code more readable, disambiguating expressions explicitly instead of relying on operator precedence.

Assignment Operators (+=, -=, *=, /=, %=)

Chapter 1 discussed the simple assignment operator, which places the value of the right-hand side of the operator into the variable on the left-hand side. Other assignment operators combine common binary operator calculations with the assignment operator. Take Listing 3.10, for example.

Listing 3.10: Common Increment Calculation

```
int x;
x = x + 2;
```

In this assignment, you first calculate the value of `x + 2` and then you assign the calculated value back to `x`. Since this type of operation is relatively frequent, an assignment operator exists to handle both the calculation and the assignment with one operator. The `+=` operator increments the variable on the left-hand side of the operator with the value on the right-hand side of the operator, as shown in Listing 3.11.

Listing 3.11: Using the += Operator

```
int x;
x += 2;
```

This code, therefore, is equivalent to Listing 3.10.

Numerous other combination assignment operators exist to provide similar functionality. You can use the assignment operator in conjunction with not only addition, but also subtraction, multiplication, division, and the remainder operators, as Listing 3.12 demonstrates.

Listing 3.12: Other Assignment Operator Examples

```
x -= 2;
x /= 2;
x *= 2;
x %= 2;
```

Increment and Decrement Operators (++, --)

C# includes special operators for incrementing and decrementing counters. The **increment operator,** `++`, increments a variable by one each time it is used. In other words, all of the code lines shown in Listing 3.13 are equivalent.

Listing 3.13: Increment Operator

```
spaceCount = spaceCount + 1;
spaceCount += 1;
spaceCount++;
```

Similarly, you can also decrement a variable by one using the **decrement operator,** `--`. Therefore, all of the code lines shown in Listing 3.14 are also equivalent.

Listing 3.14: Decrement Operator

```
lines = lines - 1;
lines -= 1;
lines--;
```

BEGINNER TOPIC

A Decrement Example in a Loop

The increment and decrement operators are especially prevalent in loops, such as the `while` loop described later in the chapter. For example, Listing 3.15 uses the decrement operator in order to iterate backward through each letter in the alphabet.

Listing 3.15: Displaying Each Character's ASCII Value in Descending Order

```
char current;
int asciiValue;

// Set the initial value of current.
current='z';

do
{
  // Retrieve the ASCII value of current.
  asciiValue = current;
  System.Console.Write("{0}={1}\t", current, asciiValue);

  // Proceed to the previous letter in the alphabet;
  current--;
}
while(current>='a');
```

Output 3.9 shows the results of Listing 3.15.

Output 3.9:

```
z=122   y=121   x=120   w=119   v=118   u=117   t=116   s=115   r=114
q=113   p=112   o=111   n=110   m=109   l=108   k=107   j=106   i=105
h=104   g=103   f=102   e=101   d=100   c=99    b=98    a=97
```

The increment and decrement operators are used to count how many times to perform a particular operation. Notice also that in this example, the increment operator is used on a character (`char`) data type. You can use increment and decrement operators on various data types as long as some meaning is assigned to the concept of "next" or "previous" for that data type.

Just as with the assignment operator, the increment operator also returns a value. In other words, it is possible to use the assignment operator simultaneously with the increment or decrement operator (see Listing 3.16 and Output 3.10).

Listing 3.16: Using the Post-Increment Operator

```
int count;
int result;
count = 0;
result = count++;
```

```
System.Console.WriteLine("result = {0} and count = {1}",
  result, count);
```

Output 3.10:

```
result = 0 and count = 1
```

You might be surprised that `count` is assigned to `result` *before* it is incremented. This is why `result` ends up with a value of `0` even though `count` ends up with a value of `1`.

If you want the increment or decrement operator to take precedence over the assignment operator and to execute before assigning the value, you need to place the operator before the variable being incremented, as shown in Listing 3.17.

Listing 3.17: Using the Pre-Increment Operator

```
int count;
int result;
count = 0;
result = ++count;
System.Console.WriteLine("result = {0} and count = {1}",
  result, count);
```

Output 3.11 shows the results of Listing 3.17.

Output 3.11:

```
result = 1 and count = 1
```

Where you place the increment or decrement operator determines the order of operations, which affects how the code functions. If the increment or decrement operator appears before the operand, the value returned will be the new value. If `x` is `1`, then `++x` will return `2`. However, if a postfix operator is used, `x++`, the value returned by the expression will still be `1`. Regardless of whether the operator is postfix or prefix, the resultant value of `x` will be different. The difference between prefix and postfix behavior appears in Listing 3.18. The resultant output is shown in Output 3.12.

Listing 3.18: Comparing the Prefix and Postfix Increment Operators

```
class IncrementExample
{
  public static void Main()
  {
      int x;

      x = 1;
      // Display 1, 2.
      System.Console.WriteLine("{0}, {1}, {2}", x++, x++, x);
      // x now contains the value 3.

      // Display 4, 5.
      System.Console.WriteLine("{0}, {1}, {2}", ++x, ++x, x);
      // x now contains the value 5.
      // ...
  }
}
```

Output 3.12:

```
1, 2, 3
4, 5, 5
```

As Listing 3.18 demonstrates, where the increment and decrement operators appear relative to the operand can affect the result returned from the operator. Pre-increment/decrement operators return the result after incrementing/decrementing the operand. Post-increment/decrement operators return the result before changing the operand. Developers should use caution when embedding these operators in the middle of a statement. When in doubt as to what will happen, use these operators independently, placing them within their own statements. This way, the code is also more readable and there is no mistaking the intention.

ADVANCED TOPIC

Thread-Safe Incrementing and Decrementing

In spite of the brevity of the increment and decrement operators, these operators are not atomic. A thread context switch can occur during the execution of the operator and can cause a race condition. Instead of using a

lock statement to prevent the race condition, the System.Threading.Interlocked class includes the thread-safe methods Increment() and Decrement(). These methods rely on processor functions for performing fast thread-safe increments and decrements.

Constant Expressions (const)

The previous chapter discussed literal values, or values embedded directly into the code. It is possible to combine multiple literal values in a **constant expression** using operators. By definition, a constant expression is one that the C# compiler can evaluate at compile time (instead of calculating it when the program runs). For example, the number of seconds in a day can be assigned as a constant expression whose result can then be used in other expressions.

The const keyword in Listing 3.19 locks the value at compile time. Any attempt to modify the value later in the code results in a compile error.

Listing 3.19: Declaring a Constant

```
// ...
public long Main()
{
    const int secondsPerDay = 60 * 60 * 24;
    const int secondsPerWeek = secondsPerDay * 7;

    // ...
}
```

Constant Expression

Constant

Note that even the value assigned to secondsPerWeek is a constant expression, because the operands in the expression are also constants, so the compiler can determine the result.

Introducing Flow Control

Later in this chapter is a code listing (Listing 3.42) that shows a simple way to view a number in its binary form. Even such a simple program, however, cannot be written without using control flow statements. Such statements control the execution path of the program. This section discusses how to change the order of statement execution based on conditional checks. Later, you will learn how to execute statement groups repeatedly through loop constructs.

TABLE 3.1: CONTROL FLOW STATEMENTS

STATEMENT	GENERAL SYNTAX STRUCTURE	EXAMPLE

`if` statement	`if(`*boolean-expression*`)`
	  embedded-statement
		`if (input == "quit")`
		`{`
		`  System.Console.WriteLine(`
		`  "Game end");`
		`  return;`
		`}`

	`if(`*boolean-expression*`)`
	  embedded-statement
	`else`
	  embedded-statement
		`if (input == "quit")`
		`{`
		`  System.Console.WriteLine(`
		`  "Game end");`
		`  return;`
		`}`
		`else`
		`  GetNextMove();`

`while` statement	`while(`*boolean-expression*`)`
	  embedded-statement
		`while(count < total)`
		`{`
		`  System.Console.WriteLine(`
		`  "count = {0}", count);`
		`  count++;`
		`}`

`do while` statement	`do`
	  embedded-statement
	`while(`*boolean-expression*`);`
		`do`
		`{`
		`  System.Console.WriteLine(`
		`  "Enter name:");`
		`  input =`
		`  System.Console.ReadLine();`
		`}`
		`while(input != "exit");`

Continues

TABLE 3.1: CONTROL FLOW STATEMENTS *(Continued)*

STATEMENT	GENERAL SYNTAX STRUCTURE	EXAMPLE

for statement	`for(for-initializer;`
	`  boolean-expression;`
	`  for-iterator)`
	`  embedded-statement`
		`for (int count = 1;`
		`  count <= 10;`
		`  count++)`
		`{`
		`  System.Console.WriteLine(`
		`  "count = {0}", count);`
		`}`

foreach statement	`foreach(type identifier in`
	`  expression)`
	`  embedded-statement`
		`foreach (char letter in email)`
		`{`
		`  if(!insideDomain)`
		`  {`
		`  if (letter == '@')`
		`  {`
		`  insideDomain = true;`
		`  }`
		`  continue;`
		`  }`
		`  System.Console.Write(`
		`  letter);`
		`}`

continue statement	`continue;`	

TABLE 3.1: CONTROL FLOW STATEMENTS *(Continued)*

STATEMENT	GENERAL SYNTAX STRUCTURE	EXAMPLE

switch statement	**switch**(governing-type-expression)
	{
	  ...
	  case const-expression:
	  statement-list
	  jump-statement
	  default:
	  statement-list
	  jump-statement
	}
		switch(*input*)
		{
		  case *"exit":*
		  case *"quit":*
		  System.Console.WriteLine(
		  "Exiting app....");
		  break;
		  case *"restart":*
		  Reset();
		  goto case *"start";*
		  case *"start":*
		  GetMove();
		  break;
		  default:
		  System.Console.WriteLine(
		  input);
		  break;
		}

break statement	**break;**	
goto statement	**goto** *identifier*;	
	goto case *const-expression*;	
	goto default;	

A summary of the control flow statements appears in Table 3.1. Note that the General Syntax Structure column indicates common statement use, not the complete lexical structure.
An `embedded-statement` in Table 3.1 corresponds to any statement, including a code block (but not a declaration statement or a label).

Each C# control flow statement in Table 3.1 appears in the tic-tac-toe program found in Appendix B. The program displays the tic-tac-toe board, prompts each player, and updates with each move.

The remainder of this chapter looks at each statement in more detail. After covering the `if` statement, it introduces code blocks, scope, Boolean expressions, and bitwise operators before continuing with the remaining control flow statements. Readers who find the table familiar because of C#'s similarities to other languages can jump ahead to the section titled C# Preprocessor Directives or skip to the Summary section at the end of the chapter.

`if` Statement

The `if` statement is one of the most common statements in C#. It evaluates a **Boolean expression** (an expression that returns a Boolean), and if the result is `true`, the following statement (or block) is executed. The general form is as follows:

```
if(boolean-expression)
  true-statement
[else
  false-statement]
```

There is also an optional `else` clause for when the Boolean expression is false. Listing 3.20 shows an example.

Listing 3.20: `if/else` Statement Example

```
class TicTacToe      // Declares the TicTacToe class.
{
  static void Main() // Declares the entry point of the program.
  {
      string input;

      // Prompt the user to select a 1- or 2- player game.
      System.Console.Write (
          "1 - Play against the computer\n" +
```

```
            "2 - Play against another player.\n" +
            "Choose:"
        );
        input = System.Console.ReadLine();

        if(input=="1")
            // The user selected to play the computer.
            System.Console.WriteLine(
                "Play against computer selected.");
        else
            // Default to 2 players (even if user didn't enter 2).
            System.Console.WriteLine(
                "Play against another player.");
    }
}
```

In Listing 3.20, if the user enters `1`, the program displays `"Play against computer selected."`. Otherwise, it displays `"Play against another player."`.

Nested `if`

Sometimes code requires multiple `if` statements. The code in Listing 3.21 first determines whether the user has chosen to exit by entering a number less than or equal to `0`; if not, it checks whether the user knows the maximum number of turns in tic-tac-toe.

Listing 3.21: Nested `if` Statements

```
1   class TicTacToeTrivia
2   {
3     static void Main()
4     {
5         int input;    // Declare a variable to store the input.
6
7         System.Console.Write(
8             "What is the maximum number " +
9             "of turns in tic-tac-toe?" +
10            "(Enter 0 to exit.): ");
11
12        // int.Parse() converts the ReadLine()
13        // return to an int data type.
14        input = int.Parse(System.Console.ReadLine());
15
16        if (input <= 0)
17                // Input is less than or equal to 0.
18            System.Console.WriteLine("Exiting...");
19        else
20            if (input < 9)
21                // Input is less than 9.
22                System.Console.WriteLine(
23                    "Tic-tac-toe has more than {0}" +
24                    " maximum turns.", input);
```

```
25              else
26                  if(input>9)
27                      // Input is greater than 9.
28                      System.Console.WriteLine(
29                          "Tic-tac-toe has fewer than {0}" +
30                          " maximum turns.", input);
31                  else
32                      // Input equals 9.
33                      System.Console.WriteLine(
34                          "Correct, " +
35                          "tic-tac-toe has a max. of 9 turns.");
36      }
37  }
```

Output 3.13 shows the results of Listing 3.21.

OUTPUT 3.13:

```
What's the maximum number of turns in tic-tac-toe? (Enter 0 to exit.): 9
Correct, tic-tac-toe has a max. of 9 turns.
```

Assume the user enters `9` when prompted at line 14. Here is the execution path:

1. *Line 16:* Check if input is less than 0. Since it is not, jump to line 20.
2. *Line 20:* Check if input is less than 9. Since it is not, jump to line 26.
3. *Line 26:* Check if input is greater than 9. Since it is not, jump to line 33.
4. *Line 33:* Display that the answer was correct.

Listing 3.21 contains nested `if` statements. To clarify the nesting, the lines are indented. However, as you learned in Chapter 1, whitespace does not affect the execution path. Without indenting and without newlines, the execution would be the same. The code that appears in the nested `if` statement in Listing 3.22 is equivalent.

LISTING 3.22: `if/else` FORMATTED SEQUENTIALLY

```
if (input < 0)
  System.Console.WriteLine("Exiting...");
else if (input < 9)
  System.Console.WriteLine(
      "Tic-tac-toe has more than {0}" +
      " maximum turns.", input);
else if(input>9)
```

```
    System.Console.WriteLine(
        "Tic-tac-toe has less than {0}" +
        " maximum turns.", input);
else
    System.Console.WriteLine(
        "Correct, tic-tac-toe has a maximum of 9 turns.");
```

Although the latter format is more common, in each situation, use the format that results in the clearest code.

Code Blocks ({})

In the previous `if` statement examples, only one statement follows `if` and `else`, a single `System.Console.WriteLine()`, similar to Listing 3.23.

Listing 3.23: `if` Statement with No Code Block

```
if(input<9)
    System.Console.WriteLine("Exiting");
```

However, sometimes you might need to execute multiple statements. Take, for example, the highlighted code block in the radius calculation in Listing 3.24.

Listing 3.24: `if` Statement Followed by a Code Block

```
class CircleAreaCalculator
{
  static void Main()
  {
      double radius;  // Declare a variable to store the radius.
      double area;    // Declare a variable to store the area.

      System.Console.Write("Enter the radius of the circle: ");

      // double.Parse converts the ReadLine()
      // return to a double.
      radius = double.Parse(System.Console.ReadLine());

      if(radius>=0)
      {
          // Calculate the area of the circle.
          area = 3.14*radius*radius;
          System.Console.WriteLine(
              "The area of the circle is: {0}", area);
      }
```

```
        else
        {
            System.Console.WriteLine(
                "{0} is not a valid radius.", radius);
        }
    }
}
```

Output 3.14 shows the results of Listing 3.24.

Output 3.14:

```
Enter the radius of the circle: 3
The area of the circle is: 28.26
```

In this example, the `if` statement checks whether the `radius` is positive. If so, the area of the circle is calculated and displayed; otherwise, an invalid radius message is displayed.

Notice that in this example, two statements follow the first `if`. However, these two statements appear within curly braces. The curly braces combine the statements into a single unit called a **code block.**

If you omit the curly braces that create a code block in Listing 3.24, only the statement immediately following the Boolean expression executes conditionally. Subsequent statements will execute regardless of the `if` statement's Boolean expression. The invalid code is shown in Listing 3.25.

Listing 3.25: Relying on Indentation, Resulting in Invalid Code

```
if(radius>=0)
  area = 3.14*radius*radius;
  System.Console.WriteLine(    // Error!! Needs code block.
      "The area of the circle is: {0}", area);
```

In C#, indentation is for code readability only. The compiler ignores it, and therefore, the previous code is semantically equivalent to Listing 3.26.

Listing 3.26: Semantically Equivalent to Listing 3.25

```
if(radius>=0)
{
  area = 3.14*radius*radius;
}
```

```
System.Console.WriteLine(    // Error!! Place within code block.
  "The area of the circle is: {0}", area);
```

Programmers should take great care to avoid subtle bugs such as this, perhaps even going so far as to always include a code block after a control flow statement, even if there is only one statement.

ADVANCED TOPIC

Math Constants

In Listing 3.25 and Listing 3.26, the value of pi as 3.14 was hardcoded—a crude approximation at best. There are much more accurate definitions for pi and E in the `System.Math` class. Instead of hardcoding a value, code should use `System.Math.PI` and `System.Math.E`.

Scope

Scope is the hierarchical context bound by a code block or language construct. C# prevents two declarations with the same name declared in the same scope. For example, it is not possible to define two local variables in the same code block with the same name; the code block bounds the scope. Similarly, it is not possible to define two methods called `Main()` within the same class.

Scope is hierarchical because it is not possible to define a local variable directly within a method and then to define a new variable with the same name inside an `if` block of the same method. The scope of the initial variable declaration spans the scope of all code blocks defined within the method. However, a variable declared within the `if` block will not be in the same scope as a variable defined within the `else` block. Furthermore, the same local variable name can be used within another method because the method bounds the scope of the local variable.

Scope restricts accessibility. A local variable, for example, is not accessible outside its defining method. Similarly, code that defines a variable in an `if` block makes the variable inaccessible outside the `if` block, even while still in the same method. In Listing 3.27, defining a message inside

the `if` statement restricts its scope to the statement only. To avoid the error, you must declare the string outside the `if` statement.

Listing 3.27: Variables Inaccessible Outside Their Scope

```
class Program
{
   static void Main(string[] args)
   {
       int playerCount;
       System.Console.Write(
           "Enter the number of players (1 or 2):");
       playerCount = int.Parse(System.Console.ReadLine());
       if (playerCount != 1 && playerCount != 2)
       {
           string message =
               "You entered an invalid number of players.";
       }
       else
       {
           // ...
       }
       // Error:  message is not in scope.
       Console.WriteLine(message);
   }
}
```

Output 3.15 shows the results of Listing 3.27.

Output 3.15:

```
...

...\Program.cs(18,26): error CS0103: The name 'message' does not exist
in the current context
```

Boolean Expressions

The portion of the `if` statement within parentheses is the **Boolean expression,** sometimes referred to as a **conditional.** In Listing 3.28, the Boolean expression is highlighted.

Listing 3.28: Boolean Expression

```
if(input < 9)
{
    // Input is less than 9.
```

```
        System.Console.WriteLine(
            "Tic-tac-toe has more than {0}" +
             " maximum turns.", input);
    }
    // ...
```

Boolean expressions appear within many control flow statements. The key characteristic is that they always evaluate to `true` or `false`. For `input<9` to be allowed as a Boolean expression, it must return a `bool`. The compiler disallows `x=42`, for example, because it assigns `x`, returning the new value, instead of checking whether `x`'s value is 42.

Language Contrast: C++—Mistakenly Using = in Place of ==

The significant feature of Boolean expressions in C# is the elimination of a common coding error that historically appeared in C/C++. In C++, Listing 3.29 is allowed.

Listing 3.29: C++, But Not C#, Allows Assignment as a Boolean Expression

```
if(input=9)    // COMPILE ERROR:  Allowed in C++, not in C#.
  System.Console.WriteLine(
      "Correct, tic-tac-toe has a maximum of 9 turns.");
```

Although this appears to check whether `input` equals `9`, Chapter 1 showed that = represents the assignment operator, not a check for equality. The return from the assignment operator is the value assigned to the variable—in this case, `9`. However, `9` is an `int`, and as such it does not qualify as a Boolean expression and is not allowed by the C# compiler.

Relational and Equality Operators

Included in the previous code examples was the use of relational operators. In those examples, relational operators were used to evaluate user input. Table 3.2 lists all the relational and equality operators.

In addition to determining whether a value is greater than or less than another value, operators are also required to determine equivalency. You test for equivalence by using equality operators. In C#, the syntax follows

Table 3.2: Relational and Equality Operators

Operator	Description	Example
<	Less than	`input<9;`
>	Greater than	`input>9;`
<=	Less than or equal to	`input<=9;`
>=	Greater than or equal to	`input>=9;`
==	Equality operator	`input==9;`
!=	Inequality operator	`input!=9;`

the C/C++/Java pattern with ==. For example, to determine whether `input` equals `9` you use `input==9`. The equality operator uses two equals signs to distinguish it from the assignment operator, `=`.

The exclamation point signifies NOT in C#, so to test for inequality you use the inequality operator, `!=`.

The relational and equality operators are binary operators, meaning they compare two operands. More significantly, they always return a Boolean data type. Therefore, you can assign the result of a relational operator to a `bool` variable, as shown in Listing 3.30.

Listing 3.30: Assigning the Result of a Relational Operator to a `bool`

```
bool result = 70 > 7;
```

In the tic-tac-toe program (see Appendix B), you use the equality operator to determine whether a user has quit. The Boolean expression of Listing 3.31 includes an OR (||) logical operator, which the next section discusses in detail.

Listing 3.31: Using the Equality Operator in a Boolean Expression

```
if (input == "" || input == "quit")
{
  System.Console.WriteLine("Player {0} quit!!", currentPlayer);
  break;
}
```

Logical Boolean Operators

Logical operators have Boolean operands and return a Boolean result. Logical operators allow you to combine multiple Boolean expressions to form other Boolean expressions. The logical operators are `||`, `&&`, and `^`, corresponding to OR, AND, and exclusive OR, respectively.

OR Operator (||)

In Listing 3.31, if the user enters `quit` or presses the Enter key without typing in a value, it is assumed that she wants to exit the program. To enable two ways for the user to resign, you use the logical OR operator, `||`.

The `||` operator evaluates two Boolean expressions and returns a `true` value if *either* of them is true (see Listing 3.32).

Listing 3.32: Using the OR Operator

```
if((hourOfTheDay > 23) || (hourOfTheDay < 0))
  System.Console.WriteLine("The time you entered is invalid.");
```

Note that with the Boolean OR operator, it is not necessary to evaluate both sides of the expression. The OR operators go from left to right, so if the left portion of the expression evaluates to `true`, then the right portion is ignored. Therefore, if `hourOfTheDay` has the value `33`, `(hourOfTheDay > 23)` will return `true` and the OR operator ignores the second half of the expression. Short-circuiting an expression also occurs with the Boolean AND operator.

AND Operator (&&)

The Boolean AND operator, `&&`, evaluates to `true` only if both operands evaluate to `true`. If either operand is false, the combined expression will return `false`.

Listing 3.33 displays that it is time for work as long as the current hour is both greater than 10 and less that 24.[2] As you saw with the OR operator, the AND operator will not always evaluate the right side of the expression. If the left operand returns `false`, then the overall result will be false regardless of the right operand, so the runtime ignores the right operand.

2. The typical hours programmers work.

Listing 3.33: Using the AND Operator

```
if ((hourOfTheDay > 10) && (hourOfTheDay < 24))
  System.Console.WriteLine(
  "Hi-Ho, Hi-Ho, it's off to work we go.");
```

Exclusive OR Operator (^)

The caret symbol, ^, is the "exclusive OR" (XOR) operator. When applied to two Boolean operands, the XOR operator returns `true` only if exactly one of the operands is true, as shown in Table 3.3.

Table 3.3: Conditional Values for the XOR Operator

Left Operand	Right Operand	Result
True	True	False
True	False	True
False	True	True
False	False	False

Unlike the Boolean AND and Boolean OR operators, the Boolean XOR operator does not short-circuit: It always checks both operands, because the result cannot be determined unless the values of both operands are known.

Logical Negation Operator (!)

Sometimes called the NOT operator, the **logical negation operator,** !, inverts a `bool` data type to its opposite. This operator is a unary operator, meaning it requires only one operand. Listing 3.34 demonstrates how it works, and Output 3.16 shows the results.

Listing 3.34: Using the Logical Negation Operator

```
bool result;
bool valid = false;
result = !valid;
// Displays "result = True".
System.Console.WriteLine("result = {0}", result);
```

OUTPUT 3.16:

```
result = True
```

To begin, `valid` is set to `false`. You then use the negation operator on `valid` and assign a new value to `result`.

Conditional Operator (?)

In place of an `if` statement that functionally returns a value, you can use the conditional operator instead. The conditional operator is a question mark (`?`), and the general format is as follows:

```
conditional? expression1: expression2;
```

The conditional operator is a ternary operator, because it has three operands: `conditional`, `expression1`, and `expression2`. If the conditional evaluates to `true`, then the conditional operator returns `expression1`. Alternatively, if the conditional evaluates to `false`, then it returns `expression2`.

Listing 3.35 is an example of how to use the conditional operator. The full listing of this program appears in Appendix B.

LISTING 3.35: CONDITIONAL OPERATOR

```
public class TicTacToe
{
  public static string Main()
  {
      // Initially set the currentPlayer to Player 1;
      int currentPlayer = 1;

      // ...

      for (int turn = 1; turn <= 10; turn++)
      {
          // ...

          // Switch players
          currentPlayer = (currentPlayer == 2) ? 1 : 2;
      }
  }
}
```

The program swaps the current player. To do this, it checks whether the current value is 2. This is the conditional portion of the conditional statement. If the result is `true`, the conditional operator returns the value 1. Otherwise, it returns 2. Unlike an `if` statement, the result of the conditional operator must be assigned (or passed as a parameter). It cannot appear as an entire statement on its own.

Use the conditional operator sparingly, because readability is often sacrificed and a simple `if/else` statement may be more appropriate.

Bitwise Operators (<<, >>, |, &, ^, ~)

An additional set of operators common to virtually all programming languages is the set of operators for manipulating values in their binary formats: the bit operators.

BEGINNER TOPIC

Bits and Bytes

All values within a computer are represented in a binary format of 1s and 0s, called **bits.** Bits are grouped together in sets of eight, called **bytes.** In a byte, each successive bit corresponds to a value of 2 raised to a power, starting from 2^0 on the right, to 2^7 on the left, as shown in Figure 3.1.

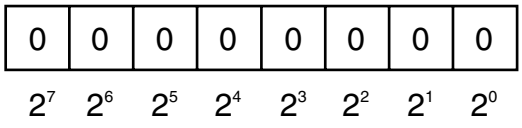


FIGURE 3.1: Corresponding Placeholder Values

In many instances, particularly when dealing with low-level or system services, information is retrieved as binary data. In order to manipulate these devices and services, you need to perform manipulations of binary data.

As shown in Figure 3.2, each box corresponds to a value of 2 raised to the power shown. The value of the byte (8-bit number) is the sum of the powers of 2 of all of the eight bits that are set to 1.

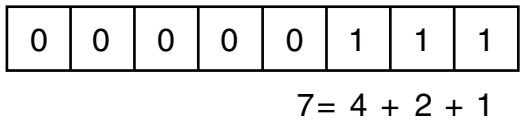


FIGURE 3.2: Calculating the Value of an Unsigned Byte

The binary translation just described is significantly different for signed numbers. Signed numbers (`long`, `short`, `int`) are represented using a 2s complement notation. With this notation, negative numbers behave differently than positive numbers. Negative numbers are identified by a 1 in the leftmost location. If the leftmost location contains a 1, you add the locations with 0s rather than the locations with 1s. Each location corresponds to the negative power of 2 value. Furthermore, from the result, it is also necessary to subtract 1. This is demonstrated in Figure 3.3.

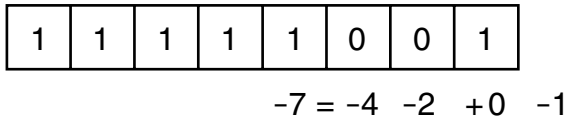


FIGURE 3.3: Calculating the Value of a Signed Byte

Therefore, `1111 1111 1111 1111` corresponds to a –1, and `1111 1111 1111 1001` holds the value –7. `1000 0000 0000 0000` corresponds to the lowest negative value a 16-bit integer can hold.

Shift Operators (<<, >>, <<=, >>=)

Sometimes you want to shift the binary value of a number to the right or left. In executing a left shift, all bits in a number's binary representation are shifted to the left by the number of locations specified by the operand on the right of the shift operator. Zeroes are then used to backfill the locations on the right side of the binary number. A right-shift operator does almost the same thing in the opposite direction. However, if the number is negative, the values used to backfill the left side of the binary number are ones and not zeroes. The shift operators are `>>` and `<<`, the right-shift and left-shift operators, respectively. In addition, there are combined shift and assignment operators, `<<=` and `>>=`.

Consider the following example. Suppose you had the `int` value `-7`, which would have a binary representation of `1111 1111 1111 1111 1111 1111 1111 1001`. In Listing 3.36 you right-shift the binary representation of the number –7 by two locations.

Listing 3.36: Using the Right-Shift Operator

```
int x;
x = (-7 >> 2); // 11111111111111111111111111111001 becomes
               // 11111111111111111111111111111110
// Write out "x is -2."
System.Console.WriteLine("x = {0}.", x);
```

Output 3.17 shows the results of Listing 3.36.

Output 3.17:

```
x = -2.
```

Because of the right shift, the value of the bit in the rightmost location has "dropped off" the edge and the negative bit indicator on the left shifts by two locations to be replaced with 1s. The result is `-2`.

Bitwise Operators (&, |, ^)

In some instances, you might need to perform logical operations, such as AND, OR, and XOR, on a bit-by-bit basis for two operands. You do this via the `&`, `|`, and `^` operators, respectively.

BEGINNER TOPIC

Logical Operators Explained

If you have two numbers, as shown in Figure 3.4, the bitwise operations will compare the values of the locations beginning at the leftmost significant value and continuing right until the end. The value of "1" in a location is treated as "true," and the value of "0" in a location is treated as "false."

Therefore, the bitwise AND of the two values in Figure 3.4 would be the bit-by-bit comparison of bits in the first operand (12) with the bits in the second operand (7), resulting in the binary value `000000100`, which is 4.

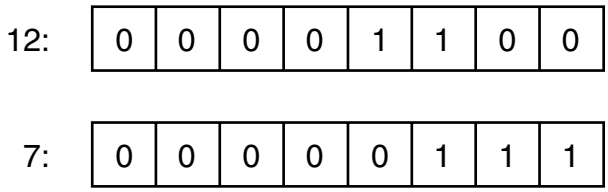


FIGURE 3.4: The Numbers 12 and 7 Represented in Binary

Alternatively, a bitwise OR of the two values would produce `00001111`, the binary equivalent of 15. The XOR result would be `00001011`, or decimal 11.

Listing 3.37 demonstrates how to use these bitwise operators. The results of Listing 3.37 appear in Output 3.18.

LISTING 3.37: USING BITWISE OPERATORS

```
byte and, or, xor;
and = 12 & 7;   // and = 4
or = 12 | 7;    // or = 15
xor = 12 ^ 7;   // xor = 11
System.Console.WriteLine(
    "and = {0} \nor = {1}\nxor = {2}",
    and, or, xor);
```

OUTPUT 3.18:

```
and = 4
or = 15
xor = 11
```

In Listing 3.37, the value 7 is the **mask;** it is used to expose or eliminate specific bits within the first operand using the particular operator expression.

In order to convert a number to its binary representation, you need to iterate across each bit in a number. Listing 3.38 is an example of a program that converts an integer to a string of its binary representation. The results of Listing 3.38 appear in Output 3.19.

LISTING 3.38: GETTING A STRING REPRESENTATION OF A BINARY DISPLAY

```
public class BinaryConverter
{
  public static void Main()
```

```
    {
        const int size = 64;
        ulong value;
        char bit;

        System.Console.Write ("Enter an integer: ");
        // Use long.Parse() so as to support negative numbers
        // Assumes unchecked assignment to ulong.
        value = (ulong)long.Parse(System.Console.ReadLine());

        // Set initial mask to 100....
        ulong mask = 1ul << size - 1;
        for (int count = 0; count < size; count++)
        {
            bit = ((mask & value) > 0) ? '1': '0';
            System.Console.Write(bit);
            // Shift mask one location over to the right
            mask >>= 1;
        }
        System.Console.WriteLine();
    }
}
```

OUTPUT 3.19:

```
Enter an integer: 42
0000000000000000000000000000000000000000000000000000000000101010
```

Notice that within each iteration of the `for` loop (discussed shortly), you use the right-shift assignment operator to create a mask corresponding to each bit in `value`. By using the `&` bit operator to mask a particular bit, you can determine whether the bit is set. If the mask returns a positive result, you set the corresponding bit to `1`; otherwise, it is set to `0`. In this way, you create a string representing the binary value of an unsigned long.

Bitwise Assignment Operators (&=, |=, ^=)

Not surprisingly, you can combine these bitwise operators with assignment operators as follows: `&=`, `|=`, and `^=`. As a result, you could take a variable, OR it with a number, and assign the result back to the original variable, which Listing 3.39 demonstrates.

Listing 3.39: Using Logical Assignment Operators

```
byte and, or, xor;
and = 12;
and &= 7;   // and = 4
or = 12;
or |= 7;   // or = 15
xor = 12;
xor ^= 7;   // xor = 11
System.Console.WriteLine(
    "and = {0} \nor = {1}\nxor = {2}",
    and, or, xor);
```

The results of Listing 3.39 appear in Output 3.20.

Output 3.20:

```
and = 4
or = 15
xor = 11
```

Bitwise Complement Operator (~)

The **bitwise complement operator** takes the complement of each bit in the operand, where the operand can be an `int`, `uint`, `long`, or `ulong`. `~1`, therefore, returns `1111 1111 1111 1111 1111 1111 1111 1110` and `~(1<<31)` returns `0111 1111 1111 1111 1111 1111 1111 1111`.

Control Flow Statements, Continued

With the additional coverage of Boolean expressions, it's time to consider more of the control flow statements supported by C#. As indicated in the introduction to this chapter, many of these statements will be familiar to experienced programmers, so you can skim this section for information specific to C#. Note in particular the `foreach` loop, as this may be new to many programmers.

The `while` and `do/while` Loops

Until now, you have learned how to write programs that do something only once. However, one of the important capabilities of the computer is that it can perform the same operation multiple times. In order to do this, you need

to create an instruction loop. The first instruction loop I will discuss is the `while` loop. The general form of the `while` statement is as follows:

```
while(boolean-expression )
  statement
```

The computer will repeatedly execute `statement` as long as `Boolean-expression` evaluates to `true`. If the statement evaluates to `false`, then code execution continues at the line following `statement`. The Fibonacci calculator shown in Listing 3.40 demonstrates this.

Listing 3.40: `while` Loop Example

```
class FibonacciCalculator
{
  static void Main()
  {
      decimal current;
      decimal previous;
      decimal temp;
      decimal input;

      System.Console.Write("Enter a positive integer:");

      // decimal.Parse convert the ReadLine to a decimal.
      input = decimal.Parse(System.Console.ReadLine());

      // Initialize current and previous to 1, the first
      // two numbers in the Fibonacci series.
      current = previous = 1;

      // While the current Fibonacci number in the series is
      // less than the value input by the user.
      while(current <= input)
      {
          temp = current;
          current = previous + current;
          previous = temp;
      }

      System.Console.WriteLine(
          "The Fibonacci number following this is {0}",
          current);
  }
}
```

A **Fibonacci number** is a member of the **Fibonacci series,** which includes all numbers that are the sum of the previous two numbers in the series,

beginning with 1 and 1. In Listing 3.40, you prompt the user for an integer. Then you use a `while` loop to find the Fibonacci number that is greater than the number the user entered.

BEGINNER TOPIC

When to Use a `while` Loop

The remainder of this chapter considers other types of statements that cause a block of code to execute repeatedly. The term *loop* refers to the block of code that is to be executed within the `while` statement, since the code is executed in a "loop" until the exit condition is achieved. It is important to understand which loop construct to select. You use a `while` construct to iterate while the condition evaluates to `true`. A `for` loop is used most appropriately whenever the number of repetitions is known, such as counting from 0 to *n*. A `do/while` is similar to a `while` loop, except that it will always loop at least once.

The `do/while` loop is very similar to the `while` loop, except that it is used when the number of repetitions is from 1 to *n* and *n* is indeterminate when iterating begins. This pattern occurs most commonly when repeatedly prompting a user for input. Listing 3.41 is taken from the tic-tac-toe program.

Listing 3.41: `do/while` Loop Example

```
// Repeatedly request player to move until he
// enter a valid position on the board.
do
{
  valid = false;

  // Request a move from the current player.
  System.Console.Write(
    "\nPlayer {0}: Enter move:", currentPlayer);
  input = System.Console.ReadLine();

  // Check the current player's input.
  // ...

} while (!valid);
```

In Listing 3.41, you always initialize `valid` to `false` at the beginning of each **iteration,** or loop repetition. Next, you prompt and retrieve the number the user input. Although not shown here, you then check whether the input was correct, and if it was, you assign `valid` equal to `true`. Since the code uses a `do/while` statement rather than a `while` statement, the user will be prompted for input at least once.

The general form of the `do/while` loop is as follows:

```
do
  statement
while(boolean-expression );
```

As with all the control flow statements, the code blocks are not part of the general form. However, a code block is generally used in place of a single statement in order to allow multiple statements.

The `for` loop

Increment and decrement operators are frequently used within a `for` loop. The `for` loop iterates a code block until a specified condition is reached in a way similar to the `while` loop. The difference is that the `for` loop has built-in syntax for initializing, incrementing, and testing the value of a counter.

Listing 3.42 shows the `for` loop used to display an integer in binary form. The results of this listing appear in Output 3.21.

Listing 3.42: Using the `for` Loop

```
public class BinaryConverter
{
  public static void Main()
  {
      const int size = 64;
      ulong value;
      char bit;

      System.Console.Write ("Enter an integer: ");
      // Use long.Parse() so as to support negative numbers
      // Assumes unchecked assignment to ulong.
      value = (ulong)long.Parse(System.Console.ReadLine());

      // Set initial mask to 100....
      ulong mask = 1ul << size - 1;
      for (int count = 0; count < size; count++)
      {
```

```
            bit = ((mask & value) > 0) ? '1': '0';
            System.Console.Write(bit);
            // Shift mask one location over to the right
            mask >>= 1;
        }
    }
}
```

OUTPUT 3.21:

```
Enter an integer: -42
11111111111111111111111111111111111111111111111111111111111010110
```

Listing 3.42 performs a bit mask 64 times, once for each bit in the number. The `for` loop declares and initializes the variable `count`, escapes once the count reaches 64, and increments the count during each iteration. Each expression within the `for` loop corresponds to a statement. (It is easy to remember that the separation character between expressions is a semicolon and not a comma, because each expression is a statement.)

You write a `for` loop generically as follows:

```
for(initial; boolean-expression; loop)
  statement
```

Here is a breakdown of the `for` loop.

- The `initial` expression performs operations that precede the first iteration. In Listing 3.42, it declares and initializes the variable `count`. The `initial` expression does not have to be a declaration of a new variable. It is possible, for example, to declare the variable beforehand and simply initialize it in the `for` loop. Variables declared here, however, are bound within the scope of the `for` statement.
- The `boolean-expression` portion of the `for` loop specifies an end condition. The loop exits when this condition is false in a manner similar to the `while` loop's termination. The `for` loop will repeat only as long as `boolean-expression` evaluates to `true`. In the preceding example, the loop exits when `count` increments to 64.
- The `loop` expression executes after each iteration. In the preceding example, `count++` executes after the right shift of the mask (`mask >>= 1`),

but before the Boolean expression is evaluated. During the sixty-fourth iteration, `count` increments to 64, causing `boolean-expression` to be `false` and, therefore, terminating the loop. Because each expression can be thought of as a separate statement, each expression in the `for` loop is separated by a semicolon.

- The `statement` portion of the `for` loop is the code that executes while the conditional expression remains true.

If you wrote out each `for` loop execution step in pseudocode without using a `for` loop expression, it would look like this:

1. Declare and initialize `count` to 0.
2. Verify that `count` is less than 64.
3. Calculate `bit` and display it.
4. Shift the mask.
5. Increment `count` by one.
6. If `count<64`, then jump back to line 3.

The `for` statement doesn't require any of the elements between parentheses. `for(;;){ ... }` is perfectly valid, assuming there is still a means to escape from the loop. Similarly, the initial and loop expressions can be a complex expression involving multiple subexpressions, as shown in Listing 3.43.

Listing 3.43: `for` Loop Using Multiple Expressions

```
for(int x=0, y=5; ((x<=5) && (y>=0)); y--, x++)
{
  System.Console.Write("{0}{1}{2}\t",
      x, (x>y? '>' : '<'), y);
}
```

The results of Listing 3.43 appear in Output 3.22.

Output 3.22:

```
0<5     1<4     2<3     3>2     4>1     5>0
```

In this case, the comma behaves exactly as it does in a declaration statement, one that declares and initializes multiple variables. However, programmers should avoid complex expressions such as this one because they are difficult to read and understand.

Generically, you can write the `for` loop as a `while` loop, as shown here:

```
initial;
while(boolean-expression)
{
  statement;
  loop;
}
```

BEGINNER TOPIC

Choosing between `for` and `while` Loops

Although you can use the two statements interchangeably, generally you would use the `for` loop whenever there is some type of counter, and the total number of iterations is known when the loop is initialized. In contrast, you would typically use the `while` loop when iterations are not based on a count or when the number of iterations is indeterminate when iterating commences.

The `foreach` Loop

The last loop statement within the C# language is `foreach`. `foreach` is designed to iterate through a collection of items, setting an identifier to represent each item in turn. During the loop, operations may be performed on the item. One feature of the `foreach` loop is that it is not possible to accidentally miscount and iterate over the end of the collection.

The general form of the `foreach` statement is as follows:

```
foreach(type identifier in collection)
  statement;
```

Here is a breakdown of the `foreach` statement.

- `type` is used to declare the data type of the identifier for each item within the collection.

- `identifier` is a read-only variable into which the `foreach` construct will automatically assign the next item within the collection. The scope of the identifier is limited to the `foreach` loop.
- `collection` is an expression, such as an array, representing multiple items.
- `statement` is the code that executes for each iteration within the `foreach` loop.

Consider the `foreach` loop in the context of the simple example shown in Listing 3.44.

Listing 3.44: Determining Remaining Moves Using the `foreach` Loop

```
class TicTacToe      // Declares the TicTacToe class.
{
  static void Main() // Declares the entry point of the program.
  {
      // Hardcode initial board as follows
      // ---+---+---
      //  1 | 2 | 3
      // ---+---+---
      //  4 | 5 | 6
      // ---+---+---
      //  7 | 8 | 9
      // ---+---+---
      char[] cells = {
        '1', '2', '3', '4', '5', '6', '7', '8', '9'
      };

      System.Console.Write(
          "The available moves are as follows: ");

      // Write out the initial available moves
      foreach (char cell in cells)
      {
        if (cell != 'O' && cell != 'X')
        {
            System.Console.Write("{0} ", cell);
        }
      }
  }
}
```

Output 3.23 shows the results of Listing 3.44.

OUTPUT 3.23:

```
The available moves are as follows: 1 2 3 4 5 6 7 8 9
```

When the execution engine reaches the `foreach` statement, it assigns to the variable `cell` the first item in the `cells` array—in this case, the value `'1'`. It then executes the code within the `foreach` statement block. The `if` statement determines whether the value of `cell` is `'O'` or `'X'`. If it is neither, then the value of `cell` is written out to the console. The next iteration then assigns the next array value to `cell`, and so on.

It is important to note that the compiler prevents modification of the identifier variable (`cell`) during the execution of a `foreach` loop.

BEGINNER TOPIC

Where the `switch` Statement Is More Appropriate

Sometimes you might compare the same value in several continuous `if` statements, as shown with the `input` variable in Listing 3.45.

LISTING 3.45: CHECKING THE PLAYER'S INPUT WITH AN `if` STATEMENT

```
// ...

bool valid = false;

// Check the current player's input.
if( (input == "1") ||
  (input == "2") ||
  (input == "3") ||
  (input == "4") ||
  (input == "5") ||
  (input == "6") ||
  (input == "7") ||
  (input == "8") ||
  (input == "9") )
{
    // Save/move as the player directed.
    // ...

    valid = true;
}
else if( (input == "") || (input == "quit") )
{
```

```
        valid = true;
    }
    else
    {
        System.Console.WriteLine(
          "\nERROR:  Enter a value from 1-9. "
          + "Push ENTER to quit");
    }

    // ...
```

This code validates the text entered to ensure that it is a valid tic-tac-toe move. If the value of `input` were `9`, for example, the program would have to perform nine different evaluations. It would be preferable to jump to the correct code after only one evaluation. To enable this, you use a `switch` statement.

The `switch` Statement

Given a variable to compare and a list of constant values to compare against, the `switch` statement is simpler to read and code than the `if` statement. The `switch` statement looks like this:

```
switch(governing-type-expression)
{
  [case constant:
      statement
      jump expression]
  [default:
      statement
      jump expression]
}
```

Here is a breakdown of the `switch` statement.

- `governing-type-expression` returns a value that is compatible with the governing types. Allowable governing data types are `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `string`, and an `enum`-type (covered in Chapter 8).
- `constant` is any constant expression compatible with the data type of the governing type.
- `statement` is one or more statements to be executed when the governing type expression equals the constant value.

- `jump expression` is a jump statement such as a `break` or `goto` statement. If the `switch` statement appears within a loop, then `continue` is also allowed.

A `switch` statement must have at least one `case` statement or a default statement. In other words, `switch(x){}` is not valid.

Listing 3.46, with a `switch` statement, is semantically equivalent to the series of `if` statements in Listing 3.45.

Listing 3.46: Replacing the `if` Statement with a `switch` Statement

```
static bool ValidateAndMove(
  int[] playerPositions, int currentPlayer, string input)
{
  bool valid = false;

  // Check the current player's input.
  switch (input)
  {
    case "1" :
    case "2" :
    case "3" :
    case "4" :
    case "5" :
    case "6" :
    case "7" :
    case "8" :
    case "9" :
      // Save/move as the player directed.
      ...
      valid = true;
      break;

    case "" :
    case "quit" :
      valid = true;
      break;
    default :
      // If none of the other case statements
      // is encountered then the text is invalid.
      System.Console.WriteLine(
        "\nERROR:  Enter a value from 1-9. "
        + "Push ENTER to quit");
      break;
  }

  return valid;
}
```

In Listing 3.46, `input` is the governing type expression. Since `input` is a string, all of the constants are strings. If the value of `input` is `1`, `2`, … `9`, then the move is valid and you change the appropriate cell to match that of the current user's token (X or O). Once execution encounters a `break` statement, it immediately jumps to the statement following the `switch` statement.

The next portion of the switch looks for `""` or `"quit"`, and sets `valid` to `true` if `input` equals one of these values. Ultimately, the `default` label is executed if no prior `case` constant was equivalent to the governing type.

There are several things to note about the `switch` statement.

- Placing nothing within the `switch` block will generate a compiler warning, but the statement will still compile.
- `default` does not have to appear last within the `switch` statement. `case` statements appearing after `default` are evaluated.
- When you use multiple constants for one `case` statement, they should appear consecutively, as shown in Listing 3.46.
- The compiler requires a jump statement (usually a `break`).

Language Contrast: C++—`switch` Statement Fall-Through

Unlike C++, C# does not allow a `switch` statement to fall through from one case block to the next if the `case` includes a statement. A jump statement is always required following the statement within a `case`. The C# founders believed it was better to be explicit and require the jump expression in favor of code readability. If programmers want to use a fall-through semantic, they may do so explicitly with a `goto` statement, as demonstrated in the section The `goto` Statement, later in this chapter.

Jump Statements

It is possible to alter the execution path of a loop. In fact, with jump statements, it is possible to escape out of the loop or to skip the remaining portion of an iteration and begin with the next iteration, even when the conditional expression remains true. This section considers some of the ways to jump the execution path from one location to another.

The `break` Statement

To escape out of a loop or a `switch` statement, C# uses a `break` statement. Whenever the `break` statement is encountered, the execution path immediately jumps to the first statement following the loop. Listing 3.47 examines the `foreach` loop from the tic-tac-toe program.

Listing 3.47: Using `break` to Escape Once a Winner Is Found

```
class TicTacToe      // Declares the TicTacToe class.
{
  static void Main() // Declares the entry point of the program.
  {
      int winner=0;
      // Stores locations each player has moved.
      int[] playerPositions = {0,0};

      // Hardcoded board position
      //  X | 2 | O
      // ---+---+---
      //  O | O | 6
      // ---+---+---
      //  X | X | X
      playerPositions[0] = 449;
      playerPositions[1] = 28;

      // Determine if there is a winner
      int[] winningMasks = {
            7, 56, 448, 73, 146, 292, 84, 273 };

      // Iterate through each winning mask to determine
      // if there is a winner.
      foreach (int mask in winningMasks)
      {
          if ((mask & playerPositions[0]) == mask)
          {
              winner = 1;
              break;
          }
          else if ((mask & playerPositions[1]) == mask)
          {
              winner = 2;
              break;
          }
      }

      System.Console.WriteLine(
          "Player {0} was the winner", winner);
  }
}
```

Output 3.24 shows the results of Listing 3.47.

Output 3.24:

```
Player 1 was the winner
```

Listing 3.47 uses a `break` statement when a player holds a winning position. The `break` statement forces its enclosing loop (or a `switch` statement) to cease execution, and the program moves to the next line outside the loop. For this listing, if the bit comparison returns `true` (if the board holds a winning position), the `break` statement causes execution to jump and display the winner.

BEGINNER TOPIC

Bitwise Operators for Positions

The tic-tac-toe example uses the bitwise operators (Appendix B) to determine which player wins the game. First, the code saves the positions of each player into a bitmap called `playerPositions`. (It uses an array so that the positions for both players can be saved.)

To begin, both `playerPositions` are `0`. As each player moves, the bit corresponding to the move is set. If, for example, the player selects cell 3, `shifter` is set to `3 - 1`. The code subtracts 1 because C# is zero-based and you need to adjust for 0 as the first position instead of 1. Next, the code sets `position`, the bit corresponding to cell 3, using the shift operator `0000000000000001 << shifter`, where `shifter` now has a value of `2`. Lastly, it sets `playerPositions` for the current player (subtracting 1 again to shift to zero-based) to `0000000000000100`. Listing 3.48 uses `|=` so that previous moves are combined with the current move.

Listing 3.48: Setting the Bit That Corresponds to Each Player's Move

```
int shifter;  // The number of places to shift
              // over in order to set a bit.
int position;  // The bit which is to be set.

// int.Parse() converts "input" to an integer.
// "int.Parse(input) - 1" because arrays
```

```
// are zero-based.
shifter = int.Parse(input) - 1;

// Shift mask of 00000000000000000000000000000001
// over by cellLocations.
position = 1 << shifter;

// Take the current player cells and OR them to set the
// new position as well.
// Since currentPlayer is either 1 or 2,
// subtract one to use currentPlayer as an
// index in a 0-based array.
playerPositions[currentPlayer-1] |= position;
```

Later in the program, you can iterate over each mask corresponding to winning positions on the board to determine whether the current player has a winning position, as shown in Listing 3.47.

The `continue` Statement

In some instances, you may have a series of statements within a loop. If you determine that some conditions warrant executing only a portion of these statements for some iterations, you use the `continue` statement to jump to the end of the current iteration and begin the next iteration. The C# `continue` statement allows you to exit the current iteration (regardless of which additional statements remain) and jump to the loop conditional. At that point, if the loop conditional remains `true`, the loop will continue execution.

Listing 3.49 uses the `continue` statement so that only the letters of the domain portion of an email are displayed. Output 3.25 shows the results of Listing 3.49.

Listing 3.49: Determining the Domain of an Email Address

```
class EmailDomain
{
  static void Main()
  {
      string email;
      bool insideDomain = false;
      System.Console.WriteLine("Enter an email address: ");

      email = System.Console.ReadLine();
```

```
        System.Console.Write("The email domain is: ");

        // Iterate through each letter in the email address.
        foreach (char letter in email)
        {
            if (!insideDomain)
            {
                if (letter == '@')
                {
                    insideDomain = true;
                }
                continue;
            }

            System.Console.Write(letter);
        }
    }
}
```

Output 3.25:

```
Enter an email address:
mark@dotnetprogramming.com
The email domain is: dotnetprogramming.com
```

In Listing 3.49, if you are not yet inside the domain portion of the email address, you need to use a `continue` statement to jump to the next character in the email address.

In general, you can use an `if` statement in place of a `continue` statement, and this is usually more readable. The problem with the `continue` statement is that it provides multiple exit points within the iteration, and this compromises readability. In Listing 3.50, the sample has been rewritten, replacing the `continue` statement with the `if/else` construct to demonstrate a more readable version that does not use the `continue` statement.

Listing 3.50: Replacing a `continue` with an `if` Statement

```
foreach (char letter in email)
{
  if (insideDomain)
  {
      System.Console.Write(letter);
  }
  else
  {
```

```
        if (letter == '@')
        {
            insideDomain = true;
        }
    }
}
```

The goto Statement

With the advent of object-oriented programming and the prevalence of well-structured code, the existence of a `goto` statement within C# seems like an aberration to many experienced programmers. However, C# supports `goto`, and it is the only method for supporting fall-through within a `switch` statement. In Listing 3.51, if the `/out` option is set, code execution jumps to the `default` case using the `goto` statement; similarly for `/f`.

Listing 3.51: Demonstrating a `switch` with `goto` Statements

```
// ...
static void Main(string[] args)
{
  bool isOutputSet = false;
  bool isFiltered = false;

  foreach (string option in args)
  {
      switch (option)
      {
          case "/out":
              isOutputSet = true;
              isFiltered = false;
              goto default;
          case "/f":
              isFiltered = true;
              isRecursive = false;
              goto default;
          default:
              if (isRecursive)
              {
                  // Recurse down the hierarchy
                  // ...

              }
              else if (isFiltered)
              {
                  // Add option to list of filters.
                  // ...
              }
```

```
                break;
            }

        }

        // ...

    }
```

Output 3.26 shows the results of Listing 3.51.

Output 3.26:

```
C:\SAMPLES>Generate /out fizbottle.bin /f "*.xml" "*.wsdl"
```

As demonstrated in Listing 3.51, `goto` statements are ugly. In this particular example, this is the only way to get the desired behavior of a `switch` statement. Although you can use `goto` statements outside `switch` statements, they generally cause poor program structure and you should deprecate them in favor of a more readable construct. Note also that you cannot use a `goto` statement to jump from outside a `switch` statement into a label within a `switch` statement.

C# Preprocessor Directives

Control flow statements evaluate conditional expressions at runtime. In contrast, the C# preprocessor is invoked during compilation. The preprocessor commands are directives to the C# compiler, specifying the sections of code to compile or identifying how to handle specific errors and warnings within the code. C# preprocessor commands can also provide directives to C# editors regarding the organization of code.

Each preprocessor directive begins with a hash symbol (#), and all preprocessor directives must appear on one line. A newline rather than a semicolon indicates the end of the directive.

A list of each preprocessor directive appears in Table 3.4.

Language Contrast: C++—Preprocessing

Languages such as C and C++ contain a preprocessor, a separate utility from the compiler that sweeps over code, performing actions based on special tokens. Preprocessor directives generally tell the compiler how to compile the code in a file and do not participate in the compilation process itself. In contrast, the C# compiler handles preprocessor directives as part of the regular lexical analysis of the source code. As a result, C# does not support preprocessor macros beyond defining a constant. In fact, the term *precompiler* is generally a misnomer for C#.

TABLE 3.4: PREPROCESSOR DIRECTIVES

STATEMENT OR EXPRESSION	GENERAL SYNTAX STRUCTURE	EXAMPLE

`#if` command	**`#if`** preprocessor-expression
	  code
	`#endif`
		`#if` `CSHARP2`
		  `Console.Clear();`
		`#endif`

`#define` command	**`#define`** conditional-symbol	**`#define`** `CSHARP2`
`#undef` command	**`#undef`** conditional-symbol	**`#undef`** `CSHARP2`
`#error` command	**`#error`** preproc-message	**`#error`** `Buggy implementation`
`#warning` command	**`#warning`** preproc-message	**`#warning`** `Needs code review`
`#pragma` command	**`#pragma`** warning	**`#pragma`** `warning disable 1030`
`#line` command	**`#line`** org-line new-line **`#line`** default	**`#line`** `467 "TicTacToe.cs"` `...` **`#line default`**

`#region` command	**`#region`** pre-proc-message
	  code
	`#endregion`
		`#region` Methods
		  ...
		`#endregion`

Excluding and Including Code (#if, #elif, #else, #endif)

Perhaps the most common use of preprocessor directives is in controlling when and how code is included. For example, to write code that could be compiled by both C# 2.0 and later compilers and the prior version 1.2 compilers, you use a preprocessor directive to exclude C# 2.0-specific code when compiling with a 1.2 compiler. You can see this in the tic-tac-toe example and in Listing 3.52.

Listing 3.52: Excluding C# 2.0 Code from a C# 1.x Compiler

```
#if CSHARP2
System.Console.Clear();
#endif
```

In this case, you call the `System.Console.Clear()` method, which is available only in the 2.0 CLI version and later. Using the `#if` and `#endif` preprocessor directives, this line of code will be compiled only if the preprocessor symbol `CSHARP2` is defined.

Another use of the preprocessor directive would be to handle differences among platforms, such as surrounding Windows- and Linux-specific APIs with `WINDOWS` and `LINUX` `#if` directives. Developers often use these directives in place of multiline comments (`/*...*/`) because they are easier to remove by defining the appropriate symbol or via a search and replace. A final common use of the directives is for debugging. If you surround code with a `#if DEBUG`, you will remove the code from a release build on most IDEs. The IDEs define the `DEBUG` symbol by default in a debug compile and `RELEASE` by default for release builds.

To handle an else-if condition, you can use the `#elif` directive within the `#if` directive, instead of creating two entirely separate `#if` blocks, as shown in Listing 3.53.

Listing 3.53: Using #if, #elif, and #endif Directives

```
#if LINUX
...
#elif WINDOWS
...
#endif
```

Defining Preprocessor Symbols (#define, #undef)

You can define a preprocessor symbol in two ways. The first is with the #define directive, as shown in Listing 3.54.

Listing 3.54: A #define Example

```
#define CSHARP2
```

The second method uses the define option when compiling for .NET, as shown in Output 3.27.

Output 3.27:

```
>csc.exe /define:CSHARP2 TicTacToe.cs
```

Output 3.28 shows the same functionality using the Mono compiler.

Output 3.28:

```
>mcs.exe -define:CSHARP2 TicTacToe.cs
```

To add multiple definitions, separate them with a semicolon. The advantage of the define complier option is that no source code changes are required, so you may use the same source files to produce two different binaries.

To undefine a symbol you use the #undef directive in the same way you use #define.

Emitting Errors and Warnings (#error, #warning)

Sometimes you may want to flag a potential problem with your code. You do this by inserting #error and #warning directives to emit an error or warning, respectively. Listing 3.55 uses the tic-tac-toe sample to warn that the code does not yet prevent players from entering the same move multiple times. The results of Listing 3.55 appear in Output 3.29.

Listing 3.55: Defining a Warning with #warning

```
#warning    "Same move allowed multiple times."
```

Output 3.29:

```
Performing main compilation...
...\tictactoe.cs(471,16): warning CS1030: #warning: '"Same move allowed
multiple times."'

Build complete -- 0 errors, 1 warnings
```

By including the #warning directive, you ensure that the compiler will report a warning, as shown in Output 3.29. This particular warning is a way of flagging the fact that there is a potential enhancement or bug within the code. It could be a simple way of reminding the developer of a pending task.

Turning Off Warning Messages (#pragma)

Warnings are helpful because they point to code that could potentially be troublesome. However, sometimes it is preferred to turn off particular warnings explicitly because they can be ignored legitimately. C# 2.0 and later compilers provide the preprocessor #pragma directive for just this purpose (see Listing 3.56).

Listing 3.56: Using the Preprocessor #pragma Directive to Disable the #warning Directive

```
#pragma warning disable  1030
```

Note that warning numbers are prefixed with the letters CS in the compiler output. However, this prefix is not used in the #pragma warning directive. The number corresponds to the warning error number emitted by the compiler when there is no preprocessor command.

To reenable the warning, #pragma supports the restore option following the warning, as shown in Listing 3.57.

Listing 3.57: Using the Preprocessor #pragma Directive to Restore a Warning

```
#pragma warning restore  1030
```

In combination, these two directives can surround a particular block of code where the warning is explicitly determined to be irrelevant.

Perhaps one of the most common warnings to disable is CS1591, as this appears when you elect to generate XML documentation using the `/doc` compiler option, but you neglect to document all of the public items within your program.

`nowarn:<warn list>` Option

In addition to the `#pragma` directive, C# compilers generally support the `nowarn:<warn list>` option. This achieves the same result as `#pragma`, except that instead of adding it to the source code, you can simply insert the command as a compiler option. In addition, the `nowarn` option affects the entire compilation, and the `#pragma` option affects only the file in which it appears. Turning off the CS1591 warning, for example, would appear on the command line as shown in Output 3.30.

Output 3.30:

```
> csc /doc:generate.xml /nowarn:1591 /out:generate.exe Program.cs
```

Specifying Line Numbers (`#line`)

The `#line` directive controls on which line number the C# compiler reports an error or warning. It is used predominantly by utilities and designers that emit C# code. In Listing 3.58, the actual line numbers within the file appear on the left.

Listing 3.58: The `#line` Preprocessor Directive

```
124         #line 113 "TicTacToe.cs"
125         #warning "Same move allowed multiple times."
126         #line default
```

Including the `#line` directive causes the compiler to report the warning found on line 125 as though it was on line 113, as shown in the compiler error message shown in Output 3.31.

Following the `#line` directive with `default` reverses the effect of all prior `#line` directives and instructs the compiler to report true line numbers rather than the ones designated by previous uses of the `#line` directive.

OUTPUT 3.31:

```
Performing main compilation...
...\tictactoe.cs(113,18): warning CS1030: #warning: '"Same move allowed
multiple times."'

Build complete -- 0 errors, 1 warnings
```

Hints for Visual Editors (`#region`, `#endregion`)

C# contains two preprocessor directives, `#region` and `#endregion`, that are useful only within the context of visual code editors. Code editors, such as the one in the Microsoft Visual Studio .NET IDE, can search through source code and find these directives to provide editor features when writing code. C# allows you to declare a region of code using the `#region` directive. You must pair the `#region` directive with a matching `#endregion` directive, both of which may optionally include a descriptive string following the directive. In addition, you may nest regions within one another.

Again, Listing 3.59 shows the tic-tac-toe program as an example.

LISTING 3.59: A `#region` AND `#endregion` PREPROCESSOR DIRECTIVE

```
...
#region Display Tic-tac-toe Board

#if CSHARP2
  System.Console.Clear();
#endif

// Display the current board;
border = 0;   //  set the first border (border[0] = "|")

// Display the top line of dashes.
// ("\n---+---+---\n")
System.Console.Write(borders[2]);
foreach (char cell in cells)
{
  // Write out a cell value and the border that comes after it.
  System.Console.Write(" {0} {1}", cell, borders[border]);

  // Increment to the next border;
  border++;

  // Reset border to 0 if it is 3.
  if (border == 3)
  {
      border = 0;
```

```
    }
}
#endregion Display Tic-tac-toe Board
...
```

One example of how these preprocessor directives are used is with Microsoft Visual Studio .NET. Visual Studio .NET examines the code and provides a tree control to open and collapse the code (on the left-hand side of the code editor window) that matches the region demarcated by the `#region` directives (see Figure 3.5).

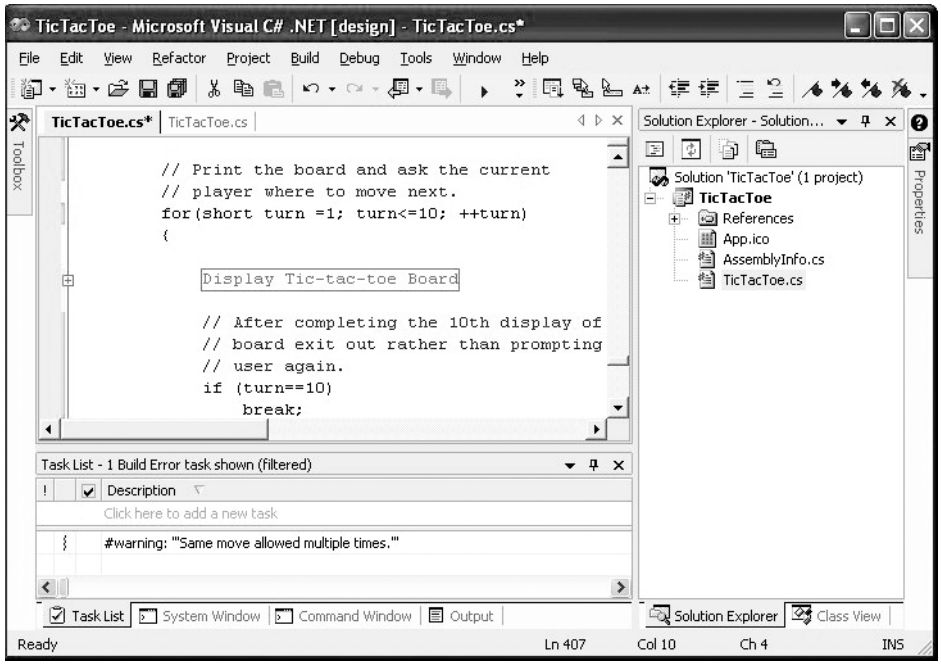


FIGURE 3.5: Collapsed Region in Microsoft Visual Studio .NET

SUMMARY

This chapter began with an introduction to the C# operators related to assignment and arithmetic. Next, you used the operators along with the `const` keyword to declare constant expressions. Coverage of all of the C# operators was not sequential, however. Before discussing the relational and logical comparison operators, I introduced the `if` statement and the

important concepts of code blocks and scope. To close out the coverage of operators I discussed the bitwise operators, especially regarding masks.

Operator precedence was discussed earlier in the chapter, but Table 3.5 summarizes the order of precedence across all operators, including several that are not yet covered.

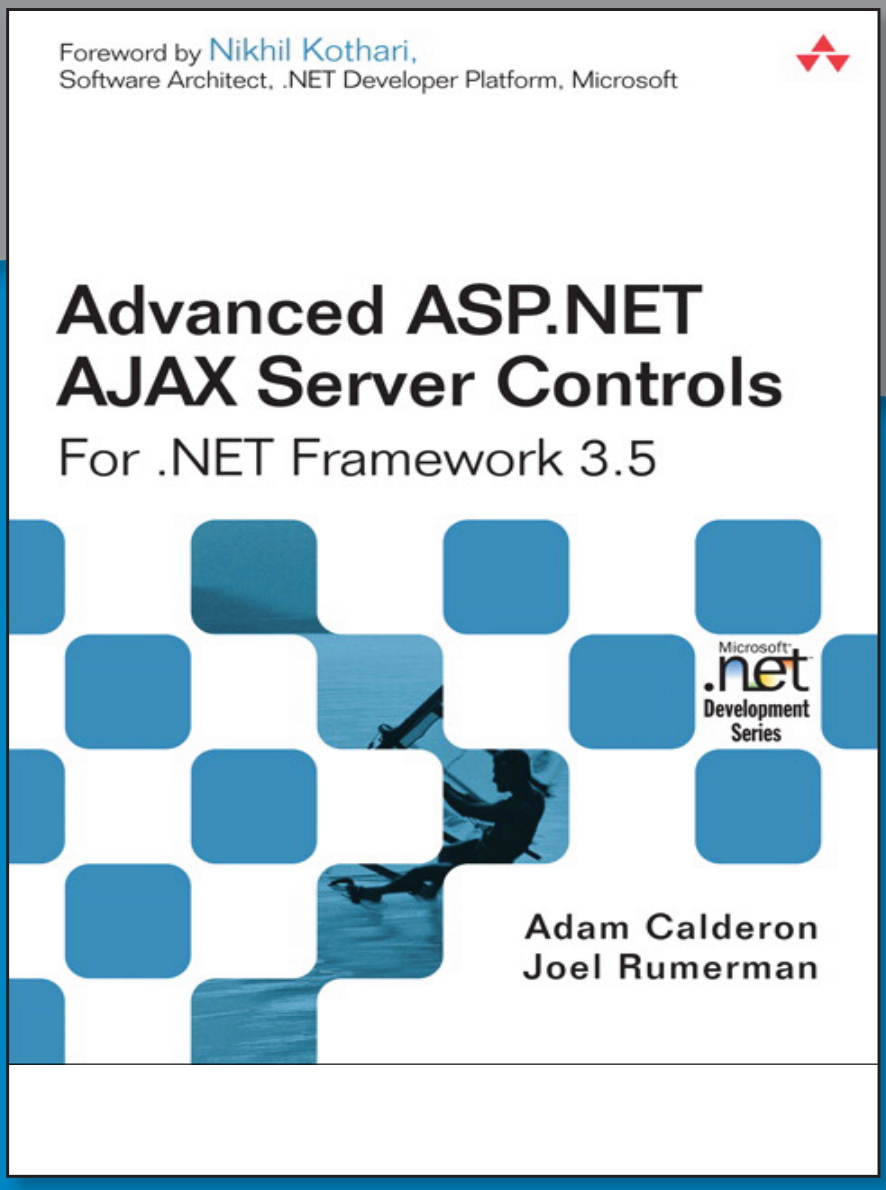
TABLE 3.5: OPERATOR ORDER OF PRECEDENCE[a]

CATEGORY	OPERATORS
Primary	`x.y  f(x)  a[x]  x++  x--  new typeof(T)  checked(x)  unchecked(x) default(T) delegate{}  ()`
Unary	`+  -  !  ~  ++x  --x  (T)x`
Multiplicative	`*  /  %`
Additive	`+  -`
Shift	`<<  >>`
Relational and type testing	`<  >  <=  >=  is  as`
Equality	`==  !=`
Logical AND	`&`
Logical XOR	`^`
Logical OR	`|`
Conditional AND	`&&`
Conditional OR	`||`
Null coalescing	`??`
Conditional	`?:`
Assignment	`= => *=  /=  %=  +=  -=  <<=  >>=  &=  ^=  |=`

a. Rows appear in order of precedence from highest to lowest.

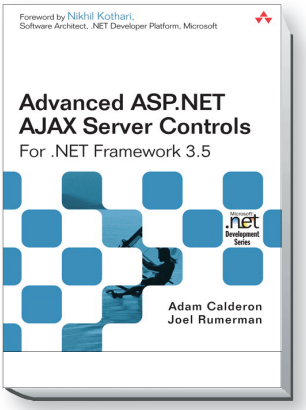
Given coverage of most of the operators, the next topic was control flow statements. The last sections of the chapter detailed the preprocessor directives and the bit operators, which included code blocks, scope, Boolean expressions, and bitwise operators.

Perhaps one of the best ways to review all of the content covered in Chapters 1–3 is to look at the tic-tac-toe program found in Appendix B. By reviewing the program, you can see one way in which you can combine all you have learned into a complete program.



Adam Calderon
Joel Rumerman

BUY ME



Advanced ASP.NET AJAX Server Controls

ASP.NET AJAX server controls can encapsulate even the most powerful AJAX functionality, helping you build more elegant, maintainable, and scalable applications. This is the first comprehensive, code-rich guide to custom ASP.NET AJAX server controls for experienced ASP.NET developers. Unlike other books on ASP.NET AJAX, this book focuses solely on server control development and reflects the significant improvements in ASP.NET 3.5 AJAX and the latest Visual Studio 2008 features for streamlining AJAX development

Adam Calderon and Joel Rumerman first review the core Microsoft AJAX Library and JavaScript techniques needed to support a rich client-side experience. Next, they build upon these techniques showing how to create distributable AJAX-enabled controls that include rich browser-independent JavaScript client-side functionality. The authors thoroughly explain both the JavaScript and .NET aspects of control development and how these two distinct environments come together to provide a foundation for building a rich user experience using ASP.NET AJAX.

- Create object-oriented cross-browser JavaScript that supports .NET style classes, interfaces, inheritance, and method overloading
- Work with components, behaviors, and controls, and learn how they relate to DOM elements
- Learn Sys.Application and the part it plays in object creation, initialization, and events in the Microsoft AJAX Library
- Build Extender and Script controls that provide integrated script generation for their corresponding client-side counterparts
- Localize ASP.NET AJAX controls including client script
- Discover ASP.NET AJAX client and server communication architecture and the new support for Windows Communication Foundation (WCF)
- Understand ASP.NET AJAX Application Services
- Create custom Application Services
- Design controls for a partial postback environment
- Understand the AJAX Control Toolkit architecture and the many features it provides
- Develop highly interactive controls using the AJAX Control Toolkit
- Understand AJAX Control Toolkit architecture and build controls that utilize the toolkit

AVAILABLE
- BOOK: 9780321514448
- SAFARI ONLINE Safari Books Online
- EBOOK: 0321574230
- KINDLE: 0321574214

About the Authors

Adam Calderon is a C# MVP and the Application Development Practice Lead at InterKnowlogy. He is an accomplished software developer, author, teacher, and speaker with more than 14 years of experience designing and developing solutions on the Microsoft platform. His involvement with ASP.NET AJAX began in late 2005 with his participation in the ASP.NET ATLAS First Access program and later as a member of the UI Server Frameworks Advisory Council. Adam was one of the fortunate few who were able to work on a production application that utilized ASP.NET AJAX in its alpha form and experienced firsthand the trials and tribulations of working in "beta land" on this exciting technology. Visit Adam's blog at http://blogs.interknowlogy.com/adamcalderon.

Joel Rumerman is a Senior .NET Developer at the CoStar Group, where he develops ASP.NET applications to support the company's commercial real estate information business. He is an adept software developer with more than eight years of experience developing .NET applications and is active in the San Diego .NET community as an author and speaker. Joel has been working with ASP.NET AJAX since late 2005 when he started work on a large-scale application for a worldwide independent software vendor. This initial entry into the ASP.NET AJAX world provided him invaluable experience as he worked closely with Microsoft as a member of the ATLAS First Access program and participated in a Strategic Design Review of the technology. Joel has gone on to implement many more solutions using ASP.NET AJAX, including a Virtual Earth mash-up that maps commercial real estate properties. Visit Joel's blog at http://seejoelprogram.wordpress.com.

Addison Wesley

informit.com/aw

3
Components

In Chapter 2, "Microsoft AJAX Library Programming," we began our discussion of the Microsoft AJAX Library and how it extends the built-in JavaScript types with new features, how to use the Prototype Model to extend the Library with our own custom types, and we even covered a few of the important prebuilt types.

In this chapter, we continue our discussion of the Microsoft AJAX Library by covering components and its two derived types, controls and behaviors. This chapter is the start of creating client objects that will be related to server controls.

Components Defined

A component is any object whose client type inherits from `Sys.Component`. They are extremely important because you'll use the `Sys.Component` base type to extend the framework to create new components. You'll want to create new components because `Sys.Component` contains a few distinct characteristics not found in any other Microsoft AJAX Library type.

First, components are designed to bridge the gap between client and server programming. Through server objects called `ScriptDescriptors`, we can instruct ASP.NET AJAX to automatically emit JavaScript that creates instances of our component types. Using this feature, we can attach client

capabilities to web server controls without actually writing any JavaScript in our web server control's class.

> **NOTE Creating Components through Server Code**
>
> We cover creating components through server code in detail in Chapter 5, "Adding Client Capabilities to Server Controls."

Second, `Sys.Application`, which is a global object that acts like a client runtime, is set up to manage any type that inherits from `Sys.Component`. This means that your component will go through a predefined lifecycle. You'll know when the component will be created and when it will be disposed, and you can inject your own custom code at these points as needed. This provides you with a great deal of control and safety.

> **NOTE Components and Web Server Controls**
>
> `Sys.Application` managing components is similar to a page managing web server controls. It was designed that way on purpose to give ASP.NET developers a familiar feel when programming within the Microsoft AJAX Library. We cover `Sys.Application` in detail in Chapter 4, "Sys.Application."

Finally, components have a lot of the common functionality that you'll need already built-in. They have a `Sys.EventHandlerList` instance, so you can create, maintain, and raise custom events. They implement the `Sys.INotifyPropertyChanged` interface, which provides property-changed notification methods. And they implement the `Sys.INotifyDisposing` interface so that other objects can be notified easily when they are disposed.

Components, Controls, and Behaviors

As if components weren't already great, there are two special-purpose component types contained within the Microsoft AJAX Library: behaviors, represented by the `Sys.UI.Behavior` class; and controls, represented by the `Sys.UI.Control` class. Figure 3.1 shows the hierarchy between the three types.

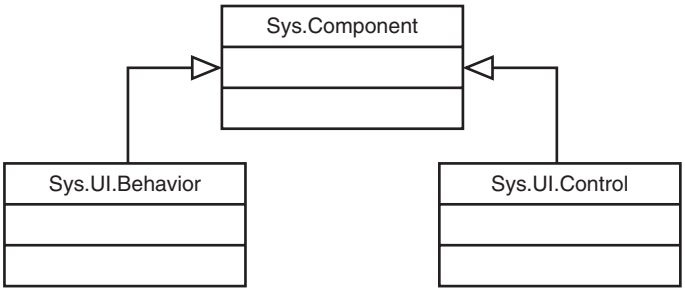


FIGURE 3.1 Class hierarchy between Sys.Component, Sys.UI.Behavior, and Sys.UI.Control

TIP Managed Components

As Figure 3.1 shows, `Sys.Component` is the base type for both `Sys.UI.Control` and `Sys.UI.Behavior`. As stated earlier, `Sys.Application` manages components. It is through inheritance that controls and behaviors are managed, too. When we talk about `Sys.Application`, we refer to it as having managed components, which could be a component, behavior, or control.

Behaviors, controls, and components are mostly the same. This is the case because when compared to the amount of functionality the base component type provides, behaviors and controls don't provide much, and the functionality they do provide doesn't take them in a radically different direction.

The one striking difference that does exist between a base component and a behavior or control is behaviors and controls have a built-in association to a DOM element because they are intended to be visual. In comparison, components do not have a built-in association to a DOM element because they are intended to be nonvisual.

Between behaviors and controls, the major difference is that a DOM element can have only one control associated to it, whereas it can have multiple associated behaviors.

Table 3.1 summarizes the differences between the three types.

Table 3.1 Differences between Components, Controls, and Behaviors

Object Type	Can Be Associated to a DOM Element	A DOM Element Can Have More Than One Associated to It	Access to Object from DOM Element
Component	Not allowed	N/A (not directly associated to a DOM element).	N/A (not directly associated to a DOM element).
Control	Must be associated to a DOM element	No, a DOM element can have only one associated control.	Yes, a control can be accessed through a `control` expando property attached to the DOM element.
Behavior	Must be associated to a DOM element	Yes, a DOM element can have one or more associated behaviors.	Yes, a behavior can be accessed through an expando property of the behavior's name from the DOM element if the behavior was named at the time it was initialized. All behaviors attached to an element can be accessed by a private `_behaviors` array attached to the DOM element.

These rules are enforced during the creation of a component, behavior, and control and dictate what base type your new type will inherit from. Figure 3.2 covers the basic decision process when determining what type of new object to create based on the feature's requirements.

Now that we covered the basics of components, controls, and behaviors, let's tackle each type individually.

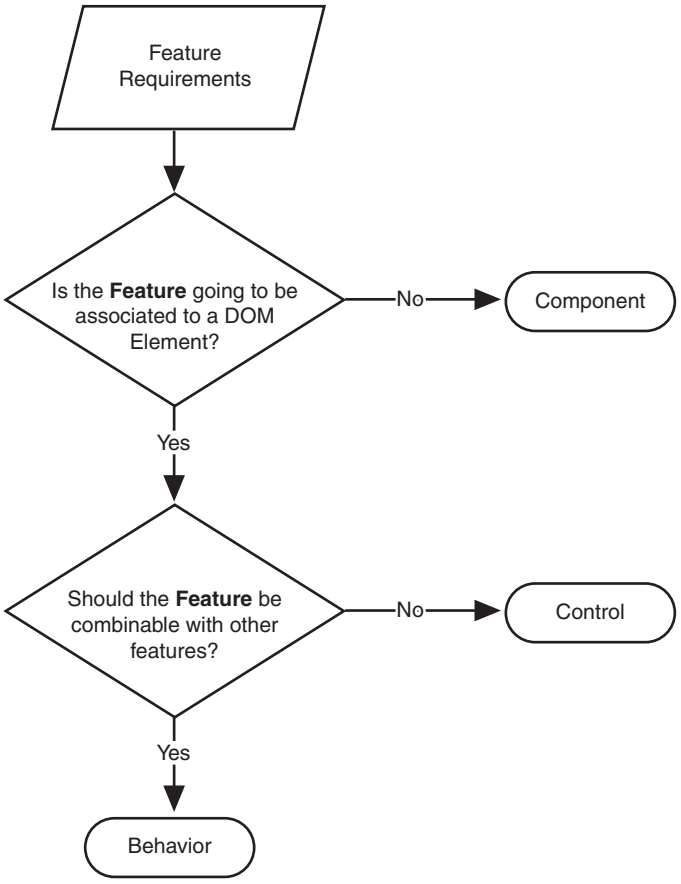


FIGURE 3.2 Decision process between component, control, and behavior

Sys.Component

`Sys.Component` is the root type of all components and provides the majority of the functionality. It does not inherit from another type, but does implement three interfaces: `Sys.IDisposable`, `Sys.INotifyPropertyChanged`, and `Sys.INotifyDisposing`. Table 3.2 details these three interfaces.

Table 3.2 Interfaces Implemented by Sys.Component

Interface	Purpose	Methods
`Sys.INotifyPropertyChanged`	Implements property-changed notification event	`add_propertyChanged` `remove_propertyChanged`
`Sys.INotifyDisposing`	Implements disposing event	`add_disposing` `remove_disposing`
`Sys.IDisposable`	Represents a disposable object.	`dispose`

`Sys.Component` also contains five internal members, as detailed in Table 3.3.

Table 3.3 Sys.Component Members

Member Name	Purpose	Type
`_id`	The unique identifier of the component. Used to find the component after it's registered with `Sys.Application`. Each component managed by `Sys.Application` must have a unique ID.	`string`
`_idSet`	Indicates whether the `_id` property has been set.	`boolean`
`_initializing`	Indicates whether the component has been through its initialization routine.	`boolean`
`_updating`	Indicates whether the component is updating.	`boolean`
`_events`	Maintains a list of events and event handlers.	`Sys.EventHandlerList`

Besides implementing the methods required by the three interfaces, `Sys.Component` exposes methods that allow interaction with its internal

members. Table 3.4 details these methods and the methods required by the three interfaces.

Table 3.4 Sys.Component Methods

Method Name	Description	Syntax
`beginUpdate`	Marks the component as updating. Called during the creation of a component.	`comp.beginUpdate();`
`endUpdate`	Marks the component as not updating. Called during the creation of a component. Executes the `initialize` method if the component is not initialized. Executes the updated method.	`comp.endUpdate();`
`updated`	Empty implementation.	`comp.updated();`
`get_isUpdating`	Getter for the updating member.	`comp.get_isUpdating();`
`initialize`	Marks the component as initialized.	`comp.initialize();`
`get_initialized`	Getter for the initialized member.	`comp.get_initialized();`
`dispose`	Executes the disposing event handlers. Removes the `_events` property from the component. Unregisters the component from `Sys.Application`.	`comp.dispose();`

TABLE 3.4 continued

Method Name	Description	Syntax
`get_events`	Getter for the events member.	`comp.get_events()`
`get_id`	Getter for the ID member.	`comp.get_id();`
`set_id`	Setter for the ID member. ID cannot be changed after it has been set (through this setter) or after the component has been registered with `Sys.Application`.	`comp.set_id(id);`
`add_disposing`	Adds an event handler to the `disposing` event.	`comp.add_disposing(`*`handler`*`);`
`remove_disposing`	Removes an event handler from the `disposing` event.	`comp.remove_disposing(`*`handler`*`);`
`add_propertyChanged`	Adds an event handler to the `propertyChanged` event.	`comp.add_propertyChanged(`*`handler`*`);`
`remove_propertyChanged`	Removes an event handler from the `propertyChanged` event.	`comp.remove_propertyChanged(`*`handler`*`)`
`raisePropertyChanged`	Executes registered `propertyChanged` event handlers passing in the name of the property that changed in the event arguments.	`comp.raisePropertyChanged (`*`propertyName`*`);`

> **NOTE beginUpdate, endUpdate, and initialize**
>
> `beginUpdate`, `endUpdate`, and `initialize` are automatically executed during the component creation process. They are normally not executed by user-defined code, but can be overridden to provide custom functionality.

Defining New Components

`Sys.Component` is extremely useful, but directly creating instances of it is not its purpose. Instead, it is intended to be used as a base class for user-defined components.

We can define a new component type using the Prototype Model we covered in Chapter 2 and registering our component to inherit from `Sys.Component`.

ErrorHandler Component

To demonstrate how to define a new component, we create a new error handling component. The `ErrorHandler` component will be responsible for publishing handled and unhandled errors to an error data service.

Skeleton

To start, we create the skeleton of our new component, as Listing 3.1 shows.

Listing 3.1 Defining Our ErrorHandler Component

```
/// <reference name="MicrosoftAjax.js"/>
ErrorHandler = function() {
  ErrorHandler.initializeBase(this);
};

ErrorHandler.prototype = {
  initialize: function() {
    ErrorHandler.callBaseMethod(this, 'initialize');
  },
  dispose: function() {
    ErrorHandler.callBaseMethod(this, 'dispose');
  }
}
ErrorHandler.registerClass('ErrorHandler', Sys.Component);
```

Besides calling `initializeBase` in the constructor and registering our class to inherit from `Sys.Component`, we overrode `Sys.Component`'s `initialize` and `dispose` methods. We included these overrides in the skeleton because overriding the `initialize` and `dispose` methods is normally the first step taken in creating a new component, and we suggest doing it right away.

Build Up and Tear Down

We can build on our skeleton definition by providing an implementation of our `initialize` and `dispose` methods.

In the `initialize` method, you build up your component. This includes adding event handlers to DOM elements, appending a new DOM element to the tree, or anything else your component requires.

In the `dispose` method, you tear down your component. This might include detaching an event from a DOM element, destroying a created DOM element, or releasing any other resources that your component created.

> **TIP dispose May Be Called More Than Once**
>
> It's a good habit to write your `dispose` method so that it can be called more than once without causing any runtime errors. With a decently complex application, it's likely you'll get into a situation where when some manager object is disposed it will call `dispose` on its child components. But, each of the child components will have also been registered as a disposable object with `Sys.Application`. When `Sys.Application` disposes and executes `dispose` on each of the registered disposable objects, it will be the second (or more time) that `dispose` will have been called on them. If you're not careful, this can cause a runtime error. Simple if-then checks can prevent most common problems.

For our new `ErrorHandler` component, we need to add a handler to the window's `error` event when the component initializes and then remove the handler when our component disposes. Listing 3.2 shows how we do this.

LISTING 3.2 Adding a Handler to Window's error Event

```
/// <reference name="MicrosoftAjax.js"/>
ErrorHandler = function () {
  ErrorHandler.initializeBase(this);
};

ErrorHandler.prototype = {
  initialize: function () {
    ErrorHandler.callBaseMethod(this, 'initialize');
    window.onerror =
      Function.createDelegate(this, this._unhandledError);
  },

  dispose: function ErrorHandler$dispose() {
    window.onerror = null;
    ErrorHandler.callBaseMethod(this, 'dispose');
  },

  _unhandledError: function(msg, url, lineNumber) {
    try {
      var stackTrace = StackTrace.createStackTrace(arguments.callee);
      ErrorDataService.PublishError
        (stackTrace, msg, url, lineNumber);
    }
    catch (e) { }
  }
}
ErrorHandler.registerClass('ErrorHandler', Sys.Component);
```

As you can see in Listing 3.2, we did some interesting things. First, in the `initialize` method, we created a delegate that pointed to the `_unhandledError` method and assigned it to the window's `error` event using the `onerror` assignment.

> **TIP window.onerror**
>
> We used the `onerror` assignment rather than the `$addHandler` method because for some reason the window's `error` event doesn't support adding events through `addEventListener` or `attachEvent`, the two browser-specific methods that `$addHandler` eventually calls.

In the `dispose` method, we went ahead and cleared the window's `error` event handler. This is the buildup and teardown of our component.

In the `unhandledError` method that will execute when an unhandled error occurs, we do two things. First, we generate a stack trace using a global `StackTrace` object passing in the `callee` property of the function's `arguments` variable. After we have our stack trace, we execute the `PublishError` method on our `ErrorDataService` web service proxy, passing to the server the stack trace, the error message, the URL of the page where the error occurred, and the line number of the error message. We also wrapped all the code in a try-catch statement because we don't want the error handling code to throw any runtime errors itself.

> **NOTE StackTrace and ErrorDataService**
>
> The global `StackTrace` object we used to generate our stack trace of the executing call stack is really useful for debugging, and its full source code is available in Appendix D, "Client Error Handling Code." Similarly, the `ErrorDataService` web service that we used to send the error information back to the server for processing can be found in Appendix D.

Using Base Class Methods and Objects

By inheriting from `Sys.Component`, our type inherits all the attributes and behaviors of `Sys.Component`. Using the base class's `Sys.EventHandlerList` object and its related functionality, we can define new events without having to write much code ourselves. Listing 3.3 expands our basic `ErrorHandler` component and adds an event that we can register with that will be raised whenever an error occurs.

Listing 3.3 Using Base Class Methods

```
… // code remains the same as before.

  _unhandledError: function (msg, url, lineNumber) {
    try {
      var stackTrace =
      StackTrace.createStackTrace(arguments.callee);
      ErrorDataService.PublishError
        (stackTrace, msg, url, lineNumber);

      var args = new ErrorEventArgs(stackTrace, msg, url, lineNumber);
      this._raiseUnhandledErrorOccured(args);
```

```
    }
    catch (e) { }
  },

  add_unhandledErrorOccurred: function(handler) {
    this.get_events().addHandler("unhandledErrorOccurred", handler);
  },

  remove_unhandledErrorOccurred: function(handler) {
    this.get_events().removeHandler("unhandledErrorOccurred", handler);
  },

  _raiseUnhandledErrorOccured: function(args) {
    var evt = this.get_events().getHandler("unhandledErrorOccurred");
    if (evt !== null) {
      evt(this, args);
    }
  },
}
ErrorHandler.registerClass('ErrorHandler', Sys.Component) ;

ErrorEventArgs = function(stackTrace, message, url, lineNumber) {
  ErrorEventArgs.initializeBase(this);
  this._message = message;
  this._stackTrace = stackTrace;
  this._url = url;
  this._lineNumber = lineNumber;
}
ErrorEventArgs.registerClass("ErrorEventArgs", Sys.EventArgs);
```

Starting from the bottom of Listing 3.3, we define the new `ErrorEventArgs` type. This type inherits from `Sys.EventArgs` and turns our error information into an object.

In the `ErrorHandler` type, we added the three methods necessary to add, remove, and raise the `unhandledErrorOccurred` event. We rely on `Sys.Component`'s event handler list, which we access through `this.get_events()` to maintain the list of events.

Finally, in the `_unhandledError`, we added code to create the `error` event arguments and then pass them on to the method that raises the event.

One final change that we make to our `ErrorHandler` component is to add a property that allows us to enable or disable the error publishing feature. Listing 3.4 shows the code changes.

Listing 3.4 Adding the Disable Error Publishing Property

```
/// <reference name="MicrosoftAjax.js"/>
ErrorHandler = function () {
```

Listing 3.4 continued

```
    ErrorHandler.initializeBase(this);
    this._disableErrorPublication = false;
};

ErrorHandler.prototype = {
…
    get_disableErrorPublication: function() {
        return this._disableErrorPublication;
    },

    set_disableErrorPublication: function(value) {
        if (!this.get_updating()) {
            this.raisePropertyChanged("disableErrorPublication");
        }
        this._disableErrorPublication = value;
    },

    _unhandledError: function(msg, url, lineNumber) {
        try {
            var stackTrace = StackTrace.createStackTrace(arguments.callee);
            if (!this._disableErrorPublication) {
                ErrorDataService.PublishError
                    (stackTrace, msg, url, lineNumber);
            }
            var args =
                new ErrorEventArgs(stackTrace, msg, url, lineNumber);
            this._raiseUnhandledErrorOccured(args);
        }
        catch (e) { }
    },
    …
}
ErrorHandler.registerClass('ErrorHandler', Sys.Component);
…
```

With that final change, we've created a useful component that we can use to send client error information to the server so that we can be aware of issues our clients are experiencing.

Creating Components

Based on what we've covered so far, you might think that to create a new component you would "new up" a component and assign it to a variable, as shown in Listing 3.5.

Listing 3.5 Creating an Instance of a Component Using new

```
var errorHandler = new ErrorHandler();
errorHandler.set_disableErrorPublication (false);
```

Although nothing is wrong with this code, after all a component is just a JavaScript object, components should be created through the `Sys.Component.create` method. Listing 3.6 shows the syntax for using the `create` method.

Listing 3.6 Creating an Instance of a Component Using Sys.Component.create

```
var newComponent =
  Sys.Component.create(
    type,
    properties,
    events,
    references,
    element);
```

The `Sys.Component.create` method, which can also be accessed through the global variable `$create`, does more than just create a new instance of a particular type. Instead, it creates an instance of a particular type, registers the instance with `Sys.Application` as a managed component, and automatically calls the component's `beginUpdate`, `endUpdate`, `updating`, and `initialize` methods. In addition to doing all this automatically, depending on what parameters are provided to the call, `$create` can assign initial property values, add event handlers to events, assign other components as references, and associate a DOM element to the component. Finally, the `$create` method returns a pointer to the created instance. So as you can see, the `$create` method does a lot more than create a new instance of a type.

> **NOTE Initialize Execution**
>
> `Initialize` always executes after all properties, events, and references have been set.

Importantly, the `$create` method not only creates instances of types that directly inherit from `Sys.Component`, but can also create types that have multiple levels of inheritance before reaching the `Sys.Component` type. This includes controls that inherit from `Sys.UI.Control` and behaviors that

inherit from `Sys.UI.Behavior`. The `$create` method works a little bit differently when creating a behavior or control, and we cover this slight difference when we cover those types later in this chapter.

> **NOTE Parameter Information**
>
> The only parameter required by the `$create` method is `type`. The other parameters—`properties`, `events`, `references`, and `element`—are all optional parameters. If you don't want to use them, supply `null`.
>
> Supplying a value other than `null` for the element parameter is valid only when the type you're creating an instance of inherits from `Sys.UI.Control` or `Sys.UI.Behavior`. If you pass in an element when creating a component that does not inherit from either of these types, an error is thrown.
>
> Likewise, if you do not pass in an element when creating an instance of a type that inherits from `Sys.UI.Control` or `Sys.UI.Behavior`, an error will be thrown.

To demonstrate how to use the `$create` method, we walk through a series of `$create` calls changing the parameters around to suit our demonstration purposes.

> **NOTE Sys.Application Is Initialized**
>
> The following explanation of the `$create` method assumes that `Sys.Application` has been initialized. Although components will not always be created under this condition, we chose this assumption for the initial walkthrough of the `$create` method so that we could have a clear path through the method without too many branches.
>
> However, there are a couple of significant differences between creating components after `Sys.Application` is initialized and creating components before `Sys.Application` is fully initialized, and we point out how the `$create` method changes when `Sys.Application` isn't fully initialized when we discuss `Sys.Application`'s initialization process in Chapter 4.

Using the type Parameter

First, let's look at the basic call where we only pass in the type we want to create and `null` values for the rest of the parameters. Listing 3.7 shows this call.

LISTING 3.7 The type Parameter

```
var errorHandler =
  Sys.Component.create(
    ErrorHandler,
    null,
    null,
    null,
    null);
```

type

Description: The `type` of component to create

Expected type: `type`

Required: Yes

Other requirements: The value assigned to `type` must inherit from `Sys.Component`.

Notes: The parameter is not enclosed in quotation marks because it's a `Type` object, not a string. A `Type` object is a `Function` object that has been registered with the Microsoft AJAX Library using the `registerClass, registerInterface, or registerEnum` method such as we did with the `ErrorHandler` component.

In this example, the first thing the `Sys.Component.create` method does is ensure that the parameter value `ErrorHandler` is a `Type` and inherits from `Sys.Component`. After it passes those tests, it creates a new instance of `ErrorHandler` and assigns it to a local variable.

> **NOTE Registering as a Disposable Object**
>
> When the new instance is created, it registers as a disposable object with `Sys.Application`. Doing so ensures that the instance's `dispose` method is executed when `Sys.Application` is disposed. We cover this topic further in Chapter 4.

Then, on our new instance, `beginUpdate` is executed. By default, `beginUpdate` does nothing more than set the internal updating flag to `true`, but it can be overridden by the new component's implementation to do more work if necessary.

Then, on our new instance, `endUpdate` is executed, which sets the internal `_updating` flag back to `false` and then executes the `initialize` method, which we overrode to attach an event handler to the window's `error` event. Once the `initialize` method has executed, the `updated` method executes. If a method override is not supplied, the `updated` method doesn't do anything. From there, the component is returned, and you can access it through a variable assigned to the method call.

> **TIP _initialized Check**
>
> In `endUpdate`, there is a check to make sure the internal member `_initialized` is `false` before `initialize` is called. In the case of the `$create` method, `initialized` will always be `false` when `endUpdate` is called. However, if you use `endUpdate` for a different purpose later in the component's lifecycle, `_initialized` will be set to `true`, and the `initialize` method won't execute again. This allows you to call `beginUpdate` and `endUpdate` without having to worry about your component being re-initialized.

This simple example tells us one important thing through the power of omission. Nowhere did we say that the component got added to `Sys.Application`'s managed objects, which is something we claimed the `Sys.Component.create` method did. This didn't happen because the ID of the component was never set, and only components that have their IDs set are automatically added to `Sys.Application`'s managed components. We can correct this by manually setting the component's ID and then adding it to `Sys.Application`'s list of managed components, as shown in Listing 3.8.

Listing 3.8 Manually Setting the ID and Calling addComponent

```
var errorHandler =
  Sys.Component.create(
    ErrorHandler,
    null,
    null,
    null,
    null);

errorHandler.set_id("ApplicationErrorHandler");
Sys.Application.addComponent(errorHandler);
```

> **NOTE Calling addComponent**
>
> If we were to manually add the component to `Sys.Application` without setting the component's ID, an error would be thrown.

Another way to correct this problem is to initially set the component's ID. If the ID is set using the `properties` parameter, the component will automatically be added to `Sys.Application`'s managed components right after the `events` parameter is processed. Listing 3.9 shows the change required.

LISTING 3.9 Setting the Component's id Inline

```
var errorHandler =
  Sys.Component.create(
    ErrorHandler,
    {id: "ApplicationErrorHandler"},
    null,
    null,
    null);
```

Because having a value for the component's ID is necessary for it to become a managed component, it should almost always be set in the `$create` call. There might be special cases where you don't want to set it or want to set it a later time, but these will be rare.

Also, the IDs of components that are managed by `Sys.Application` must be unique. If you attempt to add two components with the same ID to `Sys.Application's` managed components either through the `$create` statement or manually calling `addComponent`, an error will be thrown.

> **NOTE Using the returned Variable**
>
> The `$create` method enables us to access the created component through a pointer returned by the method, but it's useful only in certain situations. Because the component is registered with `Sys.Application`, we'll be able to get access to the component later by finding it within `Sys.Application`'s managed components using its unique ID.

Using the properties Parameter

In this example let's pass in some simple initial property values. Listing 3.10 shows how we do this.

Listing 3.10 Passing In Initial Property Values

```
var errorHandler =
  Sys.Component.create(
    ErrorHandler,
    {
      id: "ApplicationErrorHandler",
      disableErrorPublication: true
    },
    null,
    null,
    null);
```

properties

Expected type: `Object`

Required: No

Description: An object containing key-value pairs, where the key is the name of a property on the component to set, and the value is the value to assign to that property

In this example, the initial steps of the `$create` method are the same as they were in the previous example. The type is validated, the component is created, and `beginUpdate` is executed.

The next step is to assign the property values to the component's properties. The properties and their values are passed in using the string-object syntax that is highlighted in Listing 3.10. Instead of using the string-object syntax, we could have used object creation code as shown in Listing 3.11, but the string-object syntax is a shorter and more comprehensible syntax in this situation.

Listing 3.11 Creating Properties Object Using Variables

```
var initialProperties = new Object;
initialProperties.id = "ApplicationErrorHandler";
initialProperties.disableErrorPublication = true;

var errorHandler =
  $create(
    ErrorHandler,
    initialProperties,
    null,
```

```
        null,
        null);
```

Using either method, our code indicates that we want to set two properties: `id` and `disableErrorPublication`.

To do this, the `$create` method delegates control to another method, `Sys$Component_setProperties`. This is a global method available within the Microsoft AJAX Library, whose purpose is to set properties on a component. It accepts two parameters: the `target` object and the `properties` object.

Within this method, each of the expando properties attached to the `properties` parameter is accessed and successively processed according to a series of rules.

The first rule determines whether there is a getter method for the property. It does this by prefixing the current property name, `id`, with the string `get_`. In our example, the `get_id` method exists on the base `Sys.Component` class, so this rule is met.

After the getter method has been established as existing, the setter method is looked for. It does this by prefixing the current property name with the string `set_`. Again, in our example, the `set_id` method exists on the base `Sys.Component` class.

After the setter method has been determined to exist, the setter method is executed on the target, passing in the value of the current property. In our example, the value passed in is `ApplicationErrorHandler`.

The process repeats until all properties have been successfully applied to the component or an error occurs, such as the getter not existing, a setter not taking in the correct number of parameters, or a whole host of other possibilities. In our example, the `disableErrorPublication` property is initialized with the value `true`.

> **NOTE** **Iterating over the Properties**
>
> The expando properties attached to the `properties` parameter are accessed by using a for…in loop. As we discussed in Chapter 1, "Programming with JavaScript," the for…in loop iterates over the properties of an object, placing the current property name into a variable.
>
> After the property name is placed in a variable, the value associated to the property name is accessed using the associative array principle discussed in Chapter 1.

Calling the Setter Method

As mentioned, the setter method for a property is executed during the creation of a component. Just as in .NET, the setter method can contain any code it wants. If you write the method to execute a long-running process, the component creation waits until that process has completed.

With that in mind, you need to be careful to make the setter method as efficient as possible for the component creation to complete quickly.

If there is extra code that should not execute during the component creation process, one way to avoid executing it is to check the value of `_updating`, as shown here:

```
…
set_disableErrorPublication: function(value) {
  if (!this.get_updating()) {
    this.raisePropertyChanged
      ("disableErrorPublication");
  }
  this._disableErrorPublication = value;
},
…
```

In this code example, we make sure that the component is not updating before we raise the `propertyChanged` event. Checking the `_updating` flag is less expensive than going through the process of raising the event.

Caution: This is just test code. Make sure that you really don't want the `propertyChanged` event to be raised when the component is updating before using this code.

Complex Property Setting

Setting the `id` and the `disableErrorPublication` of our `ErrorHandler` instance are simple examples of property settings through the `$create` method, but there are four advanced scenarios of property setting that we can use to our advantage to create complex components in a single statement.

1. Setting a value that has no setter or getter, such as an attribute on a DOM element or a property attached to a prototype
2. Appending items to an array
3. Setting properties on a subcomponent; a component contained within another component
4. Adding properties to an existing object

Each of these concepts is illustrated with a new dummy component `MyComplexComponent`:

```
MyComplexComponent = function() {
  MyComplexComponent.initializeBase(this);
  this.city = null;
  this._areaCodes = [];
  this._myObject = { firstName: "Harry" };
  this.subComponent =
    $create(ErrorHandler,
      null,
      null,
      null,
      null);
};

MyComplexComponent.prototype = {
  someExpandoProperty: null,
  get_address: function() {
    return this._address;
  },
  set_address: function(value) {
    this._address = value;
  },
  get_areaCodes: function() {
    return this._areaCodes;
  },
  get_myObject: function() {
    return this._myObject;
  }
};

MyComplexComponent.registerClass(
  "MyComplexComponent",
  Sys.Component);

var newComponent =
  $create(
```

```
MyComplexComponent,
{
  id: "MyNewComplexComponent",
  city: "San Diego",
  areaCodes: [619, 858, 760],
  someExpandoProperty: "My Expando's Value",
  subComponent:
  {
    id: "ApplicationErrorHandler",
    disableErrorPublication: "true"
  },
  myObject: { lastName: "Houdini" }
},
null,
null,
null);
```

1. Setting a value that has no getter or setter
   This scenario is exemplified through the setting of the `city` and `someExpandoProperty` properties. These properties can be set because they are existing fields on the object. If they didn't already exist, the `setProperties` routine wouldn't add them for us.
2. Appending items to an array
   The second advanced scenario is exemplified through the `areaCodes` property. Here, we define a new array of three elements (619, 858, and 760) and assign it to the `areaCodes` property. For the elements to be appended to the existing array, there must be a getter for the property, but no setter. If there is a setter, it will be used instead, and it will be up to the setter's code to append the items to the array. Also, the array must already be instantiated. If the variable points to null or undefined, an error is thrown.
3. Setting properties on a subcomponent
   The third advanced scenario is exemplified through the `subComponent` property. Here, we define a subobject that contains an `id` and `disableErrorPublication` property, both properties on our previously defined `ErrorHandler` component. When the `setProperties` method encounters this property, it accesses the

subcomponent, and then recursively calls the `setProperties` method using the subcomponent as the `target` parameter and the subobject containing the `id` and `address` properties as the `properties` parameter. This type of recursive call could continue an infinite number of levels deep if we had set up our `properties` parameter that way.

We could have supplied a getter here and had the same effect, but if we had supplied a setter, too, setting the properties of the subcomponent would not have worked as expected.

When we call the `setProperties` method recursively using a component as the target parameter, it calls `beginUpdate` on that component before it enters the for...in loop and `endUpdate` when it exits. This is something to be aware of if you're using the `get_updating` method in your code.

4. Setting properties on a simple JavaScript object

   The fourth and final advanced scenario is exemplified through the `myObject` property. The `myObject` property defines a simple object containing the property `lastName` that has the value `Houdini`. When the `setProperties` method encounters this property, it makes a recursive call into the `setProperties` method to apply the new properties to the `myObject` member. Here, rather than pass in a component as the target, `myObject` is passed in as the `target` parameter and the new properties object is passed in as the `properties` parameter.

As you can see, the `properties` parameter of the `$create` method can handle some advanced scenarios. You'll find use for them in your code, if you remember that they're there.

Using the events Parameter

In this example, let's assign an event handler to the available initialized event using the `events` parameter. The code in Listing 3.12 demonstrates how to do this.

Listing 3.12 Passing in Event Handlers

```
$create(
  ErrorHandler,
  {
    id:"ApplicationErrorHandler",
    disableErrorPublication: true
  },
  {
    unhandledErrorOccurred:
      function(sender, args) {
        alert(args._stackTrace);
      }
  },
  null,
  null);
```

events

Expected type: `Object`

Required: No

Description: An object containing key-value pairs, where the key is the name of an event on the component to assign to, and the value is an event handler to add to the event

In this example, the initial steps of the `$create` method are the same as they were in the previous `properties` example. The type is validated, the component is created, `beginUpdate` is executed, and then the properties are set.

After the properties are set, the `events` parameter is processed. Similar to the `properties` property, the `events` parameter is an object that contains a series of key-value pairs. The `events` object is iterated over, and each key-value pair is used to add an event handler to an event until they are all added or an error occurs. Again, similar to the `properties` parameter, the event handlers are added to events by executing the appropriate method. In this case, the key, `unhandledErrorOccurred`, is automatically prefixed with `add_` to create `add_unhandledErrorOccurred`. This string is then looked for as a function contained within the component's definition. If the method `add_unhandledErrorOccurred` is successfully found and the value of the key-value pair contained in the object is a `Function` object, the value is passed into the `add_unhandledErrorOccurred` method as its parameter and executed, adding the event handler to the event.

In our example `$create` statement, we defined the event handler in line with the `$create` statement, and our event handler is successfully added to the `unhandledErrorOccurred` event. Another way to do this is to predefine an event handler function, as we show in Listing 3.13.

Listing 3.13 Predefining an Event Handler

```
function unhandledErrorHandler(sender, args) {
  alert(args._stackTrace);
}

$create(
  MyComponent,
  {address: "123 N. Fake Street" },
  {
    unhandledErrorOccurred:
      unhandledErrorHandler
    }
  },
  null,
  null);
```

Predefining the event handler allows it to be reused for other components or to be called procedurally.

In addition, if we want to handle an event with a method that is contained within our component, rather than use a global function as we did in Listing 3.13, we have to go through an extra step of creating a delegate to wrap our event handler method so that context gets pointed back to the intended component, as shown in Listing 3.14.

Listing 3.14 Wrapping an Event Handler in a Delegate

```
MyOtherComponent = function() {
  MyOtherComponent.initializeBase(this);
  this._subComponent = null;
};

MyOtherComponent.prototype = {
  _unhandledErrorOccurred: function(sender, args) {
    var stackTrace = args._stackTrace;
    if (typeof(stackTrace) != "undefined") {
      alert ("The Stack Trace of the error was: " + stackTrace);
    }
  },
```

Listing 3.14 continued

```
    initialize: function() {
      MyOtherComponent.callBaseMethod(this, "initialize");

      this._errorHandler =
        $create(
          ErrorHandler,
          {
            id:"ApplicationErrorHandler",
            disableErrorPublishing: true
          },
          {
            unhandledErrorOccurred:
              Function.createDelegate (
                this,
                this._unhandledErrorOccurred
              )
          },
          null,
          null);

      // cause an error to be thrown
      var nullObj = null;
      nullObj.causeError;
    }
  };

  MyOtherComponent.registerClass("MyOtherComponent", Sys.Component);
```

As shown in the highlighted text, we create an instance of the `ErrorHandler` component in `MyOtherComponent`'s `initialize` method. When we assign the event handler to the `unhandledErrorOccurred` event, we wrap it in a delegate so that when the code goes to execute the `_unhandledErrorOccurred` method it executes is using the correct context.

> **NOTE Functional Prefixes**
>
> Now that we've covered setting properties and adding event handlers through the `$create` method, we can see how the property prefixes `get_` and `set_` and the event handler prefix `add_` are not only aesthetic prefixes but also functional.

Using the references Parameter

With the `references` parameter, we can assign one component to a property on another, thus linking them together. You might wonder why we would need a separate parameter for this when we could already accomplish this using the `properties` parameter. We need this parameter because when we start using server code to create instances of client components, we won't know what order the components will be created in. If we use a separate parameter, the initialization process that `Sys.Application` goes through to create our components treats component references differently and delays assigning them until all components have been created. Doing this eliminates the problem of a component attempting to access an uncreated component.

To illustrate how to use the `references` parameter, we pass in one component as a reference to another component in the `$create` statement using the `references` parameter. To do that, we must first create a component that can act as a reference to our second component. Listing 3.15 shows the two `$create` statements. In this example, we use two fictitious components to keep the example clear.

LISTING 3.15 Assigning References

```
// create the first component
$create(
  MyComponent,
  {
    id: "MyFirstComponent"
  },
  null
  null,
  null
);

// create the second component and assign the first component
// to a property called subComponent
$create(
  MyComponent,
  {
    id: "MySecondComponent"
  },
  null,
  {
    subComponent:"MyFirstComponent"
  },
  null
);
```

references

Expected type: `Object`

Required: No

Description: An object that contains key-value pairs, where the key represents a component property, and the value represents a component to assign to this property. The value is the `id` of the component.

After the `$create` statement has passed the event's assignment code, it processes the `references` parameter. Similar to the `properties` and `events` parameters, the `references` parameter is an object that contains key-value pairs. The key is the property we want to assign, and the value is an `id` of a component we want to assign to the property.

In Listing 3.15, the `references` object is highlighted. The object states that we want to assign the component that has the `id` `MyFirstComponent` to the `subComponent` property of the component being created. Just like the `setProperties` method we discussed earlier, the `setReferences` method looks for a setter method that's defined by prefixing `set_` to the property name. In our example, this method's name is `set_subComponent`. When this method is found, the component `id`, `MyFirstComponent`, is looked for within `Sys.Application`'s managed components. If the component is found, the setter method is executed with the found component as its parameter.

> **NOTE Finding Managed Components**
>
> Through `Sys.Application.find`, we can find registered components by ID. When we cover `Sys.Application` in Chapter 4, we cover the `find` method in detail.

> **TIP Creation Order**
>
> As mentioned earlier, for this code sample to work correctly, `MyFirstComponent` must be available before the second `$create` statement executes. References to uncreated components can be used if `Sys.Application` is in its initialization phase. This is something we cover in Chapter 4.

Using the element Parameter

The last parameter of the `$create` method is `element`, which is used as a pointer to a DOM element. Because the `element` parameter is valid only when we're creating a new behavior or a new control, we cover the `element` parameter when we cover defining and creating those types.

Wrapping Up Components

A component is not just defined as an object that inherits from `Sys.Component`, but also as being managed by `Sys.Application`. Creating an instance of a type that inherits from `Sys.Component` using the `new` keyword will not automatically register the instance with `Sys.Application`. We have to use the `$create` method for this to happen. Using `$create` also facilitates setting properties, wiring up event handlers, assigning references to other components, and associating it with a DOM element, as we see with controls and behaviors. It also automatically calls the `initialize` method on the component, enabling you to create user-defined code that executes after all the properties have been set, event handlers added, and component references assigned.

Controls

A control is a special type of component directly associated to a DOM element. A DOM element can have *only* one associated control, and a control *must* be associated to a DOM element.

In practical terms, because we can have only one control associated to a given DOM element, their use is intended for situations where you want to have full power over the DOM element. In those cases where you're not sure whether that's your intention, start off with a behavior, and then move to a control if needed. In reality, switching back and forth between a control and a behavior is not too difficult and doesn't require too much code to be altered.

Because a control is directly tied to a DOM element, it has methods that are useful for accessing and manipulating the associated DOM element. Table 3.5 details the methods available to a control that access and manipulate the associated DOM element.

TABLE 3.5 Sys.UI.Control Methods

Method Name	Description	Syntax
`set_id`	Overrides component's `set_id` method. Throws an error because a control's `id` is always the associated DOM element's `id`.	`no valid usage`
`get_id`	Overrides component's `get_id` method. Returns the `id` of the associated DOM element.	`return ctrl.get_id();`
`get_visible`	Returns the value returned by calling `Sys.UI.DOMElement.getVisible` on the associated DOM element.	`return ctrl.get_visible();`
`set_visible`	Calls `Sys.UI.DomElement.setVisible` using the control's associated DOM element and the `Boolean` value passed into the `set_visible` call.	`ctrl.set_visible(`*`visibility`*`);`
`get_visibilityMode`	Calls `Sys.UI.DomElement.getVisibilityMode` using the control's associated DOM element.	`return ctrl.get_visibilityMode();`

`set_visibilityMode`	Calls `Sys.UI.DomElement.setVisibilityMode` using the control's associated DOM element and `Sys.UI.VisibilityMode` parameter passed in to the `set_visibilityMode` call.	`ctrl.set_visibilityMode(`
		`  `*`Sys.UI.VisibilityMode`*
		`);`

`get_element`	Returns the associated DOM element.	`return ctrl.get_element();`

Method Name	Description	Syntax
`addCssClass`	Calls `Sys.UI.DomElement.addCssClass` using the control's associated DOM element and the name of the CSS class to add.	`ctrl.addCssClass(cssClassName)`
`removeCssClass`	Calls `Sys.UI.DomElement.removeCssClass` using the control's associated DOM element and the name of the CSS class to remove.	`ctrl.removeCssClass(cssClassName);`
`toggleCssClass`	Calls `Sys.UI.DomElement.toggleCssClass` using the control's associated DOM element and the name of the CSS class to toggle.	`ctrl.toggleCssClass(cssClassName);`
`dispose`	Overrides `Sys.Component`'s dispose. Calls base class `dispose`, sets `element`'s control expando property to `undefined`, and deletes reference to the DOM element from the component.	`ctrl.dispose();`

New Concepts

Besides the methods that access and manipulate the associated DOM element, other methods introduce two new concepts: a control's parent and ASP.NET-like event bubbling.

Control's Parent

A control's parent property provides a pointer to another control. The parent can be calculated in one of two ways. If a parent has been explicitly set using the `set_parent` method, that is the control's parent. If a parent has not been explicitly set, the control's associated DOM element's `parentNode` pointer is walked until an element with a control attached to it is reached, and that is considered the control's parent.

Table 3.6 details the methods involved with the parent pointer concept.

Table 3.6 Sys.UI.Control Methods Related to Control's Parent

Method Name	Description	Syntax
`get_parent`	Returns the explicitly set parent or the first control encountered by walking up the DOM element's `parentNode` pointer	`var parent = ctrl.get_parent();`
`set_parent`	Explicitly sets the parent	`ctrl.set_parent(otherCtrl);`

Event Bubbling

Event bubbling is a method of passing events up through the parent pointer and giving parent controls the opportunity to handle those events.

Event bubbling in the Microsoft AJAX Library is similar to event bubbling using controls in ASP.NET. A control starts the process by calling `raiseBubbleEvent`, passing in a source and event arguments. In the `raiseBubbleEvent` method, the control's parent is retrieved using the `get_parent` method attached to the control, and `onBubbleEvent` is called on it. The default implementation of `Sys.UI.Control`'s `onBubbleEvent` method returns `false`, which indicates that the control did not handle the event and the bubbling should continue up the hierarchy.

If the control wants to handle the bubbled event, it may do so by overriding the default implementation of `onBubbleEvent`. In the overridden method, it can decide whether the bubbling should continue or stop. If it wants to stop the event's propagation up the parent hierarchy, it returns `true`. If it wants to allow other controls higher up in the control's parent tree the opportunity to handle the event, too, it returns `false`.

Table 3.7 details the methods involved with the event bubbling concept.

TABLE 3.7 Sys.UI.Control Methods Related to Event Bubbling

Method Name	Description	Syntax
`onBubbleEvent`	Part of the event bubbling framework. Needs to be overridden to provide functionality. Returns `false` by default.	Is automatically called by `raiseBubbleEvent`
`raiseBubbleEvent`	Part of the event bubbling framework. Walks the control's parent list, firing the `onBubbleEvent` on each parent object.	`ctrl.raiseBubbleEvent (source, args);`

NOTE Additional Methods

Because `Sys.UI.Control` inherits from `Sys.Component`, all the methods available to `Sys.Component` are available to `Sys.UI.Control`.

Defining a New Control

Like defining a new component, defining a new control follows the Prototype Model we covered in Chapter 2. To illustrate how to define a new control, we create a new control that attaches to a textbox and allows only numbers to be entered. Listing 3.16 shows the code necessary to define the new `NumberOnlyTextBox` control.

LISTING 3.16 Defining a New Control Type

```
/// <reference name="MicrosoftAjax.js"/>
NumberOnlyTextBox = function(element) {
  NumberOnlyTextBox.initializeBase(this, [element]);
  this._keyDownDelegate = null;
};

NumberOnlyTextBox.prototype = {
  initialize: function() {
    NumberOnlyTextBox.callBaseMethod(this,'initialize');
    this._keyDownDelegate =
      Function.createDelegate(this, this._keyDownHandler);
    $addHandler(this.get_element(), "keydown", this._keyDownDelegate);
  },
```

Listing 3.16 continued

```
    dispose: function() {
      $removeHandler
        (this.get_element(), "keydown", this._keyDownDelegate);
      this._keyDownDelegate = null;
      NumberOnlyTextBox.callBaseMethod(this, 'dispose');
    },

    _keyDownHandler: function(e) {
       return ((e.keyCode >= 48 && e.keyCode <= 57) || (e.keyCode == 8));
    }
  };

  NumberOnlyTextBox.registerClass("NumberOnlyTextBox", Sys.UI.Control);
```

As shown in Listing 3.16, we can see that there are two major differences between our `NumberOnlyTextBox` control and our `ErrorHandler` component we declared previously.

First, the base class of our `NumberOnlyTextBox` control is `Sys.UI.Control` and not `Sys.Component.`

Second, the constructor takes an `element` parameter and passes it to the base class's constructor through the `initializeBase` method. This parameter is the DOM element that is going to be associated to the control.

When the element is passed to `Sys.UI.Control`'s constructor, three things happen. First, the DOM element is checked to make sure that there is no other control already associated to it. If this test fails, the constructor throws an error, and the control's creation fails. If it passes, the second step the constructor takes is to assign the DOM element to the internal member `_element`. Finally, the control is assigned to the DOM element using the expando property `control`. If we created a reference to the associated element, we could access the assigned control using the following code:

```
$get("TextBox1").control;
```

Using our newly minted `control` type, Listing 3.17 demonstrates the association requirements we just discussed.

Listing 3.17 Creating an Instance of MyControl Using new

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Control Testing!</title>
</head>
```

```
<body>
  <form id="form1" runat="server">
  <asp:ScriptManager ID="SM1" runat="server" />

  // omitted NumberOnlyTextBox definition for brevity.

  <asp:TextBox ID="txtBox1" runat="server" Width="150px" />

  <script type="text/javascript">

    var numberOnlyTextBox =
      new NumberOnlyTextBox($get("txtBox1"));

    // alerts "txtBox1"
    alert ("numberOnlyTextBox's associated element's id: " +
           numberOnlyTextBox.get_element().id);

    // alerts "txtBox1"
    alert ("numberOnlyTextBox's associated control's id: " +
           $get("txtBox1").control.get_id());

    // throws a JavaScript error because a
    // control is already associated to txtBox1.
    var numberOnlyTextBox2 =
      new NumberOnlyTextBox ($get("txtBox1"));
  </script>

  </form>
</body>
</html>
```

Creating a Control

In Listing 3.17, we used the `new` keyword to create a new instance of our `NumberOnlyTextBox` type. From our component discussion, we know that the `$create` method performs a whole host of tasks besides creating a new instance of the type, and because our new type inherits from `Sys.UI.Control`, which inherits from `Sys.Component`, we can use the `$create` statement in the same manner as we did for our `ErrorHandler` component.

Instead of walking through the entire `$create` method again, we need to discuss only the use of the `element` parameter because that's the only difference. Using the example we created in Listing 3.17 as a basis, we can modify the code to use the `$create` method. Listing 3.18 shows the altered code.

Listing 3.18 Creating an Instance of MyControl Using $create

```
<html>
<head runat="server">
  <title>Control Testing!</title>
</head>
<body>
  <form id="form1" runat="server">
  <asp:ScriptManager ID="SM1" runat="server" />

  // omitted NumberOnlyTextBox definition for brevity.

  <asp:TextBox ID="txtBox1" runat="server" Width="150px" />

  <script type="text/javascript">
    $create(
      NumberOnlyTextBox,
      null,
      null,
      null,
      $get("txtBox1")
    );
  </script>

  </form>
</body>
</html>
```

The highlighted code shows the `$create` method call. Notice how `$get("txtBox1")` is passed in as the `element` parameter of the `$create` method. When the `$create` method instantiates the new instance of the `NumberOnlyTextBox`, it determines whether the `NumberOnlyTextBox` inherits from `Sys.UI.Control` or `Sys.UI.Behavior`. If it does, and in our example it does, it uses the `element` parameter as the parameter for the constructor call; similar to what we did in Listing 3.17 before we used the `$create` method to create our new control.

> **NOTE Setting the Control's id**
>
> Unlike a component or behavior, setting the `id` of a control is not allowed. The `id` of the control is always the `id` of the associated DOM element.

Wrapping Up Controls

Controls are not too different from their base component type. The main difference is that a control must be associated to a DOM element, whereas a component must not be.

Behaviors

A behavior is another special type of component that is related to DOM elements. Like controls, a behavior must be associated to a DOM element. However, unlike a control, there can be more than one behavior attached to a DOM element.

In a practical sense, behaviors define how we want a DOM element to behave. We want the DOM element to *collapse* to a single line, we want the DOM element to *float* on the page, and we want the DOM element to *fill* all the available screen space. These are all examples of behaviors that we might attach to a DOM element.

To help us define new behaviors and use instantiated behaviors, the base `Sys.Behavior` type includes a few more methods than its base `Sys.Component` type. Table 3.8 details these methods.

Table 3.8 Sys.UI.Behavior Methods

Method Name	Description	Syntax
`get_element`	Returns the DOM element associated to the behavior.	`return behavior.get_element();`
`get_name`	Returns the name of the behavior. If the name has been explicitly set, that's the name returned. If it has not been explicitly set, the name returned is the short type name of the behavior.	`return behavior.get_name();`

TABLE 3.8 continued

Method Name	Description	Syntax
`set_name`	Sets the name of the behavior. Explicitly set behavior names must be unique. The name of a behavior cannot be set after the behavior has been initialized. The behavior name must not start with a blank space, end with a blank space, or be an empty string.	`behavior.set_name("HiddenElm");`
`initialize`	Calls the base class's `initialize` method. Attaches the behavior to its associated DOM element by adding an expando property to the DOM element that's the name of the behavior.	`behavior.initialize();`
`dispose`	Overrides `Sys.Component`'s dispose. Calls base class `dispose`, removes the DOM element's expando property that is in the name of the behavior, and deletes the reference to DOM element from the behavior.	`behavior.dispose();`
`get_id`	Returns the underlying component's `id` if it's set. If it's not set, the value returned is the associated DOM element's `id` appended with the behavior's name.	`return behavior.get_id();`

Method Name	Description	Syntax
`Sys.UI.Behavior.getBehaviorsByType`	Returns all behaviors attached to a DOM element that are of a particular type.	`return Sys.UI.Behavior.getBehaviorsByType (element, typeName)`
`Sys.UI.Behavior.getBehaviorByName`	Returns a behavior attached to a DOM element if it was found.	`return Sys.UI.Behavior.getBehaviorByName (element, behaviorName)`
`Sys.UI.Behavior.getBehaviors`	Returns a copy of the behaviors attached to a DOM element. If there are no behaviors for a particular DOM element, returns an empty array.	`return Sys.UI.Behavior.getBehaviors (element);`

Defining a Behavior

Like defining a new component and control, defining a new behavior follows the Prototype Model we covered in Chapter 2. Rather than create a brand new example, we modify the `NumberOnlyTextBox` control example we used in the previous section to work as a behavior instead. Listing 3.19 shows the code necessary to define the `NumberOnlyTextBox` behavior.

Listing 3.19 Defining a Behavior Type

```
/// <reference name="MicrosoftAjax.js"/>
NumberOnlyTextBox = function(element) {
  NumberOnlyTextBox.initializeBase(this, [element]);
  this._keyDownDelegate = null;
};

NumberOnlyTextBox.prototype = {
  initialize: function() {
    NumberOnlyTextBox.callBaseMethod(this,'initialize');
    this._keyDownDelegate =
      Function.createDelegate(this, this._keyDownHandler);
    $addHandler(this.get_element(), "keydown", this._keyDownDelegate);
  },

  dispose: function() {
    $removeHandler
      (this.get_element(), "keydown", this._keyDownDelegate);
```

Listing 3.19 continued

```
        this._keyDownDelegate = null;
        NumberOnlyTextBox.callBaseMethod(this, 'dispose');
      },

      _keyDownHandler: function(e) {
         return ((e.keyCode >= 48 && e.keyCode <= 57) || (e.keyCode == 8));
      }
    };

    NumberOnlyTextBox.registerClass("NumberOnlyTextBox", Sys.UI.Behavior);
```

As you can see, the code to define our `NumberOnlyTextBox` behavior is almost identical to the code necessary to define the `NumberOnlyTextBox` as a new control. The only difference is that a behavior inherits from `Sys.UI.Behavior` rather than `Sys.UI.Control`.

`Sys.UI.Behavior`'s constructor, like `Sys.UI.Control`'s, takes in an element as a parameter. In its constructor, the internal member `_element` is assigned to the `element` parameter, associating the DOM element to the behavior. Then, the behavior is added to the element's `_behaviors` expando property. The `_behaviors` expando property is like the control's `control` property except that it is defined as an array so that more than one behavior can be associated to the DOM element.

Creating a Behavior

From our component and control discussion, we know that using the `$create` method is the correct way of instantiating a new instance of a type that inherits from `Sys.Component`, and a behavior is no different.

In fact, creating a behavior is exactly the same as creating a control, and the code shown in Listing 3.18 will suffice for an example of how to do this.

Unlike controls, however, a couple of problems could appear when creating behaviors, and they both have to do with the uniqueness of the behavior.

Behavior Uniqueness Problems

The first problem is that if a behavior's `id` is not set, the `id` is automatically generated based on the `id` of the associated DOM element and the name of the behavior. Because this is a generated value, it's likely that it could be the

same for more than one behavior. If the same `id` is generated for more than one behavior, when the second behavior attempts to register itself with `Sys.Application`, the registration fails because components managed by `Sys.Application` must have unique `id`s. Listing 3.20 demonstrates this problem.

LISTING 3.20 Creating Behaviors That Have the Same id

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Behavior Testing!</title>
</head>
<body>
  <form id="form1" runat="server">
  <asp:ScriptManager ID="SM1" runat="server" />

  // … NumberOnlyTextBox omitted for brevity.

   <asp:TextBox ID="txtBox1" runat="server" Width="150px" />

  <script type="text/javascript">
    $create(
      NumberOnlyTextBox,
      null,
      null,
      null,
      $get("txtBox1")
    );

    // this will cause an error because the id of the component will
    // be the same as the previous behavior.
    $create(
      NumberOnlyTextBox,
      null,
      null,
      null,
      $get("txtBox1")
    );
  </script>

  </form>
</body>
</html>
```

In this example, because we're not explicitly setting the `name` or the `id` of either behavior, the `id` of each behavior is `txtBox1$NumberOnlyTextBox`. The behaviors' `id`s are computed based on the DOM element's `id`

(`txtBox1`), appended with `$`, followed by the name of the behavior, which when it's not explicitly set is the type name minus any namespaces.

> **NOTE NumberOnlyTextBox**
>
> In our example, the name of our behavior is just the full name of the type: `NumberOnlyTextBox`.
>
> If we were using a namespace for our behavior, for example `MyProject.Behaviors.NumberOnlyTextBox`, the calculated name of the behavior would still be `NumberOnlyTextBox`.

When the second behavior tries to register itself with `Sys.Application`, an error occurs because a component is already registered with that `id`.

To rectify this problem, either the `name` or `id` of the behavior has to be explicitly set. In either case, the `id` or the `name` needs to be unique. In the case of the `id`, it needs to be unique among all components. In the case of the `name`, it needs to be unique among behaviors attached to the associated DOM element. Listing 3.21 shows code that would successfully create the two behaviors and attach them to the same textbox.

Listing 3.21 Setting a Behavior's id

```
<script type="text/javascript">
  $create(
    NumberOnlyTextBox,
    {id: "Behavior1" },
    null,
    null,
    $get("txtBox1")
  );

  $create(
    NumberOnlyTextBox,
    {id: "Behavior2" },
    null,
    null,
    $get("txtBox1")
  );
</script>
```

The second problem with creating behaviors can occur when we attach multiple instances of the same behavior to a DOM element and don't set

their names. Because their names will be the same calculated value (i.e., `NumberOnlyTextBox`), we won't be able find one or more of them through the `Sys.UI.getBehaviorByName` method.

Attaching the multiple instances of the same behavior to a single DOM element might be a rarer case than most, but it can occur. Listing 3.22 shows how we're only able to find one of the `NumberOnlyTextBox` behaviors attached to our textbox.

Listing 3.22 Problems Finding Behaviors by Name

```
<html>
<head runat="server">
  <title>Behavior Testing!</title>
</head>
<body>
  <form id="form1" runat="server">
  <asp:ScriptManager ID="SM1" runat="server" />

  // NumberOnlyTextBox omitted for brevity

  <asp:TextBox ID="txtBox1" runat="server" Width="150px" />

  <script type="text/javascript">
    $create(
      NumberOnlyTextBox,
      {id: "Behavior1"},
      null,
      null,
      $get("txtBox1")
    );

    $create(
      NumberOnlyTextBox,
      {id: "Behavior2"},
      null,
      null,
      $get("txtBox1")
    );

    var beh = Sys.UI.Behavior.getBehaviorByName
      ($get("txtBox1"), " NumberOnlyTextBox ");

    alert (beh.get_name());

    var behaviorsAssignedToDom =
      Sys.UI.Behavior.getBehaviors($get("txtBox1"));

    var behaviors = '';
```

Listing 3.22 continued

```
        for (var i=0; i<behaviorsAssignedToDom.length; i++) {
          behaviors += behaviorsAssignedToDom[i].get_name() + " ";
        }

        // alerts NumberOnlyTextBox NumberOnlyTextBox because
        // there are two behaviors of that name
        alert (behaviors);

      </script>

      </form>
    </body>
    </html>
```

To correct this problem we need to explicitly set the name of any behaviors we create.

To conclude this section on problems with creating behaviors, although an error won't be thrown if a behavior doesn't have its `id`/`name` set when it's created, it's clearly better to do so to avoid some of the rarer problems with behaviors. Therefore, we suggest that you always set the `id` and `name` of a behavior whenever you create an instance of one. Listing 3.23 shows this pattern.

Listing 3.23 Assigning ids and names to Behavior Instances

```
<script type="text/javascript">
  $create(
    NumberOnlyTextBox,
    {id: "Behavior1",
     name: "Behavior1"},
    null,
    null,
     $get("txtBox1")
   );

   $create(
     NumberOnlyTextBox,
     {id: "Behavior2",
      name: "Behavior2"},
     null,
     null,
     $get("txtBox1")
   );
</script>
```

Wrapping Up Behaviors

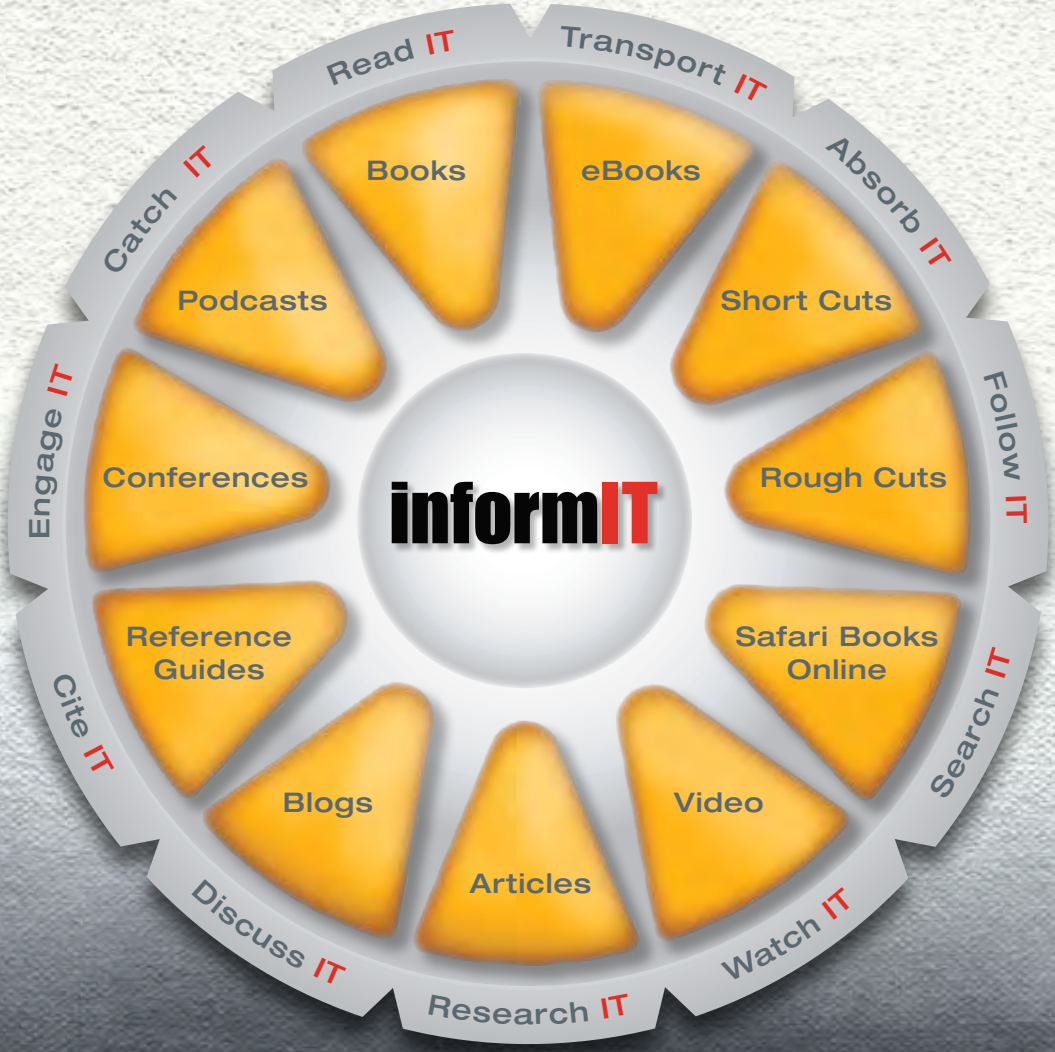
Behaviors are not too different from their base component type. The main difference is that a behavior must be associated to a DOM element, whereas a component must not be. The main difference between a control and a behavior is that a DOM element can have only one control associated to it, whereas a DOM element can have multiple behaviors.

SUMMARY

In this chapter, we examined components, controls, and behaviors. We looked at how the base component type contains commonly used objects and how controls and behaviors extend components to include a reference to a DOM element. We also looked at how you can build them by hand and how they're created using the `$create` function.

In the next chapter, we cover `Sys.Application`, which is the manager object for all components, controls, and behaviors. After we examine `Sys.Application`, we begin to tie the Microsoft AJAX Library into the server portion of ASP.NET AJAX with a chapter on how to create components, controls, and behaviors through .NET code. Finally, we wrap up components, controls, and behaviors with an in-depth look on how to localize them and how they react to being placed inside an UpdatePanel.

LearnIT at InformIT



11 WAYS TO LEARN IT AT www.InformIT.com/learn