Table of Contents

Chapter 19. Partial Rendering: The Easy Way to AJAX......................... 1

The ASP.NET AJAX Infrastructure.................................................................................................................. 2
Selective Page Updates with Partial Rendering.......................................................................................... 34
The AJAX Control Toolkit............................................................................................................................. 57
Conclusion..................................................................................................................................................... 86
Chapter 19
Partial Rendering: The Easy Way to AJAX

In this chapter:
The ASP.NET AJAX Infrastructure...........................................918
Selective Page Updates with Partial Rendering...............................917
The AJAX Control Toolkit......................................................973
Conclusion ............................................................................1002

Gone are the days when a Web application could be architected and implemented as a collection of related and linked pages. The incredible success of the Internet has whetted people's appetite for Web-related technology beyond imagination. Over the years, the users' demand for ever more powerful and Web-exposed applications and services led architects and developers to incorporate more and more features into the server platform and client browser. As a result, the traditional pattern of Web applications is becoming less adequate every day. A radical change in the design and programming model cannot be further delayed.

AJAX is a relatively new acronym that stands for Asynchronous JavaScript and XML. Coined in 2005, it is a sort of blanket term used to describe highly interactive and responsive Web applications. What's the point here? Weren't Web applications created about a decade ago specifically to be "interactive," "responsive," and deployed over a unique tool called the browser? So what's new today?

At the current state of the art, the industry needs more than just an improved and more powerful platform devised along the traditional guidelines and principles of Web applications—a true paradigm shift is required. AJAX is the incarnation of a new paradigm for the next generation of Web applications that is probably destined to last for at least the next decade.

To address the demand for more powerful features and a framework to support them, in January 2007 Microsoft released a separate framework—the ASP.NET AJAX Extensions. When this framework is installed on top of ASP.NET 2.0, developers can add AJAX capabilities to any new and existing Web sites written with ASP.NET. In ASP.NET 3.5, the ASP.NET AJAX Extensions framework has been slightly enhanced and fully incorporated in the main framework. As a result, once you upgrade to ASP.NET 3.5 you have AJAX capabilities out of the box and don't need the services of ASP.NET AJAX Extensions any longer. At the same time,
Part IV ASP.NET AJAX Extensions

ASP.NET 3.5 Web sites and ASP.NET 2.0 Web sites equipped with ASP.NET AJAX Extensions can happily live and run side by side on the same Web server.

In this chapter, I’ll first dig deeper into the motivation for and driving force behind AJAX. Next, I’ll review the AJAX features you find in ASP.NET 3.5, focusing on what appears to be the easiest route to AJAX in ASP.NET systems—partial rendering. In the next chapter, I’ll consider the use of ad hoc server-side services to more evenly distribute the workload between the client and the server.

Note Most books, articles, and documentation tend to use the expressions “Web site” and “Web application” interchangeably. My books and articles are no exception. In the majority of situations, using the two expressions as synonyms is acceptable. It should be noted, though, that strictly speaking “Web site” and “Web application” mean two distinct things. In particular, a Web site is a collection of Web pages and auxiliary resources hosted on a Web server accessible via the Internet or local area network (LAN). A Web application is an application that provides a Web front end and is accessed via the Internet or LAN. The two concepts differ if we consider a Web site made of plain HTML pages and images. But as we move on to consider a dynamic Web site whose pages are generated dynamically on the server, the difference between a Web site and application begins to blur. A server application that reaches its users over the Internet often needs a Web-based presentation layer—that is, one or more Web sites. The distinction becomes important when working with AJAX because we’re no longer simply transferring Web pages but also lower-level data. As defined, Web sites deal only with pages and other resources (such as images), whereas Web applications have the added capability to transfer information other than pure pages or resources. It’s this “other information” we’re interested in when working with AJAX.

The ASP.NET AJAX Infrastructure

From a developer-oriented perspective, AJAX collectively refers to a set of development components, tools, and techniques for creating highly interactive Web applications that give users an overall better experience. According to the AJAX paradigm, Web applications work by exchanging data rather than pages with the Web server. From a user perspective, this means that faster roundtrips occur and, more importantly, page loading is quicker and the need for refreshing the page entirely is significantly reduced.

As a result, a Web application tends to look like a classic desktop Microsoft Windows application and can invoke server code from the client, run and control server-side asynchronous tasks, and feature a strongly responsive and nonflickering user interface. An AJAX application can have a number of features that minimize user frustration, provide timely feedback about what’s going on, and deliver great mashed-up content. (Hold on! This doesn’t mean AJAX Web applications are the same as desktop applications; they simply allow for a few more desktop-like features.)
Note  AJAX applications have a number of plusses but also a few drawbacks. Overall, choosing an AJAX application rather than a classic Web application is simply a matter of weighing the trade-offs. An AJAX application certainly gives users continuous feedback and never appears held up by some remote operation. On the other hand, AJAX applications are not entirely like desktop applications, and their capabilities in terms of graphics, multimedia, and hardware control are not as powerful as in a regular (smart) client. In the end, AJAX applications are just one very special breed of a Web application; as such, they might require some code refactoring to deliver the expected performance and results.

We are all witnessing and contributing to an interesting and unique phenomenon—the Web is undergoing an epochal change right before our eyes as a result of our actions. Ten years ago, the Web was in its infancy and based on an infrastructure that was simple, ubiquitous, and effective. Ten years of Web evolution has resulted in the building of a thick layer of abstraction on the server side, but it hasn’t changed the basic infrastructure—HTTP protocol and pages. The original infrastructure, which was the chief factor for the rapid success of the Web model of applications, is still there.

In fact, the next generation of Web applications will still be based on the HTTP protocol and pages. However, the contents of pages and the capabilities of the server-side and client-side machinery will change to provide a significantly richer user experience.

The Hidden Engine of AJAX

Today Web applications work by submitting user-filled forms to the Web server and displaying the markup returned by the Web server. The client-to-server communication employs the HTTP protocol and is usually conducted by the browser. The new model that AJAX heralds simply employs an alternate and scriptable tool to conduct the HTTP communication with the Web server. The benefit is that the page that triggers the call remains up and running and refreshes its document object model (DOM) with the freshly downloaded data. No page replacement occurs, and the overall user experience is smooth and continual.

The Classic Browser-Led Model

Using the local Domain Name System (DNS) resolver in the operating system, the browser resolves the requested URL to an IP address and opens a socket. An HTTP packet travels over the wire to the given destination. The packet includes the form and all its fields. The request is captured by the Web server and typically forwarded to an internal module for further processing. At the end of the process, an HTTP response packet is prepared and the return value for the browser is inserted in the body. If the response contains an HTML page, the browser replaces the current contents entirely with the new chunk of markup.
While the request is being processed on the server, the “old” page is frozen but still displayed to the client user. As soon as the “new” page is downloaded, the browser clears the display and renders the page.

This model was just fine in the beginning of the Web age when pages contained little more than formatted text, hyperlinks, and some images. The success of the Web has prompted users to ask for increasingly more powerful features, and it has led developers and designers to create more sophisticated services and graphics. The net effect is that pages are heavy and cumbersome—even though we still insist on calling them “rich” pages. Regardless of whether they’re rich or just cumbersome, these are the Web pages of today’s applications. And nobody really believes that we’re going to return to the scanty and spartan HTML pages of a decade ago.

Given the current architecture of Web applications, each user action requires a complete redraw of the page. Subsequently, richer and heavier pages render slowly and, as a result, produce a good deal of flickering. Projected to the whole set of pages in a large, portal-like application, this mechanism is just perfect for unleashing the frustrations of the poor end user.

The New Out-of-Band Model

The chief factor that enables AJAX functionality in a Web page is the ability to issue out-of-band HTTP requests. In this context, an out-of-band call indicates an HTTP request placed using a component different from the browser. The out-of-band call is triggered via script by an HTML page event and is served by a proxy component. In AJAX frameworks, the proxy component is based on the XMLHttpRequest object.

Note About a decade ago, there was a team at Microsoft working on a technology called Remote Scripting (RS). RS never reached the stage of a version 1.0 but had a lot in common with today’s AJAX hidden engine. In RS, the proxy component was a Java applet managing the browser-to-server communication.

XMLHttpRequest is a browser’s object that is scriptable through JavaScript. It sends a regular HTTP request to the specified URL and waits, either synchronously or asynchronously, for it to be fully served. When the response data is ready, the proxy invokes a user-defined JavaScript callback to refresh any portion of the page that needs updating. Figure 19-1 provides a graphical overview of the model.
out-of-band calls are sent through a proxy component, and a JavaScript callback is used to update any portion of the page affected by returned data.

All browsers know how to replace an old page with a new page; until a few years ago, though, not all of them provided an object model to represent the current contents of the page. (Today, I can hardly mention a single modern, commercially available browser that doesn’t expose a read/write page DOM.) For browsers that supply an updatable object model for HTML pages, the JavaScript callback function can refresh specific portions of the old page, thus making them look updated, without a full reload.

Exactly what are the capabilities required of a browser to run AJAX functionalities? As mentioned, a browser needs to provide two key capabilities: a proxy mechanism to make client code able to place out-of-band HTTP calls, and an updatable DOM. And both capabilities must be achieved through standard and globally accepted interfaces.

There’s a World Wide Web Consortium (W3C) ratified standard for the updatable DOM. A W3C standard for the proxy component is currently being developed. It takes the form of the existing XMLHttpRequest object and is devised as an interface exposed by the browser to allow script code to perform HTTP client functionality, such as submitting form data or loading data from a remote Web site. The latest working draft is available at http://www.w3.org/TR/XMLHttpRequest.

In addition, browsers must support JavaScript and, preferably, cascading style sheets (CSS).

The Role of the HTML Document Object Model

The page Document Object Model (DOM) is the specification that defines a platform- and language-neutral interface for accessing and updating the contents, structure, and style of HTML and XML documents. As a recognized standard ratified by the W3C committee, the DOM is now supported by virtually all browsers. The DOM provides a standard set of objects for representing the constituent elements of HTML and XML documents. All together, these objects form a standard interface for accessing and manipulating child elements of HTML pages and, more in general, XML documents.

The DOM application programming interface (API) renders the displayed page as a tree-based structure. For a Web page, each node maps to an object that represents an HTML tag.
The object, therefore, has properties and methods that can be applied to an HTML tag. There are three fundamental operations you can accomplish on a node: find the node (including related nodes such as children, parent, or sibling nodes), create a node, and manipulate a node. Identifying a particular node is easy as long as the page author knows the ID of the corresponding element.

The W3C DOM consists of three levels that indicate, for the browser, three different levels of adherence to the standard. For more information, take a look at [http://www.w3.org/DOM](http://www.w3.org/DOM).

**From Dynamic HTML to the Standard DOM**

About ten years ago, with Internet Explorer 4.0, Microsoft introduced a proprietary object model named Dynamic HTML (DHTML) to enable page authors to update the current page dynamically using JavaScript. The success of DHTML led to the definition of a standard document object model—the W3C’s DOM. Quite obviously, the DOM evolved from DHTML and became much more generalized than DHTML.

Today most browsers support a mix of DOM and DHTML. Which one should you use? In particular, to update some contents, should you obtain a reference to the textual child node of the node that matches the intended HTML tag (the DOM way) or just grab a reference to a node and use the `innerHTML` property as you would in the DHTML way? Likewise, to add a new element, should you create a new element or just stuff in a chunk of updated HTML via `innerHTML`? Admittedly, one of the most interesting debates in the community is whether to use DHTML to manipulate pages or opt for the cleaner approach propounded by the DOM API.

The key fact is that the DOM API is significantly slower than using `innerHTML`. If you go through the DOM to generate some user interface dynamically, you have to create every element, append each into the proper container, and then set properties. The alternative only entails that you define the HTML you want and render it into the page using `innerHTML`. The browser, then, does the rest by rendering your markup into direct graphics.

Overall, DHTML and DOM manipulation are both useful depending on the context. There are many Web sites that discuss performance tests, and DHTML is always the winner. Anyway, DOM is still perfectly fast as long as you use it the right way—that is, create HTML fragments and append them to the proper container only as the final step.

---

**Note** The inclusion of a standard API to edit the currently displayed page is a key factor in the success of AJAX. Although the first working frameworks for AJAX-like remote scripting date back to a decade ago, the limited support that browsers have had for dynamic document changes through most of this time period slowed down the adoption of such technologies in the industry. Until now. With the HTML DOM becoming a widely supported standard and the availability of a mature tool for remote calls such as XMLHttpRequest, AJAX is ready for prime time.
The XMLHttpRequest Object

Created by Microsoft and adopted soon thereafter by Mozilla, the XMLHttpRequest object is fully supported these days by the majority of Web browsers. The implementation can significantly differ from one browser to the next, even though the top-level interface is nearly identical. For this reason, a W3C committee is at work with the goal of precisely documenting a minimum set of interoperable features based on existing implementations. An excellent presentation on the component can be found here: http://developer.mozilla.org/en/docs/XMLHttpRequest.

Note The XMLHttpRequest object originally shipped as a separate component with Internet Explorer 5.0 back in the spring of 1999. It is a native component of all Microsoft operating systems that have shipped since. In particular, you’ll certainly find it installed on all machines that run Windows 2000, Windows XP, and newer operating systems.

When the XMLHttpRequest object was first released, the Component Object Model (COM) was ruling the world at Microsoft. The extensibility model of products and applications was based on COM and implemented through COM components. In the late 1990s, the right and natural choice was to implement this new component as a reusable automation COM object, named Microsoft.XmlHttp.

COM objects are external components that require explicit permission to run inside of a Web browser. In particular, to run the XMLHttpRequest object, and subsequently enable any AJAX functionality built on top of it, at a minimum a client machine needs to accept ActiveX components marked safe for scripting. (See Figure 19-2.)

The XMLHttpRequest object is certainly a safe component, but to enable it users need to decrease their security settings and accept any other component “declared” safe for scripting that is around the Web sites they visit.

Mozilla adopted XMLHttpRequest immediately after its first release with Internet Explorer 5.0. However, in Mozilla-equipped browsers, the XMLHttpRequest object is part of the browser’s object model and doesn’t rely on external components. Put another way, a Mozilla browser such as Firefox publishes its own XMLHttpRequest object into the scripting engine and never uses the COM component, even when the COM component is installed on the client machine and is part of the operating system.
Part IV ASP.NET AJAX Extensions

FIGURE 19-2 The property window used to change the security settings in Internet Explorer

As a result, in Mozilla browsers, XMLHttpRequest looks like a native JavaScript object and can be instantiated through the classic new operator:

```javascript
// The object name requires XML in capital letters
var proxy = new XMLHttpRequest();
```

When the browser is Internet Explorer (up to version 6.0), the XMLHttpRequest object is instantiated using the ActiveXObject wrapper, as shown here:

```javascript
var proxy = new ActiveXObject("Microsoft.XMLHttp");
```

Generally, AJAX-style frameworks check the current browser and then decide which route to take.

Implemented as a COM component for historical reasons on Internet Explorer browsers, the XMLHttpRequest object has finally become a browser object with Internet Explorer 7.0. All potential security concerns are removed at the root. Needless to say, implemented as a browser object, the XMLHttpRequest functionality is somewhat safer, at least in the sense it doesn’t require users to change their security settings for the browser.

Using the XMLHttpRequest Object

The XMLHttpRequest object is designed to perform one key operation: sending an HTTP request. The request can be sent either synchronously or asynchronously. The following bit of
code shows the programming interface of the object as it results from the W3C working draft at the time of this writing:

```javascript
interface XMLHttpRequest {
    function onreadystatechange;
    readonly unsigned short readyState;
    void open(string method, string url);
    void open(string method, string url, bool async);
    void open(string method, string url, bool async, string user);
    void open(string method, string url, bool async, string user, string pswd);
    void setRequestHeader(string header, string value);
    void send(string data);
    void send(Document data);
    void abort();
    string getAllResponseHeaders();
    string getResponseHeader(string header);
    string responseText;
    Document responseXML;
    unsigned short status;
    string statusText;
};
```

Using the component is a two-step operation. First, you open a channel to the URL and specify the method (GET, POST, or other) to use and whether you want the request to execute asynchronously. Next, you set any required header and send the request. If the request is a POST, you pass to the `send` method the body of the request.

The `send` method returns immediately in the case of an asynchronous operation. You write an `onreadystatechange` function to check the status of the current operation and, using that function, figure out when it is done. The following code shows how to carry on a POST request using the `XMLHttpRequest` object:

```javascript
var xmlRequest, e;
try {
    xmlRequest = new XMLHttpRequest();
} catch(e) {
}
```

Using the component is a two-step operation. First, you open a channel to the URL and specify the method (GET, POST, or other) to use and whether you want the request to execute asynchronously. Next, you set any required header and send the request. If the request is a POST, you pass to the `send` method the body of the request.

The `send` method returns immediately in the case of an asynchronous operation. You write an `onreadystatechange` function to check the status of the current operation and, using that function, figure out when it is done. The following code shows how to carry on a POST request using the `XMLHttpRequest` object:

```javascript
var xmlRequest, e;
try {
    xmlRequest = new XMLHttpRequest();
} catch(e) {
    try {
        xmlRequest = new ActiveXObject("Microsoft.XMLHTTP");
    } catch(e) {
    }
}
```
// Prepare for a synchronous POST request
var body = null;  // An empty request body this time...
xmlRequest.open("POST", pageUrl, false);
xmlRequest.setRequestHeader("Content-Type",  
  "application/x-www-form-urlencoded");
xmlRequest.send(body);

In a synchronous call, the send method returns when the response has been fully downloaded and parsed by the object. You can access it as a plain string using the responseText property. If the response is an XML stream, you can have it exposed as an XML DOM object using the responseXML property.

Important If you’re going to use ASP.NET AJAX or any other AJAX-like framework for building your applications, you’ll hardly hear anything about the XMLHttpRequest object, much less use it directly in your own code. An AJAX framework completely encapsulates this object and shields page authors and application designers from it. You don’t need to know about XMLHttpRequest to write great AJAX applications, no matter how complex and sophisticated they are. However, knowing the fundamentals of XMLHttpRequest can lead you to a better and more thorough understanding of the platform and to more effective diagnoses of problems.

The Microsoft AJAX JavaScript Library

Most of the power of AJAX resides on the client and is strictly related to the browser’s and platform’s client-side functionality. No AJAX capability would ever be possible without a client-side engine; and this engine can only be written in JavaScript. Such a script code governs the execution of out-of-band calls and often kicks in and replaces regular postbacks with AJAX postbacks. Moreover, no AJAX functionality would ever be possible without JavaScript and a standard (and rich) DOM. The DOM, though, is not enough.

The DOM represents the programming gateway to the page constituent elements, but it is not designed to provide programming facilities such as those you can find in a general-purpose library. Normally, the script tools you can leverage to consume objects and contents from the DOM are those provided by the JavaScript language. Not exactly a powerful toolkit. Enter the Microsoft AJAX JavaScript library.

The AJAX extensions to ASP.NET silently leverage the Microsoft AJAX JavaScript library for all of its built-in features. The library, though, is also available to page authors to code their own JavaScript page-specific functions.

The Microsoft AJAX library is written in JavaScript, although with a strong sense of object-orientation. The JavaScript language does support objects and allow the creation of custom objects. It does not, however, support full object-orientedness because it has no native concept of true object inheritance. Nonetheless, even excluding true object orientation, JavaScript is still a modern and suitable language that can be used to build a class framework...
Chapter 19 Partial Rendering: The Easy Way to AJAX

à la the .NET Framework. ASP.NET AJAX takes the JavaScript language to the next level by adding some type-system extensions and the notions of namespace and inheritance. In addition, the ASP.NET AJAX JavaScript supports interfaces and enumerations, and it has a number of helper functions to manipulate strings and arrays.

These extensions are coded using the base set of instructions that characterize the core JavaScript language, and they’re persisted to a set of .js files. These .js files are not installed as distinct files on the Web server when you install ASP.NET. They are embedded as resources into the ASP.NET AJAX assembly—system.web.extensions. If you want them available as distinct files (for example, for your home perusal), go to http://msdn2.microsoft.com/en-us/asp.net/bb944808.aspx, check the license agreement, and get them as a single downloaded compressed file.

Let’s now dig out some of these extensions to the JavaScript language and briefly explore the features of the built-in classes. For more information, you can have a look at Chapter 2 of my Introducing ASP.NET AJAX books from Microsoft Press.

**Important** The Microsoft AJAX JavaScript library is self-contained in the .js files you get from the aforementioned URL. This means that you can embed these files in any Web page and enjoy the object-oriented features of JavaScript regardless of whether or not ASP.NET AJAX is being used to power the page. For example, you can use the <script> tag to include all required JavaScript files in a PHP or classic ASP page and enjoy the advanced capabilities of the Microsoft AJAX JavaScript library.

**JavaScript Language Extensions**

The JavaScript language features a set of built-in objects, including Function, Object, Boolean, Array, Number, and String. All intrinsic objects have a read-only property named prototype. You use the prototype property to provide a base set of functionality shared by any new instance of an object of that class. New functionality can be added to the class prototype inside of an application to extend and improve the capabilities of a given class. This is exactly what the Microsoft AJAX library does.

Type information is the most important aspect of the JavaScript language that has been enhanced. Aside from instances of the base types, everything else in JavaScript is a plain object. With the library extensions, you have a type information system that is similar to the .NET Framework. The following code now works and displays “Person” instead of a generic “object” string:

```javascript
var p = new Person("Dino", "Esposito");
alert(Object.getTypeName(p));
```

But where does the getTypeName method come from? It is a new method defined on the native JavaScript Object type. The Microsoft AJAX library contains code that defines new
objects and extends existing JavaScript objects with additional functionality. Table 19-1 lists the main global objects defined in the library.

**TABLE 19-1 Top-Level Objects in the Microsoft AJAX Library**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>Extends the native <code>Array</code> object. This object groups static methods to add, insert, remove, and clear elements of an array. It also includes static methods to enumerate elements and check whether a given element is contained in the array.</td>
</tr>
<tr>
<td>Boolean</td>
<td>Extends the native <code>Boolean</code> object. This object defines a static parse method to infer a Boolean value from a string or any expression that evaluates to a Boolean value.</td>
</tr>
<tr>
<td>Date</td>
<td>Extends the native <code>Date</code> object with a couple of instance methods: <code>localeFormat</code> and <code>format</code>. These methods format the date using the locale or invariant culture information.</td>
</tr>
<tr>
<td>Error</td>
<td>Defines a static create method to wrap the JavaScript <code>Error</code> object and add a richer constructor to it. This object incorporates a couple of properties—<code>message</code> and <code>name</code>—to provide a description of the error that occurred and identify the error by name. A number of built-in error objects are used to simulate exceptions. In this case, the <code>name</code> property indicates the name of the exception caught.</td>
</tr>
<tr>
<td>Function</td>
<td>Extends the native <code>Function</code> object. This object groups methods to define classes, namespaces, delegates, and a bunch of other object-oriented facilities.</td>
</tr>
<tr>
<td>Number</td>
<td>Extends the native <code>Number</code> object. This object defines a static parse method to infer a numeric value from a string or any expression that evaluates to a numeric value. In addition, it supports a pair of static formatting methods: <code>localeFormat</code> and <code>format</code>.</td>
</tr>
<tr>
<td>Object</td>
<td>Extends the native <code>Object</code> object. This object groups methods to read type information, such as the type of the object being used.</td>
</tr>
<tr>
<td>RegExp</td>
<td>Wraps the native <code>RegExp</code> object.</td>
</tr>
<tr>
<td>String</td>
<td>Extends the native <code>String</code> object. This object groups string manipulation methods, such as <code>trim</code> and <code>endsWith</code> and <code>startsWith</code> methods. In addition, it defines static <code>localeFormat</code> and <code>format</code> methods that are close relatives of the <code>String.Format</code> method of the managed <code>String</code> type.</td>
</tr>
</tbody>
</table>

After the Microsoft AJAX library has been added to the application, the following code will work just fine:

```javascript
var s = "Dino";
alert(s.startsWith('D'));
```

The native JavaScript `String` object doesn't feature either a `startsWith` or an `endsWith` method; the extended AJAX `String` object does.
One of the most common mistakes made when writing script code inside of Web pages is to use direct access to HTML elements instead of resorting to the `getElementById` method of the DOM. Suppose you have a text box element named `TextBox1` in your client page. The following script code won’t work on all browsers:

```javascript
alert(TextBox1.value);
```

The correct form ratified by the W3C paper for the HTML DOM standards is shown here:

```javascript
alert(document.getElementById("TextBox1").value);
```

The correct form is clearly more verbose and bothersome to write over and over again. The Microsoft AJAX library comes to the rescue with the `$get` global function. Simply put, the `$get` function is a shortcut for the `document.getElementById` function. If the Microsoft AJAX library is in use, the following expression is fully equivalent to the one just shown:

```javascript
alert($get("TextBox1").value);
```

The `$get` function has two overloads. If you call `$get` passing the sole ID, the function falls back into `document.getElementById`. Alternatively, you can specify a container as the second argument, as shown here:

```javascript
var parent = $get("Div1");
$get("TextBox1", parent);
```

If the container element supports the `getElementById` method, the function returns the output of `element.getElementById`; otherwise, the `$get` function uses the DOM interface to explore the contents of the subtree rooted in the element to locate any node with the given ID.

### Object-Oriented Extensions: Namespaces

In JavaScript, the `Function` object is the main tool you use to combine code with properties and forge new components. In the Microsoft AJAX library, the `Function` object is extended to incorporate type information, as well as namespaces, inheritance, interfaces, and enumerations.

A namespace provides a way of grouping and classifying types belonging to a library. A namespace is not a type itself, but it adds more information to the definition of each type it contains to better qualify the type. By default, all custom JavaScript functions belong to the global space of names. In the Microsoft AJAX library, you can associate a custom function
Part IV   ASP.NET AJAX Extensions

with a particular namespace, for purely organizational reasons. When declaring a custom
type in the Microsoft AJAX library, you can do as follows:

    Type.registerNamespace("Core35");
    Core35.Person = function Core35Person(firstName, lastName)
    {
        this._firstName = firstName;
        this._lastName = lastName;
    }

    // Define the body of members
    function Core35$Person$ToString() {
        return this._lastName + ", " + this._firstName;
    }

    // Define the prototype of the class
    Core35.Person.prototype = {
        ToString:      Core35$Person$ToString,
        get_FirstName: Core35$Person$get_FirstName,
        set_FirstName: Core35$Person$set_FirstName,
        get_LastName:  Core35$Person$get_LastName,
        set_LastName:  Core35$Person$set_LastName
    }

    // Register the class, extending our own IntroAjax Person class
    IntroAjax.Person.registerClass("Core35.Person");

The *Type.registerNamespace* method adds the specified namespace to the runtime environ-
ment. In a way, the *registerNamespace* method is equivalent to using the *namespace {…}*
construct in C# or the *Namespace .. End Namespace* construct in Microsoft Visual Basic.

The *Core35.Person* function defined following the namespace declaration describes a
*Person* type in the *Core35* namespace. Finally, the newly defined function must be registered
as a class with the Microsoft AJAX library framework. You use the *registerClass* method on
the current function. The *registerClass* method is defined in the prototype of the *Function*
object, as such, it is inherited by all functions. Internally, the *registerClass* method sets the
*_typeName* property of the function to the first parameter of the method—the actual name
of the class.

The *registerClass* method takes a number of parameters. The first parameter is mandatory,
and it indicates the public name that will be used to expose the JavaScript function as a class.
Additional and optional parameters are the parent class, if there is any, and any interface
implemented by the class.

Any other use
requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.
Object-Oriented Extensions: Inheritance

Let's define a new class, Citizen, that extends Person by adding a couple of properties: an address and a national identification number. Here's the skeleton of the code you need:

```javascript
// Declare the class
Core35.Citizen = function Core35$Citizen(firstName, lastName, id)
{
...
}

// Define the prototype of the class
Core35.Citizen.prototype = {
...
}

// Register the class
Core35.Citizen.registerClass("Core35.Citizen", Core35.Person);
```

Note that the first argument of registerClass is a string, but the second one has to be an object reference. Let's flesh out this code a bit.

In the constructor, you’ll set some private members and call the base constructor to initialize the members defined on the base class. The initializeBase method (defined on Function) retrieves and invokes the base constructor:

```javascript
Core35.Citizen = function Core35$Citizen(firstName, lastName, id)
{
Core35.Citizen.initializeBase(this, [firstName, lastName]);
this._id = id;
this._address = "";
}
```

You pass initializeBase the reference to the current object as well as an array with any parameters that the constructor to call requires. You can use the […] notation to define an array inline. If you omit the […] notation, be ready to handle a parameter count exception.

Quite often, developers derive a class because they need to add new members or alter the behavior of an existing method or property. Object-oriented languages define a proper keyword to flag members as overridable. How is that possible in JavaScript? By simply adding a member to the class prototype, you mark it as overridable in derived classes. In addition, if the member already exists on the base class, it is silently overridden in the new one. Here’s the prototype of the Citizen class:

```javascript
Core35.Citizen.prototype =
{
ToString:    Core35$Citizen$ToString,
get_ID:      Core35$Citizen$get_ID,
get_Address: Core35$Citizen$get_Address,
set_Address: Core35$Citizen$set_Address
}
```
Part IV  ASP.NET AJAX Extensions

The class has a read-only ID property and a read-write Address property. Furthermore, it overrides the ToString method defined in the parent class:

```javascript
function Core35$Citizen$ToString()
{
    var temp = Core35.Citizen.callBaseMethod(this, 'ToString');
    temp += "  \[" + this._id + "]";
    return temp;
}
```

You use callBaseMethod to invoke the same method on the parent class. Defined on the Function class, the callBaseMethod method takes up to three parameters: the instance, the name of the method, plus an optional array of arguments for the base method.

As mentioned earlier, the ToString method on the Person class returns a LastName, FirstName string. The ToString method on the Citizen class returns a string in the following format: LastName, FirstName [ID].

Object-Oriented Extensions: Interfaces

Finally, an interface describes a group of related behaviors that are typical of a variety of classes. In general, an interface can include methods, properties, and events; in JavaScript, it contains only methods.

Keeping in mind the constraints of the JavaScript language, to define an interface you create a regular class with a constructor and a prototype. The constructor and each prototyped method, though, will just throw a not-implemented exception. Here’s the code for the sample Sys.IDisposable built-in interface:

```javascript
Type.registerNamespace("Sys");
Sys.IDisposable = function Sys$IDisposable()
{
    throw Error.notImplemented();
};
function Sys$IDisposable$dispose()
{
    throw Error.notImplemented();
}
Sys.IDisposable.prototype =
{
    dispose: Sys$IDisposable$dispose
}
Sys.IDisposable.registerInterface('Sys.IDisposable');
```

The following statement registers the Citizen class, makes it derive from Person, and implements the IDisposable interface:

```javascript
Core35.Citizen.registerClass('Core35.Citizen',
           Core35.Person, Sys.IDisposable);
```
To implement a given interface, a JavaScript class simply provides all methods in the interface and lists the interface while registering the class:

```javascript
function Core35$Citizen$dispose {
    this._id = "";
    this._address = "";
}
Core35.Citizen.prototype = {
    dispose: Core35$Citizen$dispose
    ...
}
```

Note, though, that you won’t receive any runtime error if the class that declares to implement a given interface doesn’t really support all the methods.

If a class implements multiple interfaces, you simply list all required interfaces in the `registerClass` method as additional parameters. Here’s an example:

```javascript
Sys.Component.registerClass('Sys.Component', null,
    Sys.IDisposable,
    Sys.INotifyPropertyChange,
    Sys.INotifyDisposing);
```

As you can see, in this case you don’t have to group interfaces in an array.

The Application Core Component

The AJAX client library is made up of three main logical layers: JavaScript extensions, core framework classes, and user-interface (UI) framework classes. (See Figure 19-3.)

![Figure 19-3: A graphical view of the Microsoft AJAX JavaScript library](image-url)
As mentioned, JavaScript extensions add new methods and capabilities to native JavaScript objects and enable registration methods to simulate object-oriented constructs such as classes, namespaces, inheritance, and interfaces. The UI framework includes base components to define client behaviors, controls, DOM elements, and input devices such as keyboard and mouse buttons. The core framework classes form a sort of base library that incorporates a set of commonly used classes for event handling, string manipulation, Web services, debugging, and network operations.

The execution of each ASP.NET AJAX page is controlled by an application object that is instantiated in the body of the library. The application object is an instance of a private class—the `Sys._Application` class. Whenever an ASP.NET AJAX page is loaded in the browser, an instance of the `Sys._Application` class is promptly created and assigned to the `Sys.Application` object:

```javascript
Sys.Application = new Sys._Application();
```

In addition, each ASP.NET AJAX page is injected with the following script code:

```javascript
<script type="text/javascript">
   <!--
   Sys.Application.initialize();
   // -->
</script>
```

This code is placed immediately after the closing tag of the page's form, and it commands the loading of any script files registered for loading with the page's script manager. As a result, the `Sys.Application` object is the nerve center of the ASP.NET AJAX page.

**Note** JavaScript has no notion of private members; therefore, private members are conventionally indicated by the underscore symbol (_) prefixing their names.

The `Sys.Application` object serves one main purpose: providing access to page components. Its `findComponent` method scrolls the runtime hierarchy of Microsoft AJAX components for the current page until it finds a component with a matching ID. The method has two possible prototypes:

```javascript
Sys._Application.findComponent(id);
Sys._Application.findComponent(id, parent);
```

The former overload takes the ID of the component, uses it to look up the component, and then navigates the hierarchy all the way down from the root. When a non-null `parent` argument is specified, the search is restricted to the subtree rooted in the context object. The `id` parameter must be a string; the `parent` parameter must be a Microsoft AJAX library object. The method returns the object that matches the ID, or it returns null if no such object is found.
The Microsoft AJAX library also supports a shortcut for retrieving runtime components—the $find method. The $find method is an alias for findComponent:

```javascript
var $find = Sys.Application.findComponent;
```

You can use this method to locate all components created by server controls and extenders, as well as by your own JavaScript code. You can't use $find to locate DOM elements; for DOM elements, you must resort to $get.

When a page using the Microsoft AJAX library page first loads up, the load event is fired for the client code to perform any required initialization. Note that the event refers to the page lifetime, not the application lifetime. So whenever a classic postback occurs, you receive a new load event. You don't receive such events for any AJAX-style postback conducted via XMLHttpRequest. Likewise, the unload event is fired when the page is unloaded.

The load event occurs after a page has been loaded and initialized completely. For such a page, the load event is preferable to the browser's onload for initialization purposes. Only when you get the Microsoft AJAX library load event, therefore, can you be sure that the page is ready for user interaction. The unload event occurs just before the Microsoft AJAX library runtime releases the page and all of its resources. For the sake of the application's stability, you should use this event instead of the browser's onunload event for clean-up tasks.

The easiest way to define load and unload handlers is by means of predefined function names: pageLoad and pageUnload. These functions need to be global and parameterless:

```javascript
<script type="text/JavaScript" language="JavaScript">
function pageLoad()
{
    alert("Being loaded");
}
function pageUnload()
{
    alert("Being unloaded");
}
</script>
```

Because this piece doesn't directly call into any of the Microsoft AJAX library objects—including Sys.Application—you can safely place it everywhere, even at the top of the page.

### The Network Stack

AJAX libraries in general, and ASP.NET AJAX Extensions in particular, owe their growing popularity to their ability to execute out-of-band Web requests from the client. In particular, ASP.NET AJAX extensions allow you to invoke Web service methods as well as static methods defined on the code-behind page class. This ability leverages the networking support built into the Microsoft AJAX library.
In the Microsoft AJAX library, a remote request is represented by an instance of the `Sys.Net.WebRequest` class. Table 19-2 lists the properties of the class.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>body</td>
<td>Gets and sets the body of the request</td>
</tr>
<tr>
<td>executor</td>
<td>Gets and sets the Microsoft AJAX library object that will take care of executing the request</td>
</tr>
<tr>
<td>headers</td>
<td>Gets the headers of the request</td>
</tr>
<tr>
<td>httpVerb</td>
<td>Gets and sets the HTTP verb for the request</td>
</tr>
<tr>
<td>timeout</td>
<td>Gets and sets the timeout, if any, for the request</td>
</tr>
<tr>
<td>url</td>
<td>Gets and sets the URL of the request</td>
</tr>
</tbody>
</table>

The `WebRequest` class defines the `url` property to get and set the target URL and the `headers` property to add header strings to the request. If the request is going to be a POST, you set the body of the request through the `body` property. A request executes through the method `invoke`. The `completed` event informs you about the completion of the request.

Each Web request is executed through an internal class—the Web request manager—that employs an "executor" to open the socket and send the packet. All executors derive from a common base class—the `Sys.Net.WebRequestExecutor` class.

The Microsoft AJAX library defines just one HTTP executor—the `Sys.Net.XMLHttpExecutor` class. As the name suggests, this executor uses the popular `XMLHttpRequest` object to execute the HTTP request.

**Note** All AJAX libraries are associated with the `XMLHttpRequest` browser object. So what else could an executor be other than a reference to the `XMLHttpRequest` browser object? In general, an HTTP executor is any means you can use to carry out a Web request. An alternative executor might be based on HTTP frames. The idea is to use a dynamically created inline frame to download the response of a given request and then parse that result into usable objects.

**DOM Events**

Building cross-browser compatibility for events is not an easy task. Internet Explorer has its own eventing model, and so do Firefox and Safari. For this reason, the event model of the Microsoft AJAX library is a new abstract API that joins together the standard W3C API and the Internet Explorer nonextensible model. The new API is closely modeled after the standard W3C API.

In addition to using different method names (`addEventListener` is for Firefox, and `attach/detachEvent` is for Internet Explorer), browsers differ in the way they pass event data...
down to handlers. In Internet Explorer, an event handler receives its data through the global `window.event` object; in Firefox, the event data is passed as an argument to the handler. In the Microsoft AJAX library, event handlers receive a parameter with proper event data.

Another significant difference is in the way mouse and keyboard events are represented. The Microsoft AJAX library abstracts away any differences between browsers by providing ad hoc enumerated types, such as `Sys.UI.Key` and `Sys.UI.MouseButton`. Here's some sample code:

```javascript
function button1_Click(e)
{  
    if (e.button === Sys.UI.MouseButton.leftButton)
        ...
}

function keyboard_EnterPressed(e)
{  
    if (e.keyCode === Sys.UI.Key.enter)
        ...
}
```

The Microsoft AJAX library provides a shorthand notation to create DOM event hookups and removal. For example, you can use the `$addHandler` and `$removeHandler` aliases to add and remove a handler. Here's the syntax:

```javascript
$addHandler(element, "eventName", handler);
$removeHandler(element, "eventName", handler);
```

In many cases, you'll want to hook up several handlers to a DOM event for a component. Rather than manually creating all the required delegates and related handlers, you can use a condensed syntax to add and remove multiple handlers:

```javascript
initialize: function()
{  
    var elem = this.get_element();
    $addHandlers(  
        elem,
        [
            'mouseover': this._mouseHoverHandler,
            'mouseout': this._mouseOutHandler,
            'focus': this._focusHandler,
            'blur': this._blurHandler
        ],
        this);
}
```

The `$clearHandlers` alias, conversely, removes all handlers set for a particular DOM element in a single shot.
If you write a component and wire up some events, it is essential that you clear all handlers when the component is unloaded, or even earlier, if you don’t need the handler any longer. For example, you should do that from the component’s `dispose` method to break circular references between your JavaScript objects and the DOM. Correctly applied, this trick easily prevents nasty memory leaks.

**Note** You won’t receive any event data if you bind the handler via markup—for example, by setting the `onclick` attribute of an `<input>` tag. Everything said here applies only to event handlers added via methods (and aliases) of the `Sys.UI.DomEvent` class. Events bound through attributes are still processed, but you have to resort to your knowledge of the browser’s event model to correctly grab associated information.

### Other Facilities

The Microsoft AJAX library contains a number of other miscellaneous components to provide additional facilities to ASP.NET AJAX developers.

The `Sys.StringBuilder` class adds advanced text manipulation capabilities to ASP.NET AJAX pages. As the name suggests, the class mimics the behavior of the managed `StringBuilder` class defined in the .NET Framework. When you create an instance of the builder object, you specify initial text. The builder caches the text in an internal array by using an element for each added text or line. The `Sys.StringBuilder` object doesn’t accept objects other than non-null strings. The `toString` method composes the text by using the `join` method of the JavaScript array class.

The Microsoft AJAX library `String` class is also enriched with a format method that mimics the behavior of the `Format` method on the .NET Framework `String` class:

```
alert(String.format("Today is: [0]", new Date()));
```

You define placeholders in the format string using `{n}` elements. The real value for placeholders is determined by looking at the `n`th argument in the format method call.

Another class that is worth mentioning is the `Sys._Debug` class. An instance of this internal class is assigned to the `Sys.Debug` global object:

```
Sys.Debug = new Sys._Debug();
```

In your pages, you use the `Sys.Debug` object to assert conditions, break into the debugger, or trace text. For example, the `traceDump` method writes the contents of the specified object in a human-readable format in the Microsoft AJAX library trace area. The trace area is expected to be a `<textarea>` element with an ID of `traceConsole`. You can place this element anywhere in the page:

```
<textarea id="traceConsole" cols="40" rows="10" />
```
The `traceDump` method accepts two parameters, as shown here:

```
Sys.Debug.traceDump(object, name)
```

The `name` parameter indicates descriptive text to display as the heading of the object dump. The text can contain HTML markup. Figure 19-4 shows the results.

![The Microsoft AJAX library debugging tracer in action](image)

**FIGURE 19-4** The Microsoft AJAX library debugging tracer in action

### The Script Manager Control

The main control in the server infrastructure of ASP.NET AJAX is the `ScriptManager` control and its twin, the `ScriptManagerProxy` control. You will find just one instance of the `ScriptManager` control in each ASP.NET AJAX page. No AJAX capabilities can be enabled in ASP.NET pages that lack a reference to one `ScriptManager` control. The `ScriptManagerProxy` control is used only in master pages scenarios to reference the original script manager from content pages. (See Chapter 6 for more information about master pages.)

The `ScriptManager` control manages and delivers script resources, thus enabling client scripts to make use of the JavaScript type system extensions and other JavaScript features that we covered earlier in this chapter. The `ScriptManager` control also enables features such as partial-page rendering and Web service and page method calls. The following code shows the simplest and most common way to insert the script manager in an ASP.NET page:

```
<asp:ScriptManager runat="server" ID="ScriptManager1" />
```

The control produces no user interface, works exclusively on the server, and doesn’t add any extra bytes to the page download.
### Properties of the `ScriptManager` Control

The `ScriptManager` control features a number of properties for you to configure its expected behavior. Table 19-3 details the supported properties.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllowCustomErrorsRedirect</td>
<td>Indicates whether custom error redirects will occur during an asynchronous postback. The property is set to <code>true</code> by default.</td>
</tr>
<tr>
<td>AsyncPostBackErrorMessage</td>
<td>Gets and sets the error message to be sent to the client when an unhandled exception occurs on the server during an asynchronous postback. If this property is not set, the native exception's message will be used.</td>
</tr>
<tr>
<td>AsyncPostBackSourceElementID</td>
<td>Gets the ID of the server control that triggered the asynchronous postback. If there's no ongoing asynchronous postback, the property is set to the empty string.</td>
</tr>
<tr>
<td>AsyncPostBackTimeout</td>
<td>Gets and sets the timeout period in seconds for asynchronous postbacks. A value of zero indicates no timeout. The property is set to 90 by default.</td>
</tr>
<tr>
<td>AuthenticationService</td>
<td>Gets an object through which you can set preferences for the client-side authentication service.</td>
</tr>
<tr>
<td>EnablePageMethods</td>
<td>Indicates whether static page methods on an ASP.NET page can be called from client script. The property is set to <code>false</code> by default.</td>
</tr>
<tr>
<td>EnablePartialRendering</td>
<td>Indicates whether partial rendering is enabled for the page. The property is set to <code>true</code> by default.</td>
</tr>
<tr>
<td>EnableScriptGlobalization</td>
<td>Indicates whether the <code>ScriptManager</code> control renders script in the client that supports parsing and formatting of culture-specific information. The property is set to <code>false</code> by default.</td>
</tr>
<tr>
<td>EnableScriptLocalization</td>
<td>Indicates whether the <code>ScriptManager</code> control retrieves script files for the current culture, if they exist. The property is set to <code>false</code> by default.</td>
</tr>
<tr>
<td>IsDebuggingEnabled</td>
<td>Indicates whether the debug versions of client script libraries will be rendered. The <code>debug</code> attribute on the <code>@Page</code> directive doesn't affect this property.</td>
</tr>
<tr>
<td>IsInAsyncPostBack</td>
<td>Indicates whether the current page request is due to an asynchronous postback.</td>
</tr>
<tr>
<td>LoadScriptsBeforeUI</td>
<td>Indicates whether scripts are loaded before or after markup for the page UI is loaded.</td>
</tr>
<tr>
<td>ProfileService</td>
<td>Gets an object through which you can set preferences for the client-side profile service.</td>
</tr>
</tbody>
</table>
Chapter 19: Partial Rendering: The Easy Way to AJAX

Property Description

RoleService
- Gets an object through which you can set preferences for the client-side role service. This property is not available in ASP.NET AJAX Extensions for ASP.NET 2.0.

ScriptMode
- Gets and sets the type (debug or retail) of scripts to load when more than one type is available. Possible values come from the ScriptMode enumeration type: Auto, Inherit, Debug, or Release. The default value is Auto, meaning that the type of script is determined on the fly.

ScriptPath
- Indicates that scripts should be loaded from this path instead of from assembly Web resources.

Scripts
- Gets a collection of script references that the ScriptManager control should include in the page.

Services
- Gets a collection of service references that the ScriptManager control should include in the page.

SupportsPartialRendering
- Indicates whether a particular browser or browser version can support partial page rendering. If this property is set to false, regardless of the value of the EnablePartialRendering property, no partial rendering will be supported on the page. The property is set to true by default.

The script manager is the nerve center of any ASP.NET AJAX pages and does all the work that is necessary to make AJAX features function as expected. Enabling AJAX features mostly means injecting the right piece of script in the right place. The script manager saves ASP.NET developers from dirtiring their hands with JavaScript.

Methods of the ScriptManager Control

Table 19-4 lists the methods defined on the ScriptManager control.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetCurrent</td>
<td>Static method, returns the instance of the ScriptManager control active on the current page.</td>
</tr>
<tr>
<td>GetRegisteredArrayDeclaration</td>
<td>Static method, returns a read-only collection of ECMAScript array declarations that were previously registered with the page. This method is not available in ASP.NET AJAX Extensions for ASP.NET 2.0.</td>
</tr>
<tr>
<td>GetRegisteredClientScriptBlocks</td>
<td>Static method, returns a read-only collection of client script blocks that were previously registered with the ScriptManager control. This method is not available in ASP.NET AJAX Extensions for ASP.NET 2.0.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>GetRegisteredDisposeScripts</td>
<td>Static method, returns a read-only collection of dispose scripts that were previously registered with the page. This method is not available in ASP.NET AJAX Extensions for ASP.NET 2.0.</td>
</tr>
<tr>
<td>GetRegisteredExpandoAttributes</td>
<td>Static method, returns a read-only collection of custom (expando) attributes that were previously registered with the page. This method is not available in ASP.NET AJAX Extensions for ASP.NET 2.0.</td>
</tr>
<tr>
<td>GetRegisteredHiddenFields</td>
<td>Static method, returns a read-only collection of hidden fields that were previously registered with the page. This method is not available in ASP.NET AJAX Extensions for ASP.NET 2.0.</td>
</tr>
<tr>
<td>GetRegisteredOnSubmitStatements</td>
<td>Static method, returns a read-only collection of onsubmit statements that were previously registered with the page. This method is not available in ASP.NET AJAX Extensions for ASP.NET 2.0.</td>
</tr>
<tr>
<td>GetRegisteredStartupScripts</td>
<td>Static method, returns a read-only collection of startup scripts that were previously registered with the page. This method is not available in ASP.NET AJAX Extensions for ASP.NET 2.0.</td>
</tr>
<tr>
<td>RegisterArrayDeclaration</td>
<td>Static method, ensures that an ECMAScript array is emitted in a partial rendering page.</td>
</tr>
<tr>
<td>RegisterAsyncPostBackControl</td>
<td>Takes note that the specified control can trigger an asynchronous postback event from within an updatable panel.</td>
</tr>
<tr>
<td>RegisterClientScriptBlock</td>
<td>Static method, ensures that the specified script is emitted in a partial rendering page.</td>
</tr>
<tr>
<td>RegisterClientScriptInclude</td>
<td>Static method, ensures that the markup to import an external script file through the src attribute of the <code>&lt;script&gt;</code> tag is emitted in a partial rendering page.</td>
</tr>
<tr>
<td>RegisterClientScriptResource</td>
<td>Static method, ensures that the markup to import an external script from the page's resources is emitted in a partial rendering page.</td>
</tr>
<tr>
<td>RegisterDataItem</td>
<td>Registers a string of data that will be sent to the client along with the output of a partially rendered page.</td>
</tr>
<tr>
<td>RegisterDispose</td>
<td>Registers controls that require a client script to run at the end of an asynchronous postback to dispose of client resources.</td>
</tr>
<tr>
<td>RegisterExpandoAttribute</td>
<td>Static method, ensures that the markup to import a custom, nonstandard attribute is emitted in a partial rendering page.</td>
</tr>
<tr>
<td>RegisterExtenderControl</td>
<td>Registers an extender control with the current ASP.NET AJAX page.</td>
</tr>
</tbody>
</table>
## Method Description

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>RegisterHiddenField</code></td>
<td>Static method, ensures that the specified hidden field is emitted in a partial rendering page.</td>
</tr>
<tr>
<td><code>RegisterOnSubmitStatement</code></td>
<td>Static method, ensures that that client-side script associated with the form's OnSubmit event is emitted in a partial rendering page.</td>
</tr>
<tr>
<td><code>RegisterPostBackControl</code></td>
<td>Takes note that the specified control can trigger a full postback event from within an updatable panel.</td>
</tr>
<tr>
<td><code>RegisterStartupScript</code></td>
<td>Static method, ensures that client-side script is emitted at the end of the <code>&lt;form&gt;</code> tag in a partial rendering page. In this way, the script will execute as the page refresh is completed.</td>
</tr>
<tr>
<td><code>SetFocus</code></td>
<td>Allows you to move the input focus to the specified client element after an asynchronous postback.</td>
</tr>
</tbody>
</table>

All static methods emit some form of script and markup in the client page. These static methods are the AJAX counterpart of similar methods defined on the page’s `ClientScript` object that you should know from ASP.NET 2.0. The static `RegisterXXX` methods on the `ScriptManager` class ensure that the given piece of script and markup is properly emitted only once in each partial update of the ASP.NET AJAX page. Similarly, other nonstatic `RegisterXXX` methods should be seen as tools to emit proper script code in ASP.NET AJAX pages—especially script code that is associated with custom controls.

### Note

Script registration is an old feature of ASP.NET, in spite of the slight changes that occurred in the transition from version 1.x to 2.0. To most developers, script registration is a pretty neat and clear feature. However, the advent of ASP.NET AJAX extensions mixed things up a little bit. What’s the difference between `RegisterXXX` methods in the `ScriptManager` control and the page’s `ClientScript` object, which is an instance of the `ClientScriptManager` class?

`ClientScriptManager`’s and `ScriptManager`’s registration methods serve the same purpose but in radically different scenarios. You need to use the `ScriptManager`’s methods only if you need to emit script code during an AJAX partial rendering postback operation. An AJAX partial rendering postback operation is processed by the runtime as usual, except for the rendering stage. At this time, the markup is generated and any registered script is emitted. Because during AJAX postbacks the `ScriptManager` is responsible for the markup rendering, it’s the `ScriptManager` that needs to know about registered scripts to emit. If you stick to using `ClientScriptManager`’s methods in an AJAX page, you risk that no script will be emitted during the refresh of an updatable panel. As a result, a portion of your page might display strange and weird behaviors.
Events of the ScriptManager Control

Table 19-5 details the two events fired by the ScriptManager control.

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AsyncPostBackError</td>
<td>Occurs when an exception goes unhandled on the server during an asynchronous postback.</td>
</tr>
<tr>
<td>ResolveScriptReference</td>
<td>Occurs when the ScriptManager control is going to resolve a script reference.</td>
</tr>
</tbody>
</table>

Both events are much more than mere notifications of something that has happened on the server. Both give you good chances to intervene effectively in the course of the application. For example, by handling the ResolveScriptReference event, you can change the location from where the script is going to be downloaded on the client:

```csharp
protected void ResolveScript(object sender, ScriptReferenceEventArgs e)
{
    // Check Path or Name on the e.Script object based on what you've put in Scripts.
    // Next, you specify the real file to load
    if (String.Equals(e.Script.Path, "personal.js", StringComparison.OrdinalIgnoreCase))
    
    e.Script.Path = "person.js";
}
```

By handling the AsyncPostBackError event, you can edit the error message being returned to the client. Here’s an example:

```csharp
protected void AsyncPostBackError(object sender, AsyncPostBackErrorEventArgs e)
{
    ScriptManager sm = sender as ScriptManager;
    if (Request.UserHostAddress == "127.0.0.1")
    {
        sm.AsyncPostBackErrorMessage = String.Format(
            "<b>An error occurred. <br/>{0}<b>",
        e.Exception.Message);
    }
    else
    {
        sm.AsyncPostBackErrorMessage = String.Format(
            "<b>An error occurred. <br/>{0}<b>",
            "Please contact your Web master.");
    }
}
```

When executed locally, the client-side error message appears as you see in Figure 19-5.
What if you don’t like to display the message directly in the page popups and want to redirect the user to an error page instead? In this case, you configure the page to use the traditional error-handling mechanism for ASP.NET pages. You configure the `<customErrors>` section in the `web.config` file and indicate HTML error pages to reach in case of specific errors. (See Chapter 5.) This behavior is fully supported by ASP.NET AJAX and can be disabled by setting to `false` the value of the `AllowCustomErrorRedirects` property of the `ScriptManager` object.

**Note** When an exception is thrown during a partial rendering operation, the HTTP request returns a regular HTTP 200 status code, but instead of including the updated markup, it includes a full description of the error. In ASP.NET AJAX Extensions for ASP.NET 2.0, the default error handler pops up a client-side message box with the exception message or any text you assign to the `AsyncPostBackErrorMessage` property. In ASP.NET 3.5, on the other hand, you get a JavaScript exception.

The `ScriptManagerProxy` Control

Only one instance of the `ScriptManager` control can be added to an ASP.NET AJAX page. However, there are two ways in which you can do this. You can add it directly on the page using the `<asp:ScriptManager>` tag or indirectly by importing a component that already contains a script manager. Typically, you can accomplish the second alternative by importing a user control, creating a content page for a master page, or authoring a nested master page.

What if a content page needs to add a new script reference to the manager? In this case, you need a reference to the script manager. Although it’s defined in the master page (or in a user control), the script manager might not be publicly exposed to the content page. You can use the static method `GetCurrent` on the class `ScriptManager` to get the right instance:

```csharp
// Retrieve the instance of the ScriptManager active on the page
sm = ScriptManager.GetCurrent(this.Page);
```
The **ScriptManagerProxy** class saves you from this sort of coding. In general, in cases where you need features of the **ScriptManager** control but lack a direct reference to it, you can instead include a **ScriptManagerProxy** control in the content page.

You can’t have two script managers in the context of the same page; however, you can have a script manager and a proxy to retrieve it. The **ScriptManagerProxy** control enables you to add scripts and services to nested components, and it enables partial page updates in user controls and nested master pages. When you use the proxy, the Scripts and Services collections on the **ScriptManager** and **ScriptManagerProxy** controls are merged at runtime.

### Note
The **ScriptManagerProxy** class is a very simple wrapper around the **GetCurrent** method of the **ScriptManager** class, and its programming interface is not an exact clone of the **ScriptManager**. From within the proxy, you have access only to a limited number of properties, including Scripts, Services, AuthenticationService, RoleService, and ProfileService. If you need to modify anything else, refer to the **GetCurrent** static method of the **ScriptManager** class.

### Script Binding and Loading

By extensively relying on client capabilities, ASP.NET AJAX requires a lot of script code. The framework itself links a lot of code, as do custom controls and actual user pages. The only HTML-supported way of linking script files is the `<script>` tag and its `src` attribute. The **ScriptManager** control can be used to save yourself the direct manipulation of quite a few `<script>` tags and also obtain richer features, such as built-in management of localized scripts.

You use the Scripts collection to tell the **ScriptManager** about the scripts you want to add to the page. The collection can be accessed either declaratively or programmatically. In addition to the user-requested scripts, the **ScriptManager** control automatically emits in the client page any ASP.NET AJAX required script. This means that, as a page developer, you don’t have to worry about linking the Microsoft AJAX library or any other ASP.NET AJAX native feature. The following example illustrates the script loading model you can use to load optional and custom scripts:

```xml
<asp:ScriptManager runat="server" ID="ScriptManager1">
  <Scripts>
    <asp:ScriptReference
      Name="YourCompany.Scriptlibrary.CoolUI.js"
      Assembly="YourCompany.ScriptLib" />
    <asp:ScriptReference
      Path="/Scripts/MyLib.js" />
  </Scripts>
</asp:ScriptManager>
```

Table 19-6 lists the properties of the **ScriptReference** class by means of which you can control the loading of scripts.
### TABLE 19-6 Events of ScriptManager

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly</td>
<td>Indicates the assembly that contains its resources the script to download on the client.</td>
</tr>
<tr>
<td>IgnoreScriptPath</td>
<td>Boolean value, indicates whether the ScriptPath value optionally set at the top ScriptManager level has to be ignored. This property is set to false by default.</td>
</tr>
<tr>
<td>Name</td>
<td>Name of the script to download on the client.</td>
</tr>
<tr>
<td>NotifyScriptLoaded</td>
<td>Boolean value, indicates whether the script resource loader should automatically append a script-loaded notification statement to let the Sys.Application object know when the script is loaded. This property is set to true by default.</td>
</tr>
<tr>
<td>Path</td>
<td>Indicates the server path where the script to download on the client can be found.</td>
</tr>
<tr>
<td>ResourceUICultures</td>
<td>A comma-delimited string of valid user-interface cultures supported by the path.</td>
</tr>
<tr>
<td>ScriptMode</td>
<td>Specifies the algorithm for choosing between the debug and release versions of the script file. If no debug version exists, the ScriptReference class automatically falls back to release code. Feasible values for the property come from the ScriptMode enumeration type.</td>
</tr>
</tbody>
</table>

You can reference script files, including ASP.NET AJAX system scripts, either from an assembly or from a disk file. There's a benefit in using disk files. You gain something in performance because less work is required to load the script in memory directly from a file. We'll see how to reference from disk the principal ASP.NET AJAX script file—MicrosoftAjax.js—which alone contains two-thirds of the Microsoft AJAX library. The technique is also valid for any custom script file, however.

Normally, you don't take care of MicrosoftAjax.js—you just find it downloaded care of the script manager. If you examine the HTML source of an ASP.NET AJAX page, you can hardly find a reference to such a file. Here's what you find instead:

```html
<script src="/Core35/ScriptResource.axd?d=...&amp;t=...
 type="text/javascript">
</script>
```

Script references obtained from embedded Web resources are served by the ScriptResource.axd HTTP handler. In ASP.NET AJAX, this handler replaces an old acquaintance, the WebResource.axd handler—a native component of ASP.NET 2.0. What's the difference? In addition to serving script references, the ScriptResource.axd handler also appends any localized JavaScript resource types for the file.
Part IV ASP.NET AJAX Extensions

To load a system file from disk, or to load a manually modified version of a system script file, you create a directory structure that roots under a custom folder the following subdirectories:

System.Web.Extensions\3.5.0.0

Now set the ScriptPath property on ScriptManager to a custom parent folder specific to your application. Say, you call it JS:

```<asp:ScriptManager ID="ScriptManager1" runat="server" ScriptPath="~/JS" />```

All of a sudden, the MicrosoftAjax.js script file is now referenced as shown here:

```<script src="~/JS/System.Web.Extensions/3.5.0.0/MicrosoftAjax.js" type="text/javascript" /></script>```

Needless to say, your pages will fail if no such script files can be found in the specified directory path.

Handling Debug and Release Script Files

One of the additional free services offered by ScriptManager that isn't offered by the classic `<script>` tag is the ability to automatically link debug or release script files, as appropriate. ASP.NET uses a special naming convention to distinguish between debug and release script files. Given a release script file named `script.js`, its debug version is expected to be filed as `script.debug.js`.

In general, the main difference between debug and release scripts is that the release scripts remove unnecessary blank characters, comments, trace statements, and assertions. Normally the burden of switching the links to debug and release scripts is left to the developer.

The ScriptManager control takes on this burden and, based on the aforementioned naming convention, distinguishes between debug and release scripts. The ScriptManager control picks debug scripts when the `debug` attribute of the `<compilation>` section in the web.config file is true.

Localized Scripts

Script files can have localizable elements such as text strings for messages and user-interface elements. When the `EnableScriptLocalization` property is set to true and a UI culture is properly set in the page, the script manager automatically retrieves script files for the current culture, if any.
Localization is driven by the `UICulture` attribute in the `@Page` directive and the `UICulture` property in the `Page` class:

```xml
<%@ Page Language="C#" UICulture="it-IT" ... %>
```

This information is not enough for the `ScriptManager` to pick up localized scripts, if any. You also need to specify which UI cultures you intend to support for each referenced script. You indicate the supported cultures through the `ResourceUICultures` property on individual script references. The property is a comma-separated string of culture symbols. Here's an example:

```xml
<asp:ScriptManager ID="ScriptManager1" runat="server" EnableScriptLocalization="true">
    <Scripts>
        <asp:ScriptReference Path="Person.js" ResourceUICultures="it-IT" />
    </Scripts>
</asp:ScriptManager>
```

Note that `ResourceUICultures` is ignored if the `Path` attribute is not specified on the script reference tag.

At this point, if the page requires a script named `person.js` and the UI culture is set to `it-IT`, the `ScriptManager` object attempts to retrieve a script file named `person.it-IT.js` from the same path.

### Script Globalization

Globalization is a programming feature that refers to the code’s ability to support multiple cultures. A request processed on the server has a number of ways to get and set the current culture settings. For example, you can use the `Culture` attribute on the `@Page` directive, the `Culture` property on the `Page` class, or perhaps the `<globalization>` section in the `web.config` file. How can you access the same information on the client from JavaScript?

When the `EnableScriptGlobalization` property is `true`, the `ScriptManager` emits proper script code that sets up a client-side global `Sys.CultureInfo` object that JavaScript classes can consume to display their contents in a culture-based way. Only a few methods and a few JavaScript objects support globalization. In particular, it will work for the `localeFormat` method of `Date`, `String`, and `Number` types. Custom JavaScript classes, though, can be made global by simply calling into these methods or accepting a `Sys.CultureInfo` object in their signatures.

---

**Note** For more information about script loading and script-related features in ASP.NET AJAX pages, refer to Chapter 3 of my book *Introducing ASP.NET AJAX* (Microsoft Press, 2007).
Selective Page Updates with Partial Rendering

AJAX is not a particular technology or product. It refers to a number of client features, and related development techniques, that make Web applications look like desktop applications. AJAX doesn’t require any plug-in modules either and is not browser specific. Virtually any browser released in the past five years can serve as a great host for AJAX-based applications. AJAX development techniques revolve around one common software element—the XMLHttpRequest object. The availability of this object in the object model of most browsers is the key to the current ubiquity and success of AJAX applications. In addition to XMLHttpRequest, a second factor contributes to the wide success of AJAX—the availability of a rich document object model in virtually any browser.

Based on this quick assay of the AJAX paradigm, the programming model of AJAX applications seem to be clear and unquestionable. You write code that captures client-side events, conduct an operation on the server via XMLHttpRequest, get the results, and update the user interface. All the client-side programming is done through JavaScript. This model is a real performance booster when applied to individual features or bottlenecks in existing pages, but it’s hard to scale to a large application because it proves quite expensive in terms of skills to acquire and time to implement.

It is the downside of the loudly requested change of paradigm for Web applications. When it comes to rewriting Web applications for AJAX, nearly all aspects of the application need to be redesigned, refactored, and rewritten. Opting for AJAX all the way might be too much for too many companies; and it’s not a step to take with a light heart.

Today you do much of your ASP.NET programming using server controls. A server control normally emits HTML markup. In an AJAX scenario, a server control emits markup plus some script code to support AJAX requests. This is not exactly the loudly requested change of paradigm, but it is a good compromise between the today’s Web and AJAX. Most third-party vendors prepared their own offering based on this idea. They just provide you with a new set of controls that supply a server and client programming model and manage any browser-to-server communication for you.

But what if you aren’t using any third-party library? Should you write new AJAX-enabled controls yourself? An AJAX server control can be the AJAX version of a traditional server control—for example, an AJAX-enabled drop-down list that supports client insertions and moves them back to the server without a full-page postback. But it can also be a generic control container that takes care of refreshing all of its children without a full-page postback. Enter partial rendering.
ASP.NET partial rendering works according to this idea. It provides a new container control—the `UpdatePanel` control—that you use to surround portions of existing pages, or portions of new pages developed with the usual programming model of ASP.NET 2.0. A postback that originates within any of these updatable regions is managed by the `UpdatePanel` control and updates only the controls in the region.

**The UpdatePanel Control**

In ASP.NET AJAX, partial rendering is the programming technique centered around the `UpdatePanel` control. In ASP.NET, the `UpdatePanel` control represents the shortest path to AJAX. It allows you to add effective AJAX capabilities to sites written according to the classic programming model of ASP.NET 2.0. As a developer, you have no new skills to learn, except the syntax and semantics of the `UpdatePanel` control. The impact on existing pages is very limited, and the exposure to JavaScript is very limited, and even null in most common situations.

You might wonder how partial rendering differs from classic postbacks. The difference is in how the postback is implemented—instead of letting the browser perform a full-page refresh, the `UpdatePanel` control intercepts any postback requests and sends an out-of-band request for fresh markup to the same page URL. Next, it updates the DOM tree when the response is ready. Let’s investigate the programming interface of the control.

**The UpdatePanel Control at a Glance**

The `UpdatePanel` control is a container control defined in the `System.Web.Extensions` assembly. It belongs specifically to the `System.Web.UI` namespace. The control class is declared as follows:

```csharp
public class UpdatePanel : Control
{
    ...
}
```

Although it’s logically similar to the classic ASP.NET `Panel` control, the `UpdatePanel` control differs from the classic panel control in a number of respects. In particular, it doesn’t derive from `Panel` and, subsequently, it doesn’t feature the same set of capabilities as ASP.NET panels, such as scrolling, styling, wrapping, and content management.

The `UpdatePanel` control derives directly from `Control`, meaning that it acts as a mere AJAX-aware container of child controls. It provides no user-interface-related facilities. Any required styling and formatting should be provided through the child controls. In contrast, the control
sports a number of properties to control page updates and also exposes a client-side object model. Consider the following classic ASP.NET code:

```xml
<asp:GridView ID="GridView1" runat="server"
    DataSourceID="ObjectDataSource1"
    AllowPaging="True"
    AutoGenerateColumns="False" Width="450px">
    <Columns>
        <asp:BoundField DataField="ID" HeaderText="ID">
            <ItemStyle Width="70px" />
        </asp:BoundField>
        <asp:BoundField DataField="CompanyName" HeaderText="Company">
            <ItemStyle Width="300px" />
        </asp:BoundField>
        <asp:BoundField DataField="Country" HeaderText="Country">
            <ItemStyle Width="80px" />
        </asp:BoundField>
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    TypeName="Core35.DAL.Customers"
    SelectMethod="LoadAll" />
```

This code causes a postback each time you click to view a new page, edit a record, or sort by a column. As a result, the entire page is redrawn even though the grid is only a small fragment of it. With partial rendering, you take the preceding markup and just wrap it with an `UpdatePanel` control, as shown here:

```xml
<asp:UpdatePanel ID="UpdatePanel1" runat="server">
    <ContentTemplate>
        ...
    </ContentTemplate>
</asp:UpdatePanel>
```

In addition, you need to add a `ScriptManager` control to the page. That's the essence of partial rendering. And it magically just works. Well, not just magically, but it works.

---

**Note** From this simple but effective example, you might be led to think that it suffices that you surround the whole body of the page with an `UpdatePanel` control and you're done. If you do, it certainly works. It might not be particularly efficient though. In the worst case, you need the same bandwidth as you do with classic ASP.NET; however, you still give your users an infinitely better experience because only a portion of the page actually refreshes.

As we'll learn in the rest of the chapter, partial rendering offers a number of attributes to optimize the overall behavior and performance. However, the majority of users are more than happy with the sole effect of a partial page rendering.
The Programming Interface of the Control

Table 19-7 details the properties defined on the UpdatePanel control that constitute the aspects of the control’s behavior that developers can govern.

**TABLE 19-7 Properties of the UpdatePanel Control**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ChildrenAsTriggers</strong></td>
<td>Indicates whether postbacks coming from child controls will cause the UpdatePanel to refresh. This property is set to true by default. When this property is false, postbacks from child controls are ignored. You can’t set this property to false when the UpdateMode property is set to Always.</td>
</tr>
<tr>
<td><strong>ContentTemplate</strong></td>
<td>A template property, defines what appears in the UpdatePanel when it is rendered.</td>
</tr>
<tr>
<td><strong>ContentTemplateContainer</strong></td>
<td>Retrieves the dynamically created template container object. You can use this object to programmatically add child controls to the UpdatePanel.</td>
</tr>
<tr>
<td><strong>IsInPartialRendering</strong></td>
<td>Indicates whether the panel is being updated as part of an asynchronous postback. Note that this property is designed for control developers. Page authors should just ignore it.</td>
</tr>
<tr>
<td><strong>RenderMode</strong></td>
<td>Indicates whether the contents of the panel will be rendered as a block &lt;div&gt; tag or as an inline &lt;span&gt; tag. The feasible values for the property—Block and Inline—are defined in the UpdatePanelRenderMode enumeration. The default is Block.</td>
</tr>
<tr>
<td><strong>UpdateMode</strong></td>
<td>Gets or sets the rendering mode of the control by determining under which conditions the panel gets updated. The feasible values—Always and Conditional—come from the UpdatePanelUpdateMode enumeration. The default is Always.</td>
</tr>
<tr>
<td><strong>Triggers</strong></td>
<td>Defines a collection of trigger objects, each representing an event that causes the panel to refresh automatically.</td>
</tr>
</tbody>
</table>

A bit more explanation is needed for the IsInPartialRendering read-only Boolean property. It indicates whether the contents of an UpdatePanel control are being updated. From this description, it seems to be a fairly useful property. Nonetheless, if you read its value from within any of the handlers defined in a code-behind class, you’ll find out that the value is always false.

As mentioned, IsInPartialRendering is a property designed for control developers only. So if it is assigned its proper value only at rendering time—that is, well past the PreRender event you can capture from a code-behind class. Developers creating a custom version of the UpdatePanel control will likely override the Render method. From within this context, they can leverage the property to find out whether the control is being rendered in a full-page refresh or in a partial rendering operation.
As a page author, if you just need to know whether a portion of a page is being updated as a result of an AJAX postback, you use the IsInAsyncPostBack Boolean property on the ScriptManager control.

**Note**: Like any other ASP.NET AJAX feature, partial rendering requires a ScriptManager control in the page. It is essential, though, that the EnablePartialRendering property on the manager be set to true—which is the default case. If this property is set to false, the UpdatePanel control works like a regular panel.

### Populating the Panel Programatically

The content of an updatable panel is defined through a template property—the ContentTemplate property. Just like any other template property in ASP.NET controls, ContentTemplate can be set programmatically. Consider the following page fragment:

```xml
<asp:ScriptManager ID="ScriptManager1" runat="server" />
<asp:UpdatePanel ID="UpdatePanel1" runat="server">
    <!-- Left empty deliberately. Will be filled out programmatically -->
</asp:UpdatePanel>
```

In the PreInit event of the code-behind page, you can set the ContentTemplate programmatically, as shown here:

```csharp
protected void Page_PreInit(object sender, EventArgs e)
{
    // You could also read the URL of the user control from a configuration file
    string ascx = "customerview.ascx";
    UpdatePanel1.ContentTemplate = this.LoadTemplate(ascx);
}
```

You are not allowed to set the content template past the PreInit event. However, at any time before the rendering stage, you can add child controls programmatically. In ASP.NET, to add or remove a child control, you typically use the Controls property of the parent control, as shown here:

```csharp
UpdatePanel1.Controls.Add(new LiteralControl("Test"));
```

If you try to add a child control programmatically to the Controls collection of an UpdatePanel—as in the preceding code snippet—all that you get is a runtime exception. You should use the ContentTemplateContainer property instead. The reason is that what you really want to do is add or remove controls to the content template, not to the UpdatePanel directly. That's why Controls doesn't work and you have to opt for the actual
container of the template. The following code shows how to populate the content template programmatically:

```csharp
public partial class Samples_Ch19_Partial_Dynamic : System.Web.UI.Page
{
    private Label Label1;
    protected void Page_Load(object sender, EventArgs e)
    {
        UpdatePanel upd = new UpdatePanel();
        upd.ID = "UpdatePanel1";

        // Define the button
        Button button1 = new Button();
        button1.ID = "Button1";
        button1.Text = "What time is it?";
        button1.Click += new EventHandler(Button1_Click);

        // Define the literals
        LiteralControl lit = new LiteralControl("<br>");

        // Define the label
        Label1 = new Label();
        Label1.ID = "Label1";
        Label1.Text = "[time]";

        // Link controls to the UpdatePanel
        upd.ContentTemplateContainer.Controls.Add(button1);
        upd.ContentTemplateContainer.Controls.Add(lit);
        upd.ContentTemplateContainer.Controls.Add(Label1);

        // Add the UpdatePanel to the list of form controls
        this.Form.Controls.Add(upd);
    }

    protected void Button1_Click(object sender, EventArgs e)
    {
        Label1.Text = DateTime.Now.ToShortTimeString();
    }
}
```

You can add an `UpdatePanel` control to the page at any time in the life cycle. Likewise, you can add controls to an existing panel at any time. However, you can't set the content template programmatically past the page's `PreInit` event.

Master Pages and Updatable Regions

You can safely use `UpdatePanel` controls from within master pages. Most of the time, the use of updatable panels is easy and seamless. There are a few situations, though, that deserve a bit of further explanation.
If you add a ScriptManager control to a master page, partial rendering is enabled by default for all content pages. In addition, initial settings on the script manager are inherited by all content pages. What if you need to change some of the settings (for example, add a new script file or switch on script localization) for a particular content page? You can't have a new script manager, but you need to retrieve the original one defined on the master page.

In the content page, you can declaratively reference a ScriptManagerProxy and change some of its settings. The proxy retrieves the script manager currently in use and applies changes to it.

The ScriptManagerProxy control, though, is mostly designed to let you edit the list of scripts and services registered with the manager in a declarative manner, and it doesn't let you customize, say, error handling or script localization. You can do the same (and indeed much more) by programmatically referencing the script manager in the master page. Here's how:

```csharp
protected void Page_Init(object sender, EventArgs e)
{
    // Work around the limitations in the API of the ScriptManagerProxy control
    ScriptManager.GetCurrent(this).EnableScriptLocalization = true;
}
```

In the content page, you create a handler for the page's Init event, retrieve the script manager instance using the static GetCurrent method on the ScriptManager class, and apply any required change.

User Controls and Updatable Regions

User controls provide an easy way to bring self-contained, auto-updatable AJAX components into an ASP.NET page. Because each page can have at most one script manager, you can't reasonably place the script manager in the user control. That would work and make the user control completely self-contained, but it would also limit you to using exactly one instance of the user control per page. On the other hand, the UpdatePanel control requires a script manager. Multiple script managers, or the lack of at least one script manager, will cause an exception.

The simplest workaround is that you take the script manager out of the user control and place it in the host page. User controls therefore assume the presence of a script manager, and they use internally as many updatable panels as needed:

```xml
<asp:ScriptManager runat="server" ID="ScriptManager1" />
<x:Clock runat="server" ID="Clock1" />
<br />
<x:Clock runat="server" ID="Clock2" />
```
Note You can’t call `Response.Write` from within a postback event handler (for example, `Button1_Click`) that gets called during an asynchronous AJAX postback. If you do so, you’ll receive a client exception stating that the message received from the server could not be parsed. In general, calls to `Response.Write`—but also response filters, HTTP modules, or server tracing (`Trace=true`)—modify the stream returned to the client by adding explicit data that alters the expected format.

Optimizing the Usage of the `UpdatePanel` Control

Partial rendering divides the page into independent regions, each of which controls its own postbacks and refreshes without causing, or requiring, a full-page update. This behavior is desirable when only a portion—and perhaps only a small portion—of the page needs to change during a postback. Partial updates reduce screen flickering and allow you to create more interactive Web applications. An ASP.NET page can contain any number of `UpdatePanel` controls.

An `UpdatePanel` control refreshes its content under the following conditions:

- When another `UpdatePanel` control in the same page refreshes
- When any of the child controls originates a postback (for example, a button click or a change of selection in a drop-down list with `AutoPostBack=true`)
- When handling a postback event the page invokes the `Update` method on the `UpdatePanel` control
- When the `UpdatePanel` control is nested inside another `UpdatePanel` control and the parent update panel is updated
- When any of the trigger events for the `UpdatePanel` occur

You can control these conditions through a number of properties such as `UpdateMode`, `ChildrenAsTriggers`, and the collection `Triggers`. To minimize the total number of postbacks and the amount of data being roundtripped, you should pay a lot of attention to the values you assign to these properties. Let’s delve deeper into this topic.

Configuring for Conditional Refresh

By default, all updatable panels in a page are synchronized and refresh at the same time. To make each panel refresh independently from the others, you change the value of the `UpdateMode` property. The default value is `Always`, meaning that the panel’s content is updated on every postback that originates from anywhere in the page, from inside and outside the updatable region.
By changing the value of the `UpdateMode` property to `Conditional`, you instruct the updatable panel to update its content only if it is explicitly ordered to refresh. This includes calling the `Update` method, intercepting a postback from a child control, or any of the events declared as triggers.

Normally, any control defined inside of an `UpdatePanel` control acts as an implicit trigger for the panel. You can stop all child controls from being triggers by setting the value of `ChildrenAsTriggers` to `false`. In this case, a button inside an updatable panel, if clicked, originates a regular full postback.

What if you want only a few controls within an `UpdatePanel` to act as triggers? You can define them as triggers of a particular `UpdatePanel`, or you can use the `RegisterAsyncPostBackControl` method on the `ScriptManager` class.

The `RegisterAsyncPostBackControl` method enables you to register controls to perform an asynchronous postback instead of a synchronous postback, which would update the entire page. Here is an example of the `RegisterAsyncPostBackControl` method:

```csharp
protected void Page_Load(object sender, EventArgs e)
{
    ScriptManager1.RegisterAsyncPostBackControl(Button1);
}
```

The control object you pass as an argument will be a control not included in any updatable panels and not listed as a trigger. The effects of the postback that originates from the control differ with regard to the number of `UpdatePanel` controls in the page. If there's only one `UpdatePanel` in the page, the script manager can easily figure out which one to update. The following code shows a page whose overall behavior may change if one or two `UpdatePanel` controls are used.

```csharp
protected void Button1_Click(object sender, EventArgs e)
{
    // If there's only one UpdatePanel in the page, and it includes this Label control,
    // the panel is refreshed automatically.
    Label1.Text = "Last update at: " + DateTime.Now.ToLongTimeString();

    // This Label control, not included in any UpdatePanel, doesn't have its UI
    // refreshed, its state, though, is correctly updated.
    Label2.Text = "Last update at: " + DateTime.Now.ToLongTimeString();
}
```

When multiple panels exist, to trigger the update you have to explicitly invoke the `Update` method on the panel you want to refresh:

```csharp
protected void Button1_Click(object sender, EventArgs e)
{
    Label1.Text = "Last update at: " + DateTime.Now.ToLongTimeString();
    UpdatePanel1.Update();
}
```
All controls located inside of an **UpdatePanel** control are automatically passed as an argument to the `RegisterAsyncPostBackControl` method when `ChildrenAsTriggers` is true.

**Note** A postback that originates from within an **UpdatePanel** control is often referred to as an asynchronous postback or an AJAX postback. Generally, these expressions are used to reference a postback conducted via a script taking advantage of XMLHttpRequest.

**Programmatic Updates**

We have already mentioned the `Update` method quite a few times. It’s time we learn more about it, starting with its signature:

```csharp
public void Update()
```

The method doesn’t take any special action itself, but is limited to requiring that the child controls defined in the content template of the **UpdatePanel** control be refreshed. By using the `Update` method, you can programatically control when the page region is updated in response to a standard postback event or perhaps during the initialization of the page.

An invalid operation exception can be thrown from within the `Update` method in a couple of well-known situations. One situation is if you call the method when the `UpdateMode` property is set to `Always`. The exception is thrown in this case because a method invocation prefigures a conditional update—you do it when you need it—which is just the opposite of what the `Always` value of the `UpdateMode` property indicates. The other situation in which the exception is thrown is when the `Update` method is called during or after the page’s rendering stage.

So when should you get to use the `Update` method in your pages?

You resort to the method if you have some server logic to determine whether an **UpdatePanel** control should be updated as the side effect of an asynchronous postback—whether it is one that originated from another **UpdatePanel** in the page or a control registered as an asynchronous postback control.

**Using Triggers**

As mentioned, you can associate an **UpdatePanel** control with a list of server-side events. Whenever a registered event is triggered over a postback, the panel is updated. Triggers can be defined either declaratively or programmatically. You add an event trigger declaratively using the `<Triggers>` section of the **UpdatePanel** control:

```xml
<asp:UpdatePanel runat="server" ID="UpdatePanel1">
   <ContentTemplate>
      ...
   </ContentTemplate>
</asp:UpdatePanel>
```
You need to specify two pieces of information for each trigger—the ID of the control to monitor, and the name of the event to catch. It is essential to note that the AsyncPostBackTrigger component can catch only server-side events. Both ControlID and EventName are string properties. For example, the panel described in the previous code snippet is refreshed when any of the controls in the page posts back (that is, its UpdateMode property defaults to Always) or when the selection changes on a drop-down list control named DropDownList1.

Note Keep in mind that we're talking about server-side events here. This implies that, in the previous example, the DropDownList1 control must have AutoPostBack equals to true in order to fire a postback.

The event associated with the AsyncPostBackTrigger component triggers an asynchronous AJAX postback on the UpdatePanel control. As a result, the host page remains intact except for the contents of the referenced panel and its dependencies, if any. Usually, the AsyncPostBackTrigger component points to controls placed outside the UpdatePanel. However, if the panel has the ChildrenAsTriggers property set to false, it could make sense for you to define an embedded control as the trigger. In both cases, when a control that is a naming container is used as a trigger, all of its child controls that cause postback behave as triggers.

Note You can also add triggers programmatically by using the Triggers collection of the UpdatePanel control. The collection accepts instances of the AsyncPostBackTrigger class.

Full Postbacks from Inside Updatable Panels

By default, all child controls of an UpdatePanel that post back operate as implicit asynchronous postback triggers. You can prevent all of them from triggering a panel update by setting ChildrenAsTriggers to false. Note that when ChildrenAsTriggers is false postbacks coming from child controls are processed as asynchronous postbacks and they modify the state of involved server controls, but they don’t update the user interface of the panel.
There might be situations in which you need to perform full, regular postbacks from inside an UpdatePanel control in response to a control event. In this case, you use the PostBackTrigger component, as shown here:

```xml
<asp:UpdatePanel runat="server" ID="UpdatePanel1">
    <ContentTemplate>
        ...
    </ContentTemplate>
    <Triggers>
        <asp:AsyncPostBackTrigger ControlID="DropDownList1" EventName="SelectedIndexChanged" />
        <asp:PostBackTrigger ControlID="Button1" />
    </Triggers>
</asp:UpdatePanel>
```

The preceding panel features both synchronous and asynchronous postback triggers. The panel is updated when the user changes the selection on the drop-down list; the whole host page is refreshed when the user clicks the button.

A PostBackTrigger component causes referenced controls inside an UpdatePanel control to perform regular postbacks. These triggers must be children of the affected UpdatePanel.

The PostBackTrigger object doesn't support the EventName property. If a control with that name is causing the form submission, the ASP.NET AJAX client script simply lets the request go as usual. The ASP.NET runtime then figures out which server postback event has to be raised for the postback control by looking at its implementation of IPostBackEventHandler.

**Note** When should you use a PostBackTrigger component to fire a full postback from inside an updatable panel? If you need, say, a button to refresh a given panel, why not list the Click event of the button as an asynchronous trigger and leave the button outside the panel? Especially when complex and templated controls are involved, it might not be easy to separate blocks of user interface in distinct panels and single controls. So the easiest, and often the only, solution is wrapping a whole block of user interface in an updatable panel. If a single control in this panel needs to fire a full postback, you need to use the PostBackTrigger component.

**Practical Steps for Adopting Updatable Panels**

The UpdatePanel control works with the idea of limiting the refresh of the page to only the portions of it that are touched by the postback. A clear mapping between user actions and portions of the page that are updated consequently is key to successfully adopting the UpdatePanel control in an ASP.NET site.
The first practical step for successfully migrating page behavior to partial rendering entails that you, given the expected behavior of the page, identify the portions of the page subject to refresh. If you have, say, a complex table layout but only a small fragment of only one cell changes in the page lifetime, there’s no reason to keep the whole table in an UpdatePanel control. Only the server-side control that displays the modifiable text should be wrapped by the panel.

The portions of the page that you should consider to be candidates to be wrapped by an UpdatePanel control should be as small as possible. They also should include the minimum amount of markup and ASP.NET controls.

The second step consists of associating each candidate region with a list of refresh conditions. You basically answer the question, “When does this region get updated?” After you have compiled a list of candidate regions, and for each you have a list of refresh events, you’re pretty much done.

The final step is mapping this information to UpdatePanel controls and triggers. If all the regions you have identified are disjointed, you’re fine. If not, you use properties and triggers on the UpdatePanel control to obtain the expected page behavior, thereby minimizing the impact of postbacks and page flickering.

If needed, updatable panels can be nested. There’s no syntax limitation to the levels of nesting allowed. Just consider that any nested panel refreshes when its parent is refreshed regardless of the settings.

Let’s be honest. It might not be a trivial task, and getting a disjoint set of regions is not always possible. However, given the number of properties supported by the UpdatePanel control, there’s always room for a good compromise between user experience and performance.

**Giving Feedback to the User**

A partial rendering operation still requires a postback; it still uploads and downloads the view state and fires the well-known page life cycle on the server. The benefits of an asynchronous postback lie in the fact that no full-page refresh is required and only a smaller amount of HTML markup is returned to the client. An asynchronous postback might still take a few seconds to complete if the server operation is a lengthy one.

The mechanics of the asynchronous postback keeps the displayed page up and running. So the biggest improvement of AJAX—the continuous feel with the page—can become its major weakness if not handled properly. Having the computer engaged in a potentially long task might be problematic. Will the user resist the temptation of clicking that button over and over again? Will the user patiently wait for the results to show up? Finally, will the user be frustrated and annoyed by waiting without any clue of what’s going on? After all, if the
page is sustaining a full postback, the browser itself normally provides some user feedback that this is happening. Using ASP.NET AJAX, the callback doesn’t force a regular full postback and the browser’s visual feedback system is not called upon to inform the user things are happening.

In the end, AJAX and partial rendering let developers arrange pages that provide "continuous feel" to users and increased responsiveness. The continuous experience, however, raises new issues. Feedback should be given to users to let them know that an operation is taking place. In addition, user-interface elements should be disabled if the user would start new operations by clicking on the element.

ASP.NET AJAX supplies the UpdateProgress control to display a templated content while any of the panels in the page are being refreshed.

The UpdateProgress Control

The UpdateProgress control is designed to provide any sort of feedback on the browser while one or more UpdatePanel controls are being updated. If you have multiple panels in the page, you might want to find a convenient location in the page for the progress control or, if possible, move it programmatically to the right place with respect to the panel being updated. You can use cascading style sheets (CSSs) to style and position the control at your leisure.

The user interface associated with an UpdateProgress control is displayed and hidden by the ASP.NET AJAX framework and doesn’t require you to do any work on your own. The UpdateProgress control features the properties listed in Table 19-8.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AssociatedUpdatePanelID</td>
<td>Gets and sets the ID of the UpdatePanel control that this control is associated with.</td>
</tr>
<tr>
<td>DisplayAfter</td>
<td>Gets and sets the time in milliseconds after which the progress template is displayed. Set to 500 by default.</td>
</tr>
<tr>
<td>DynamicLayout</td>
<td>Indicates whether the progress template is dynamically rendered in the page. Set to true by default.</td>
</tr>
<tr>
<td>ProgressTemplate</td>
<td>Indicates the template displayed during an asynchronous postback that is taking longer than the time specified through the DisplayAfter property.</td>
</tr>
</tbody>
</table>

An UpdateProgress control can be bound to a particular UpdatePanel control. You set the binding through the AssociatedUpdatePanelID string property. If no updatable panel is specified, the progress control is displayed for any panels in the page. The user interface of the progress bar is inserted in the host page when the page is rendered. However, it is initially hidden from view using the CSS display attribute.
When set to none, the CSS display attribute doesn't display a given HTML element and reuses its space in the page so that other elements can be shifted up properly. When the value of the display attribute is toggled on, existing elements are moved to make room for the new element.

If you want to reserve the space for the progress control and leave it blank when no update operation is taking place, you just set the DynamicLayout property to false.

**Composing the Progress Screen**

The ASP.NET AJAX framework displays the contents of the ProgressTemplate property while waiting for a panel to update. You can specify the template either declaratively or programmatically. In the latter case, you assign the property any object that implements the ITemplate interface. For the former situation, you can easily specify the progress control’s markup declaratively, as shown in the following code:

```xml
<asp:UpdateProgress runat="server" ID="UpdateProgress1">
  <ProgressTemplate>
  ...
  </ProgressTemplate>
</asp:UpdateProgress>
```

You can place any combination of controls in the progress template. However, most of the time, you’ll probably just put some text there and an animated GIF. (See Figure 19-6.)

![Figure 19-6 A progress template informing users that some work is occurring on the server](image)

Note that the UpdateProgress control is not designed to be a gauge component, but rather a user-defined panel that the ScriptManager control shows before the panel refresh begins and that it hides immediately after its completion.
Important If you’re looking for a real gauge bar to monitor the progress of a server-side task, partial rendering and the UpdateProgress control are not the right tools. As we’ll see later in the chapter, polling is one of the main drawbacks of partial rendering and polling is unavoidable for monitoring server tasks from the client.

Client-Side Events for Richer Feedback

Each asynchronous postback is triggered on the client via script. The entire operation is conducted by the PageRequestManager client object, which invokes, under the hood, the XMLHttpRequest object. What kind of control do developers have on the underlying operation? If you manage XMLHttpRequest directly, you have full control over the request and response. But when these key steps are managed for you, there’s not much you can do unless the request manager supports an eventing model.

The Sys.WebForms.PageRequestManager object provides a few events so that you can customize handling of the request and response. Table 19-9 lists the supported events that signal the main steps around an AJAX postback that partially update a page. The events are listed in the order in which they fire to the client page.

<table>
<thead>
<tr>
<th>Event</th>
<th>Event Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>initializeRequest</td>
<td>InitializeRequestEventArgs</td>
<td>Occurs before the request is prepared for sending</td>
</tr>
<tr>
<td>beginRequest</td>
<td>BeginRequestEventArgs</td>
<td>Occurs before the request is sent</td>
</tr>
<tr>
<td>pageLoading</td>
<td>PageLoadingEventArgs</td>
<td>Occurs when the response has been acquired but before any content on the page is updated</td>
</tr>
<tr>
<td>pageLoaded</td>
<td>PageLoadedEventArgs</td>
<td>Occurs after all content on the page is refreshed as a result of an asynchronous postback</td>
</tr>
<tr>
<td>endRequest</td>
<td>EndRequestEventArgs</td>
<td>Occurs after an asynchronous postback is finished and control has been returned to the browser</td>
</tr>
</tbody>
</table>

To register an event handler, you use the following JavaScript code:

```javascript
var manager = Sys.WebForms.PageRequestManager.getInstance();
manager.addBeginRequest(OnBeginRequest);
```

The prototype of the event handler method—OnBeginRequest in this case—is shown here:

```javascript
function beginRequest(sender, args)
```

The real type of the args object, though, depends on the event data structure. By using any of these events, you can control in more detail the steps of an asynchronous request. Let’s dig out more.

The initializeRequest event is the first in the client life cycle of an asynchronous request. The life cycle begins at the moment in which a postback is made that is captured by the UpdatePanel’s client-side infrastructure. You can use the initializeRequest event to evaluate the postback source and do any additional required work. The event data structure is the InitializeRequestEventArgs class. The class features three properties—postBackElement, request, and cancel.

The postBackElement property is read-only and evaluates to a DomElement object. It indicates the DOM element that is responsible for the postback. The request property (read-only) is an object of type Sys.Net.WebRequest and represents the ongoing request. Finally, cancel is a read-write Boolean property that can be used to abort the request before it is sent.

Immediately after calling the initializeRequest handler, if any, the PageRequestManager object aborts any pending async requests. Next, it proceeds with the beginRequest event and then sends the packet.

When the response arrives, the PageRequestManager object first processes any returned data and separates hidden fields, updatable panels and whatever pieces of information are returned from the server. Once the response data is ready for processing, the PageRequestManager object fires the pageLoading client event. The event is raised after the server response is received but before any content on the page is updated. You can use this event to provide a custom transition effect for updated content or to run any clean-up code that prepares the panels for the next update. The event data is packed in an instance of the class PageLoadingEventArgs. The class has three properties: panelsUpdating, panelsDeleting, and dataItems. The first two are arrays and list the updatable panels to be updated and deleted, respectively.

The pageLoaded event is raised after all content on the page is refreshed. You can use this event to provide a custom transition effect for updated content, such as flashing or highlighting updated contents. The event data is packed in the class PageLoadedEventArgs, which has three properties: panelsUpdated, panelsDeleted, and dataItems. The first two are arrays and list the updatable panels that were just updated and deleted, respectively.

The endRequest event signals the termination of the asynchronous request. You receive this event regardless of the success or failure of the asynchronous postback.
Disabling Visual Elements During Updates

If you want to prevent users from generating more input while a partial page update is being processed, you can also consider disabling the user interface—all or in part. To do so, you write handlers for `beginRequest` and `endRequest` events:

```javascript
<script type="text/javascript">
function pageLoad()
{
  var manager = Sys.WebForms.PageRequestManager.getInstance();
  manager.add_beginRequest(OnBeginRequest);
  manager.add_beginRequest(OnEndRequest);
}
</script>
```

You typically use the `beginRequest` event to modify the user interface as appropriate and notify the user that the postback is being processed:

```javascript
// Globals
var currentPostBackElem;
function OnBeginRequest(sender, args)
{
  // Get the reference to the button click (i.e., btnStartTask)
  currentPostBackElem = args.get_postBackElement();
  if (typeof(currentPostBackElem) === "undefined")
    return;
  if (currentPostBackElem.id.toLowerCase() === "btnStartTask")
  {
    // Disable the button
    $get("btnStartTask").disabled = true;
  }
}
```

The `beginRequest` handler receives event data through the `BeginRequestEventArgs` data structure—the `args` formal parameter. The class features only two properties—`request` and `postBackElement`. The properties have the same characteristics of analogous properties on the aforementioned `InitializeRequestEventArgs` class.

In the preceding code snippet, I disable the clicked button to prevent users from repeatedly clicking the same button.

At the end of the request, any temporary modification to the user interface must be removed. So animations must be stopped, altered styles must be restored, and disabled controls must be re-enabled. The ideal place for all these operations is the `endRequest` event.

The event passes an `EndRequestEventArgs` object to handlers. The class has a few properties, as described in Table 19-10.
### TABLE 19-10 Properties of the EndRequestEventArgs Control

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dataItems</td>
<td>Returns the client-side dictionary packed with server-defined data items for the page or the control that handles this event. (More on registering data items later.)</td>
</tr>
<tr>
<td>Error</td>
<td>Returns an object of type Error that describes the error (if any) that occurred on the server during the request.</td>
</tr>
<tr>
<td>errorHandled</td>
<td>Gets and sets a Boolean value that indicates whether the error has been completely handled by user code. If this property is set to true in the event handler, no default error handling will be executed by the ASP.NET AJAX client library. We saw an example of this property in Chapter 3.</td>
</tr>
<tr>
<td>Response</td>
<td>Returns an object of type Sys.Net.WebRequest_executor that represents the executor of the current request. Most of the time, this object will be an instance of Sys.Net.XMLHttpExecutor. For more information, refer to Chapter 2.</td>
</tr>
</tbody>
</table>

As you can see, when the endRequest event occurs there’s no information around about the client element that fired the postback. If you need to restore some user interface settings from inside the endRequest event handler, you might need a global variable to track which element caused the postback:

```javascript
function OnEndRequest(sender, args)
{
    if (typeof(currentPostBackElem) === “undefined”)
        return;
    if (currentPostBackElem.id.toLowerCase() === “btnStartTask”)
    {
        $get(“btnStartTask”).disabled = false;
    }
}
```

Wouldn’t it be nice if you could visually notify users that a certain region of the screen has been updated? As we’ve seen, partial rendering improves the user experience with pages by eliminating a good number of full refreshes. If you look at it from the perspective of the average user, though, a partial page update doesn’t have a clear start and finish like a regular Web roundtrip. The user doesn’t see the page redrawn and might not notice changes in the user interface. A good pattern is employing a little animation to show the user what has really changed with the latest operation. You can code this by yourself using the pair of beginRequest and endRequest events, or you can resort to a specialized component—an UpdatePanel extender control—as we’ll see in a moment.

---

**Important** The disabled HTML attribute works only on INPUT elements. It has no effect on hyperlinks and `<a>` tags. If you plan to use LinkButton controls, you have to resort to other JavaScript tricks to disable the user interface. One possible trick is temporarily replacing the onclick handler of the hyperlink with a return value of false. Another effective trick might be to cover the area to be disabled with a partially opaque DIV.
Aborting a Pending Update

A really user-friendly system always lets its users cancel a pending operation. How can you obtain this functionality with an UpdateProgress control? The progress template is allowed to contain an abort button. The script code injected in the page will monitor the button and stop the ongoing asynchronous call if it’s clicked. To specify an abort button, you add the following to the progress template:

```html
<input type="button" onclick="abortTask()" value="Cancel" />
```

In the first place, the button has to be a client-side button. So you can express it either through the `<input>` element or the `<button>` element for the browsers that support this element. If you opt for the `<input>` element, the `type` attribute must be set to `button`. The script code you wire up to the `onclick` event is up to you, but it will contain at least the following instructions:

```javascript
<script type="text/JavaScript">
function abortTask()
{
  var manager = Sys.WebForms.PageRequestManager.getInstance();
  if (manager.get_isInAsyncPostBack())
    manager.abortPostBack();
}
</script>
```

You retrieve the instance of the client `PageRequestManager` object active in the client page and check whether an asynchronous postback is going on. If so, you call the `abortPostBack` method to stop it.

---

**Important** Canceling an ongoing update in this way is equivalent to closing the connection with the server. No results will ever be received, and no updates will ever occur on the page. However, canceling the update is a pure client operation and has no effect over what’s happening on the server. If the user started a destructive operation, the client-side Cancel button can just do nothing to stop that operation on the server.

Light and Shade of Partial Rendering

Partial rendering is definitely the easiest way to add AJAX capabilities to an ASP.NET Web site. It has a relatively low impact on the structure of existing pages, doesn’t require significant new skills, doesn’t require exposure to JavaScript, and leaves the application model intact. Advocates of a pure AJAX approach might say that there’s no AJAX at all in partial rendering. And such a statement is not false, indeed.

Haven’t we said that AJAX is key because it propounds a new programming paradigm for building Web applications? And now we’re back to giving kudos to partial rendering—an
Part IV  ASP.NET AJAX Extensions

approach that admittedly maintains the old programming model of classic Web applications? What's the point?

Overall, partial rendering is only one possible way to approach AJAX. It preserves most of your current investments and is relatively cheap to implement. Partial rendering just adds AJAX capabilities to your pages. It doesn’t constitute a true AJAX application. There’s no architectural new point in partial rendering. It’s a great technique to quickly update legacy applications, and it is an excellent choice when you lack the time, skills, or budget to move on and redesign the application. But in a good number of cases, an improved user interface and optimized rendering is all that your users demand. So partial rendering would perfectly fit in.

On the other hand, building true AJAX applications where all the presentation logic lives on the client written in JavaScript is not trivial either, no matter how much help third-party libraries might offer.

In the end, you should be aware of the structural limitations that partial rendering has. You might want to start with partial rendering to improve your pages and then move on to other, more purely AJAX, solutions to fix particular bottlenecks that still remain. My advice is that a pure AJAX approach where a lot of JavaScript is involved is a solution that should be considered carefully. And that you should have good reasons for both adopting or refusing it.

Note  Why is it so darned hard to write pure AJAX applications? AJAX applications are all about the client, and the client is JavaScript and HTML. Both have significant limitations in light of the complexity of applications these days. JavaScript is an interpreted language, and it does not have a particularly modern syntax. Additionally, JavaScript is subject to the implementation that browsers provide. So a feature might be flaky in one browser and super-optimized in another. Originally born as a document format, HTML is used more as an application delivery format. But for this purpose, HTML is simply inadequate because it lacks strong and built-in graphics and layout capabilities. Silverlight 2.0 with its embedded common language runtime (CLR), support for managed languages, and full support for Windows Presentation Foundation (WPF) seems to address both issues.

Issues with Concurrent Calls

Partial rendering has a number of positives, but it also has a couple of key drawbacks. In particular, it doesn’t support concurrent asynchronous postbacks. This means that you are not allowed to have two asynchronous postbacks going on at the same time. Partial rendering bypasses the standard browser’s mechanism that handles an HTTP request. It hooks up the submit event of the form, cuts the standard browser handler out, and finally places the HTTP request using XMLHttpRequest.

The request that reaches the Web server differs from a regular ASP.NET request only for an extra HTTP header. The request sends in the contents of the posting form, including the view-state hidden field. The response is not pure HTML but represents a text record where
each field describes the new status of a page element—update panels, hidden fields, scripts to run on loading.

As you can see, the underlying model of partial rendering is still the model of classic ASP.NET pages. It is a sort of stop-and-go model where the users posts back, waits for a while, and then receives a new page. While waiting for the next page, there’s not much the user can do. Only one server operation per session occurs at a time. Partial rendering is only a smarter way of implementing the old model.

From a technical standpoint, the major factor that prevents multiple asynchronous postbacks is the persistence of the view-state information. When two requests go, both send out the same copy of the view state, but each reasonably returns a different view state. Which one has to be good for the page then?

Is dropping the view state entirely an option, at least for asynchronous postbacks? Whatever way you look at it, dropping the view state increases the amount of JavaScript needed for each page. The view state allows you to keep the server logic in C# or Visual Basic .NET and generate the user interface through server controls. This is not to say, though, that another approach isn’t possible. Anyway, partial rendering works this way.

Whenever a request for an asynchronous postback is raised, the `PageRequestManager` class checks whether another operation is pending. If so, by default, it silently kills the ongoing request to make room for the new one—a last-win discipline.

This fact has a clear impact on developers. In fact, you should always modify the user interface to ensure that users can’t start a second operation before the first is terminated. Otherwise, the first operation is aborted in favor of the second. This happens in any case, even when the two operations are logically unrelated.

**Note** When concurrent calls are necessary, you should consider moving that page (if not the whole application) to a more AJAX-oriented design. Alternatively, you can consider implementing that feature within the page using some of the features covered in the next chapter, such as page methods or script services.

**Issues with Polling**

Among other things, AJAX pages are popular because they can bring on the client information in a timely manner. A page starts polling a remote URL, grabs fresh information, and returns it to the client for the actual display. Implemented via partial rendering, polling is subject to being interrupted when the user starts a new partial rendering operation to restart automatically at the end.
If this is not a problem for you, you can use the new `Timer` server control, as shown here:

```xml
<asp:Timer ID="Timer1" runat="server" Enabled="true" Interval="1000" ontick="Timer1_Tick" />
<asp:Button ID="Button1" runat="server" Text="Start task" onclick="Button1_Click" />
<asp:UpdateProgress ID="UpdateProgress1" runat="server" DynamicLayout="false">
  <ProgressTemplate>
    <img src="loading.gif" />
  </ProgressTemplate>
</asp:UpdateProgress>
<asp:UpdatePanel ID="UpdatePanel1" runat="server">
  <ContentTemplate>
    <asp:Label ID="Label1" runat="server" />
  </ContentTemplate>
  <Triggers>
    <asp:AsyncPostBackTrigger ControlID="Button1" EventName="Click" />
  </Triggers>
</asp:UpdatePanel>
<br />
<asp:UpdatePanel ID="UpdatePanel2" runat="server">
  <ContentTemplate>
    <asp:Label ID="lblClock" runat="server" />
  </ContentTemplate>
  <Triggers>
    <asp:AsyncPostBackTrigger ControlID="Timer1" EventName="Tick" />
  </Triggers>
</asp:UpdatePanel>
```

The `Timer` control is the server counterpart of a client timer created using the `window.setTimeout` method. In the preceding code, the `Timer` control causes a postback every second as specified by the `Interval` property. The postback fires the `Tick` event. By using the timer as the trigger of an updatable panel, you can refresh the content of the panel periodically. In the code, the second `UpdatePanel` control just renders out a digital clock:

```csharp
protected void Timer1_Tick(object sender, EventArgs e)
{
    // Update the clock
    lblClock.Text = DateTime.Now.ToString();
}
```

As in Figure 19-7, the clock stops working while the remote task triggered by the other button is still running.
The AJAX Control Toolkit

In addition to partial rendering, developers can use control extenders to add a predefined client-side behavior to new and existing ASP.NET controls. A client-side behavior is a block of JavaScript code that adds a new capability to the markup generated by a given ASP.NET control. An extender is basically a server control that emits proper script code—the client behavior—to enhance how a given ASP.NET control behaves on the client. An extender is not simply a custom control derived from an existing control. Rather, it represents a general behavior—such as auto-completion, focus management, generation of popups, and draggability—that can be declaratively applied to various target control types.

For the time being, there's no way to add rich capabilities and functionalities to the Web client other than by crafting good and tricky JavaScript code. Control extenders offer a suitable model, but other frameworks are available too, including Dojo (http://www.dojotoolkit.org) and Gaia (http://www.ajaxwidgets.com). Plus suites of controls are available from large endors such as ComponentArt (http://www.componentart.com), Infragistics (http://www.infragistics.com), and Telerik (http://www.telerik.com).

The AJAX Control Toolkit (ACT) is a shared-source library of Web controls specifically designed for ASP.NET. It is not included in the ASP.NET 3.5 platform and should be downloaded separately. You can get it from http://www.asp.net/ajax/ajaxcontroltoolkit.

Enhancing Controls with Extenders

ASP.NET comes with a fairly rich collection of built-in controls. In addition, plenty of custom controls are available for developers from third-party vendors, from community projects, and even from contributions by volunteers. If you still can't find the control you are looking
for, you typically write one yourself or buy a new specialized control that extends the
original control and adds the desired behavior. Object orientation, of course, encourages this
approach.

All in all, it's rare that you need to write a completely new control yourself. More often, your
control will derive from an existing ASP.NET control. Blindly using inheritance for building
specialized versions of controls might not be a wise choice, though. Even in relatively small
projects, in fact, it can lead straight to a proliferation of controls.

ASP.NET control extenders go in the opposite direction. First and foremost, an extender con-
trol is a server control itself. An extender represents a logical behavior that can be attached
to one or more control types to extend their base capabilities. Extenders decouple controls
from behaviors and make it possible to extend existing controls with new behaviors.

From a technology point of view, ASP.NET AJAX and extenders are not strictly related. In the-
ory, one could develop extenders for ASP.NET 1.1 and ASP.NET 2.0 that work without AJAX
extensions. In practice, though, ACT provides some interesting facilities for writers of extend-
er controls—specifically, base classes and, more importantly, the Microsoft AJAX library for
developing JavaScript functionalities more comfortably.

Extenders at a Glance: the TextBoxWatermark Control

To better understand the goals and characteristics of control extenders, let's briefly
consider the behavior encapsulated by one of the extenders contained in the ACT—the
TextBoxWatermark extender.

A text box watermark is a string of text that is displayed in an empty text box as a guide to
the user. This help text is stripped off when the text box is submitted and is automatically re-
moved as the user starts typing in the field. Likewise, it is automatically re-inserted when the
user wipes out any text in the text box. You start by linking the ACT assembly to the project
and then place an extendee TextBox control in the page:

```html
<%@ Register Assembly="AjaxControlToolkit" Namespace="AjaxControlToolkit" TagPrefix="act" %>
...
<asp:TextBox ID="TextBox1" runat="server" />
Later in the ASPX source, you add a new control—the TextBoxWatermarkExtender control:

```html
<act:TextBoxWatermarkExtender runat="server" ID="TextBoxWatermark1"
  TargetControlID="TextBox1"
  WatermarkText="Type First Name Here"
  WatermarkCssClass="watermarked" />
```

The watermark extender targets the specified control ID and adds a new behavior to it. The
behavior is further configured using a few public properties on the extender control, such as
WatermarkText and WatermarkCssClass in the previous example.
In particular, the watermark behavior injects script code that hooks up three HTML events: `onfocus`, `onblur`, and `onkeypress`. In its initialization stage, the injected script also sets a new style and default text for the target text box if the body of the field is empty. When the text box gets the input focus, the event handler promptly removes the watermark text and restores the original style. As the user types, the handler for `onkeypress` ensures that the current text box is watermarked. Finally, when the input field loses the focus—the `onblur` event—the handler sets the watermark back if the content of the field is the empty string.

**Note** To add a watermark behavior to an ASP.NET `TextBox`, you use the aforementioned extender control. Alternatively, if you feel comfortable with ASP.NET control development and JavaScript, you can develop a custom `TextBox` control and use a client-side code fragment to achieve the same results.

### Creating a New Extender Control

ASP.NET 3.5 doesn’t include any concrete implementation of an extender. However, it defines the base class from which all custom extenders, as well as all extenders in the ACT, derive. This class is named `ExtenderControl`. You can create your own extenders starting from this class; it is not recommended, though. Why is it so? There’s an easier and faster way that leverages the extensions available in the ACT library.

The following code shows the source code of the focus extender control. The sample extender adds to its target control a highlighting behavior that changes the appearance of the control when this gets focused:

```csharp
using AjaxControlToolkit;
...
namespace Core35
{
    [TargetControlType(typeof(Control))]
    [ClientScriptResource("Core35.FocusBehavior", "focusBehavior.js")]
    public class FocusExtender : AjaxControlToolkit.ExtenderControlBase
    {
        [ExtenderControlProperty]
        [RequiredProperty]
        public string HighlightCssClass
        {
            get { return GetPropertyValue("HighlightCssClass", ""); }
            set { SetPropertyValue("HighlightCssClass", value); }
        }

        [ExtenderControlProperty]
        public string NoHighlightCssClass
        {
            get { return GetPropertyValue("NoHighlightCssClass", ""); }
            set { SetPropertyValue("NoHighlightCssClass", value); }
        }
    }
}
```
The TargetControlType attribute indicates the type of controls this behavior can be attached to. The ClientScriptResource attribute indicates the name of the script class to inject in the client page and its source file. The base class is ExtenderControlBase, which is defined in the ACT library.

All that you do with managed code is define the set of properties that developers can customize on both the server and the client. Each property must be decorated with the ExtenderControlProperty attribute and, optionally, the RequiredProperty attribute.

The property is not directly responsible for the persistence of its assigned value. It is limited to getting and setting the value through the GetPropertyValue and SetPropertyValue methods of the base class. These stock methods take care of persistence.

The core part of an AJAX extender control is its JavaScript code. Here's the JavaScript code you need for the focus extender:

```
Type.registerNamespace('Core35');
Core35.FocusBehavior = function(element)
{
    Core35.FocusBehavior.initializeBase(this, [element]);
    this._highlightCssClass = null;
    this._nohighlightCssClass = null;
}
Core35.FocusBehavior.prototype =
{
    initialize : function()
    {
        Core35.FocusBehavior.callBaseMethod(this, 'initialize');
        this._onfocusHandler = Function.createDelegate(this, this._onFocus);
        this._onblurHandler = Function.createDelegate(this, this._onBlur);
        $addHandlers(this.get_element(),
        {
            'focus' : this._onFocus,
            'blur' : this._onBlur,
        }, this);
        this.get_element().className = this._nohighlightCssClass;
    },
    dispose : function()
    {
        $clearHandlers(this.get_element());
        Core35.FocusBehavior.callBaseMethod(this, 'dispose');
    },
    _onFocus : function(e)
    {
        if (this.get_element() && !this.get_element().disabled) {
            this.get_element().className = this._highlightCssClass;
        }
    },
    _onBlur : function(e)
    {
        if (this.get_element() && !this.get_element().disabled) {
            this.get_element().className = this._nohighlightCssClass;
        }
    };
```
get_highlightCssClass : function() {
    return this._highlightCssClass;
},
set_highlightCssClass : function(value) {
    if (this._highlightCssClass !== value) {
        this._highlightCssClass = value;
        this.raisePropertyChanged('highlightCssClass');
    }
},
get_nohighlightCssClass : function() {
    return this._nohighlightCssClass;
},
set_nohighlightCssClass : function(value) {
    if (this._nohighlightCssClass !== value) {
        this._nohighlightCssClass = value;
        this.raisePropertyChanged('nohighlightCssClass');
    }
}

// Optional descriptor for JSON serialization
Core35.FocusBehavior.descriptor = {
    properties: [
        {name: 'highlightCssClass', type: String},
        {name: 'nohighlightCssClass', type: String}
    ]
}

// Register the class as a type that inherits from Sys.UI.Control,
Core35.FocusBehavior.registerClass('Core35.FocusBehavior', Sys.UI.Behavior);

The script for the extender is centered around a couple of handlers for focus and blur DOM events. In the focus handler, the code sets the CSS class for the target element. In the blur handler, it resets the CSS class.

Tip The easiest way to create extenders is to use the facilities of the ACT. You run the VSI file in the toolkit download, set up the templates, and then click "Add AJAX Control Extender" from the Visual Studio 2008 Solution Explorer to obtain a scaffold for the extender and the behavior.

In a test page, all that you have to do is register the ACT assembly and the assembly that contains the focus extender and then go with the following code:

```html
<asp:TextBox ID="TextBox1" runat="server" EnableTheming="False" />
<act:FocusExtender ID="FocusExtender1" runat="server"
    TargetControlID="TextBox1"
    NoHighlightCssClass="LowLightTextBox"
    HighlightCssClass="HighLight" />
```

You can attach the focus extender behavior to virtually all ASP.NET controls and, for each control, you can specify the CSS class to use to render the control as highlighted and normal. Bear in mind that you need to disable theming to make sure that CSS styles you apply through the extender take precedence over any other style set via themes. Figure 19-8 shows the extender in action.
Extenders in the ACT

Table 19-11 lists the components available in the ACT. Note that the full name of the extender class contains an “Extender” suffix that I omitted in the table for brevity. So, for example, there will be no CollapsiblePanel component in the ACT assembly; you will find a CollapsiblePanelExtender control instead.

TABLE 19-11 Extenders in the ACT

<table>
<thead>
<tr>
<th>Control</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlwaysVisibleControl</td>
<td>Pins a control to a corner of the page, and keeps it floating over the page background as the user scrolls or resizes the page. You use this extender to make sure that, say, a given panel shows up at the top-left corner of the page regardless of the scroll position or the size of the browser window.</td>
</tr>
<tr>
<td>Animation</td>
<td>Provides a specialized framework for adding animation effects to controls hosted in ASP.NET pages. You associate client-side events of the target control with one or more of the predefined animation effects.</td>
</tr>
<tr>
<td>AutoComplete</td>
<td>Associated with a text box, provides a list of suggestions for the text to type in the field.</td>
</tr>
<tr>
<td>Calendar</td>
<td>Attached to a text box, the extender provides client-side date-picking functionality with customizable date format and pop-up control.</td>
</tr>
<tr>
<td>Control</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CascadingDropDown</td>
<td>Associated with a DropDownList control. This extender automatically populates the list with data retrieved from a Web service method. The nice thing about this extender is that you can create a hierarchy of drop-down lists and have the extender automatically populate child drop-down lists based on the current selections in any of the previous lists in the hierarchy, if any.</td>
</tr>
<tr>
<td>CollapsiblePanel</td>
<td>Adds collapsible sections to a Web page. This extender can be used only with panel controls—that is, with the ASP.NET Panel control or any class derived from it. You let the extender know which panel in the page acts as the header and which panel provides the contents that collapse and expand.</td>
</tr>
<tr>
<td>ConfirmButton</td>
<td>Associated with a button control. This extender adds a confirmation JavaScript dialog box to the click event of the button. The extender is supported on any class that implements the IButtonControl interface, including Button, LinkButton, and ImageButton.</td>
</tr>
<tr>
<td>DragPanel</td>
<td>Associated with panel controls. This extender adds drag-and-drop capabilities so that you can move the panel around the page. You can specify the contents to move as well as the handle that, if pressed, triggers the dragging operation.</td>
</tr>
<tr>
<td>DropDown</td>
<td>The extender provides a mouse-over link to open a drop-down panel.</td>
</tr>
<tr>
<td>DropShadow</td>
<td>Adds drop shadows to any control available on the page. With this extender, you can specify the opacity and width of the shadow.</td>
</tr>
<tr>
<td>DynamicPopulate</td>
<td>Updates the contents of a control with the result of a Web service or page method call.</td>
</tr>
<tr>
<td>FilteredTextBox</td>
<td>Lets users enter text in a TextBox control that matches a given set of valid characters.</td>
</tr>
<tr>
<td>HoverMenu</td>
<td>Displays the contents of an associated panel control when the mouse hovers next to a given control. You can associate this extender with any ASP.NET control. The extender works as a kind of specialized and extremely flexible ToolTip.</td>
</tr>
<tr>
<td>ListSearch</td>
<td>Enables users to search for an item in a list by typing some of the characters.</td>
</tr>
<tr>
<td>MaskedTextBox</td>
<td>Lets users enter text in a TextBox control according to a given input layout.</td>
</tr>
<tr>
<td>ModalPopup</td>
<td>Associated with a control that can fire a client-side onclick event (typically, buttons and hyperlinks), this extender implements a classic modal dialog box without using HTML dialog boxes. Basically, it displays the contents of a given panel and prevents the user from interacting with the rest of the page.</td>
</tr>
</tbody>
</table>
### Part IV ASP.NET AJAX Extensions

<table>
<thead>
<tr>
<th>Control</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MutuallyExclusiveCheckBox</td>
<td>Associated with CheckBox controls, this extender lets you define logical groups of check boxes so that users can select only one in each group.</td>
</tr>
<tr>
<td>NoBot</td>
<td>Applies some anti-bot techniques to input forms. Bots, or robot applications, are software applications that run automated tasks over the Internet. For example, bots are used to fill input forms and submit ad hoc values.</td>
</tr>
<tr>
<td>NumericUpDown</td>
<td>Associated with text box controls, this extender allows you to click automatically displayed buttons to enter the next/previous value in the field. It works with numbers, custom lists, and Web service methods.</td>
</tr>
<tr>
<td>PagingBulletedList</td>
<td>Associated with BulletedList controls, this extender groups all items bound to the list and organizes them in client-side sorted pages.</td>
</tr>
<tr>
<td>PasswordStrength</td>
<td>Associated with text box controls used to type a password, this extender provides visual feedback on the strength of the password being typed.</td>
</tr>
<tr>
<td>PopupControl</td>
<td>Transforms the contents of a given panel into a pop-up window without using HTML dialog boxes. You can associate this extender with any control that can fire any of the following client-side events: onfocus, onclick, and onkeydown.</td>
</tr>
<tr>
<td>ResizableControl</td>
<td>Attaches to any page element, and allows the user to resize the element using a handle placed at the lower-right corner of the control.</td>
</tr>
<tr>
<td>RoundedCorners</td>
<td>Adds a background panel to any ASP.NET control so that the control appears with rounded corners. The overall height of the original control changes slightly.</td>
</tr>
<tr>
<td>Slider</td>
<td>Extends a TextBox control with a slider user interface.</td>
</tr>
<tr>
<td>SlideShow</td>
<td>Associated with Image controls, it can be used to transition images automatically when hosted on a page.</td>
</tr>
<tr>
<td>TextBoxWatermark</td>
<td>Associated with TextBox controls. This extender adds sample or prompt text, called a “watermark,” that illustrates the type of text the user is expected to enter in the field. For example, the watermark might say, “Type your name here.” The watermark text disappears as soon as the user starts typing and reappears if the text box becomes empty.</td>
</tr>
<tr>
<td>ToggleButton</td>
<td>Associated with CheckBox controls. This extender enables you to use custom images to render the check buttons. You can use different images to indicate the selected and cleared states.</td>
</tr>
<tr>
<td>UpdatePanelAnimation</td>
<td>Plays animations during key steps of a partial update. You can use the extender to animate the page both while the panel is being updated and when the update has completed.</td>
</tr>
</tbody>
</table>
Control Description
ValidatorCallout Works on top of ASP.NET validators, and improves their user interface. In particular, the extender displays a yellow callout with the error message.

As you can probably guess, some extenders listed in the table require rich browser capabilities, whereas others are just a smart piece of JavaScript code attached to a block of markup elements. Note that all these features work in a cross-browser way. I'll return to some of the aforementioned extenders with code samples and more details in a moment. For more complete documentation, please refer to http://www.asp.net/ajax/ajaxcontroltoolkit/samples.

Controls in the ACT
Along with all the extenders listed in Table 19-11, the ACT also supplies a few traditional server controls with rich capabilities: the Accordion, Rating, ReorderList, and TabContainer controls.

The Accordion control allows you to provide multiple collapsible panes and display only one at a time. When the user clicks a new pane, the currently displayed pane is collapsed to leave room for the new one.

The Rating control provides an intuitive user interface to let users select the number of stars that represents their rating of a given subject. The control is the wrapped-up version of the user interface that several Web sites provide to let users rate published items.

A data-bound control, ReorderList, allows its child elements to be reordered on the client using drag-and-drop functionality. To move an item in the list, the user drags the item's handle up to its new position. At the end of the operation, the control posts back so that the new status of the data source can be recorded.

Finally, the TabContainer control is a purely client-side container of tabbed forms.

You use any of these controls in the same way you would use any native ASP.NET server controls.

Improving the User Interface with Input Extenders
In a perfect world where architects design applications in strict accordance with requirements and keeping the user's satisfaction in mind, each logical data type features its own set of input controls. In general, money, days of the week, ages, and quantities are all data represented with numbers. However, each of these logical data types might need a different user interface. The same can be said for URLs, paths, IP addresses, and dates.
Both in Windows and the Web, user interfaces are built by composing controls together. The original toolbox of controls, though, is not particularly rich in either Windows/WPF or ASP.NET. Input controls for Windows Forms are still based on the original windows in Win32. Input controls for Web pages are little more than wrappers for the HTML `<input>` element.

The need for effective input controls is even higher today, now that AJAX is taking root. The more work a page can do on the client, the better the overall experience is for the user. The user experience is widely improved because of saved postbacks but also because of the increased interactivity, dynamic data formatting, and validation that ad hoc input components can guarantee. As a result, input controls should be more interactive than ever. ACT extenders can help a lot.

**Motivation for Input Facilities**

As mentioned, in HTML there’s one primary element for accepting data—the `<input>` element. You use it for numbers, strings, dates, currency values, and so forth. So what if you need numeric input to be restricted to a specific range of values?

You can leave the user free of typing any values and then enforce this business rule through a server-side validation layer. Although server-side validation shouldn’t be avoided (for the safety of data), it would be even better if you could start yelling at wrong data as it’s entered into the input box.

JavaScript code is required to check values and ensure they comply with expected types and formats. In some cases, a specialized user interface also is required to make the user feel more comfortable. Here’s where some ad hoc input extenders fit in. Let’s start with the `Slider` and `NumericUpDown` extenders. Both force users to enter only numbers that fall in a given range.

**The Slider Extender**

The `Slider` extender hides its associated `TextBox` and replaces it with a graphical slider moving through discrete steps within a minimum and maximum interval. The underlying `TextBox` can always be set programmatically to any value you want. Note, though, that the assigned value must be compatible with the numeric range set via the slider; otherwise, the slider will silently fix it. Let’s consider the following example:

```xml
<act:SliderExtender runat="server" ID="SliderExtender1" 
    TargetControlID="txtYourIncome" 
    Minimum="0" 
    Maximum="200000" 
    Steps="41" 
    BoundControlID="lblIncomeAsText" />
```

---

© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.
The slider applies to a TextBox control named txtYourIncome and ensures it is visually assigned a value in the range 0 through 200,000. The Steps property indicates how many discrete steps should be provided. The preceding setting lets the slider jump by 5000 every time.

**Note** Why is Steps set to a weird 41 value, when a value of 40 would have probably looked better? Correctly, 40 is what you get by dividing 200,000 by 5000. However, the property Steps counts the number of steps ranging from 0 to 200000. You have to add one more tick to be able to select 0 first and then values every 5000.

If you programmatically set the slider on the server, the value is first converted to a number and then mapped to the nearest step value. If you pass in a string (that is, not a number), the slider ignores it and defaults to the initial value of the range. If you set the value of the underlying text box on the client via JavaScript, the value is correctly recognized over the next postback, but it is not immediately reflected by the user interface. To programmatically change the value in the slider from within the client, use the following code:

```javascript
王子(find("SliderExtender1")).set_Value(145678);
```

Why should you opt for find instead of get? As mentioned, the $get function is shorthand for document.getElementById and looks only for DOM elements. The $find function stands for Sys.Application.findComponent and applies to any component of the Microsoft AJAX Library that has been programmatically created.

The BoundControlID property refers to a DOM element that dynamically displays the current slider value. In most cases, you’ll use a or a Label control for this purpose. Internally, the slider distinguishes between and any other HTML elements. It sets the value property in the former case; otherwise, it uses the innerHTML property of the matching DOM element.

**Tip** The slider script hides the underlying TextBox. For this reason, it is recommended that you hide the text box declaratively using CSS styles to avoid any flashing when the page loads up. Finally, be aware that regular text boxes are displayed inline, whereas slider boxes are positioned through blocks.

Figure 19-9 shows a few input extenders in action in a sample page.
The **NumericUpDown** Extender

The NumericUpDown extender selects the next/previous value in a bound list of possible values. Despite what the name implies, NumericUpDown is not just a tool for specifying numeric quantities. The RefValues property allows you to list values to move through:

```xml
<act:NumericUpDownExtender ID="UpDown1" runat="server" Width="200" RefValues="None;1;2;3;4;5;More than 5;" TargetControlID="Children" />
```

**NumericUpDown** has the ability to retrieve reference values from a remote service. The service is invoked asynchronously each time the user clicks the up or down buttons. The service must be script-callable and include methods with the following signature:

```csharp
public int YourMethod(int current, string tag)
```

You configure the extender to use the remote service through the ServiceDownPath, ServiceDownMethod, ServiceUpPath, and ServiceUpMethod properties. Finally, the up and down buttons don’t have to be auto-generated via script. They can be buttons already defined in the page and referenced through the TargetButtonDownID and TargetButtonUpID properties.
The **FilteredTextBox Extender**

Preventing user mistakes involves controlling the input by filtering out undesired characters, invalid expressions, or data of the wrong type. Using the standard TextBox control, you enforce proper input by validating it on the server, where you can check whether the input string can be converted to a given data type. Extenders simply save you from writing the JavaScript code required to make checks on the client.

The **FilteredTextBox** extender prevents a user from entering invalid characters into a text box. It basically adds JavaScript that hooks up the keyboard activity and filters out undesired keystrokes. You can configure the extender to refuse certain characters or to ensure that only specified characters are accepted. You use the FilterMode property for this setting: it accepts only **ValidChars** and **InvalidChars** as its value:

```xml
<act:FilteredTextBoxExtender ID="Filtered1" runat="server" TargetControlID="YourAge" FilterMode="ValidChars" FilterType="Numbers" />
```

The **FilterType** property determines the type of filter to apply—only numbers, only lowercase or uppercase, and a custom set of characters. It should be noted that a TextBox with a watermark can’t be filtered to accept numeric values only. The filter layer, in fact, automatically clears any watermark text you set.

The **Calendar Extender**

Using the calendar extender, you make it virtually impossible for users to type anything other than a date. ASP.NET comes with a server **Calendar** control, but a calendar extender is really different, as it builds its user interface entirely on the client, works entirely on the client, and generates no postbacks at all as the user navigates to find the month and day. Furthermore, if a given browser doesn’t support JavaScript, the old text box is displayed.

```xml
<act:CalendarExtender ID="CalendarExtender1" runat="server" TargetControlID="Birth" Format="dd/MM/yyyy" />
```

The preceding code snippet is sufficient to display a popup calendar as the associated text box receives the focus. As an alternative, you can display the popup when the user clicks a page button. The ID of the button is set through the **PopupButtonID** property. The **Format** property indicates the format of the date as it will be written to the text box when the user dismisses the calendar popup. (See Figure 19-10.)
The Calendar extender is good for date-picking functionality; it is not as good for real calendaring functionality.

The MaskedEdit Extender

Would you really use the Calendar extender to pick a date that represents a birth date? It is an option and, all in all, it is one of the best options you have. But it is not the perfect choice. Why is that so? A date that represents a birth date would reasonably require you to navigate a few years back to find the right day. A popup calendar is great to pick close dates; it loses part of its appeal as it is employed to pick up just any date. To some extent, we're back to square one—using a text box to have users enter a date. This raises a number of new issues—involving formatting, locales, and separators.

When we start reasoning about data formatting and locales, Date is not the only critical data type. Currency, numbers, special strings such as URLs, disk paths, phone numbers, and e-mail addresses are all great examples of data types for which a specialized and masked input field is desirable. In ASP.NET, the MaskedEdit extender is a component that when attached to a TextBox control allows you to control input in a number of common scenarios.
You can use the MaskedEdit extender to enter numbers, dates, times, and date/times. The extender decides its output based on given culture settings. The following code snippet shows the typical way to use the extender with a text box that accepts a date:

```xml
<asp:TextBox runat="server" ID="TextBox1" />
<act:MaskedEditExtender ID="MaskedEditExtender1" runat="server"
TargetControlID="TextBox1"
Mask="99/99/9999"
MaskType="Date" />
```

You define an input mainly through two properties: Mask and MaskType. The full list of properties is shown in Table 19-12.

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AcceptAMPM</td>
<td>False</td>
<td>Boolean property, indicates whether an AM/PM symbol should be used when representing a time.</td>
</tr>
<tr>
<td>AcceptNegative</td>
<td>None</td>
<td>Indicates whether a negative sign (-) is required for negative values. Feasible values come from the MaskedEditShowSymbol enumeration: None, Left, Right.</td>
</tr>
<tr>
<td>AutoComplete</td>
<td>True</td>
<td>Boolean property, indicates whether empty mask characters not specified by the user must be automatically filled in.</td>
</tr>
<tr>
<td>AutoCompleteValue</td>
<td>&quot;&quot;</td>
<td>Indicates the default character to use when AutoComplete is enabled.</td>
</tr>
<tr>
<td>Century</td>
<td>1900</td>
<td>Indicates the century to use when a date mask has only two digits for the year.</td>
</tr>
<tr>
<td>ClearMaskOnLostFocus</td>
<td>True</td>
<td>Boolean property, indicates whether to remove the mask when the TextBox loses the input focus.</td>
</tr>
<tr>
<td>ClearTextOnInvalid</td>
<td>False</td>
<td>Boolean property, indicates whether to clear the TextBox when the user has entered invalid text.</td>
</tr>
<tr>
<td>ClipboardEnabled</td>
<td>True</td>
<td>Boolean property, indicates whether to allow copy/paste with the clipboard.</td>
</tr>
<tr>
<td>ClipboardText</td>
<td>&quot;&quot;</td>
<td>Indicates the prompt text to use when a clipboard paste is performed.</td>
</tr>
<tr>
<td>CultureName</td>
<td>&quot;&quot;</td>
<td>Gets and sets the name of the culture to use.</td>
</tr>
<tr>
<td>DisplayMoney</td>
<td>None</td>
<td>Indicates whether the currency symbol is displayed. Feasible values come from the MaskedEditShowSymbol enumeration: None, Left, Right.</td>
</tr>
<tr>
<td>ErrorTooltipCssClass</td>
<td>&quot;&quot;</td>
<td>Gets and sets the CSS class for the Tool Tip message.</td>
</tr>
<tr>
<td>ErrorTooltipEnabled</td>
<td>False</td>
<td>Boolean property, indicates whether to show a Tool Tip message when the mouse hovers over a TextBox with invalid content.</td>
</tr>
</tbody>
</table>
### Property Default Description

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filtered</td>
<td>&quot;&quot;</td>
<td>Gets and sets the list of valid characters for the mask type when the &quot;C&quot; placeholder is specified</td>
</tr>
<tr>
<td>InputDirection</td>
<td>LeftToRight</td>
<td>Indicates the text input direction. Feasible values come from the MaskedEditInputDirection enumeration: LeftToRight, RightToLeft.</td>
</tr>
<tr>
<td>Mask</td>
<td>&quot;&quot;</td>
<td>Specifies the mask of characters that is acceptable to the user.</td>
</tr>
<tr>
<td>MaskType</td>
<td>&quot;&quot;</td>
<td>Indicates the mask type using any of the values defined by the MaskedEditType enumeration.</td>
</tr>
<tr>
<td>MessageValidatorTip</td>
<td>True</td>
<td>Boolean property, indicates whether a help message is displayed as the user types in the TextBox.</td>
</tr>
<tr>
<td>OnBlurCssNegative</td>
<td>&quot;&quot;</td>
<td>Gets and sets the CSS class used when the TextBox loses the input focus and contains a negative value.</td>
</tr>
<tr>
<td>OnFocusCssClass</td>
<td>&quot;&quot;</td>
<td>Gets and sets the CSS class used when the TextBox receives the input focus.</td>
</tr>
<tr>
<td>OnFocusCssNegative</td>
<td>&quot;&quot;</td>
<td>Gets and sets the CSS class used when the TextBox gets the input focus and contains a negative value.</td>
</tr>
<tr>
<td>OnInvalidCssClass</td>
<td>&quot;&quot;</td>
<td>Gets and sets the CSS class used when the text is not valid.</td>
</tr>
<tr>
<td>PromptCharacter</td>
<td>_</td>
<td>Gets and sets the prompt character being used for unspecified mask characters</td>
</tr>
<tr>
<td>UserDateFormat</td>
<td>None</td>
<td>Indicates a particular date format. Feasible values are defined by the MaskedEditUserDateFormat enumeration.</td>
</tr>
<tr>
<td>UserTimeFormat</td>
<td>None</td>
<td>Indicates a particular time format. Feasible values are defined by the MaskedEditUserTimeFormat enumeration.</td>
</tr>
</tbody>
</table>

The **MaskType** property selects a general pattern from a predefined list—the **MaskedEditType** enumeration:

```csharp
public enum MaskedEditType
{
    None,
    Date,
    Number,
    Time,
    DateTime
}
```

By selecting a mask type, you inform the extender that the target control is going to accept a number, a date, a time, or both. The **Mask** property (of string type) indicates the physical sequence of characters that form a valid input for the text box. For example, "5/4/08" and "04-05-2008" are both valid dates, but they use different input masks.
To build a mask, you use a few predefined symbols as placeholders. The list of supported symbols is in Table 19-13. For example, the “999,999.99” mask makes your code accept a number with a decimal separator and, at most, a one thousand separator.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Indicates a numeric character</td>
</tr>
<tr>
<td>L</td>
<td>Indicates a letter</td>
</tr>
<tr>
<td>$</td>
<td>Indicates a letter or a blank</td>
</tr>
<tr>
<td>C</td>
<td>Indicates a custom case-sensitive character as defined by the Filtered property</td>
</tr>
<tr>
<td>A</td>
<td>Indicates a letter or a custom character as defined by the Filtered property</td>
</tr>
<tr>
<td>N</td>
<td>Indicates a numeric or custom character as defined by the Filtered property</td>
</tr>
<tr>
<td>?</td>
<td>Indicates any character</td>
</tr>
<tr>
<td>/</td>
<td>Indicates a date separator according to the current culture</td>
</tr>
<tr>
<td>:</td>
<td>Indicates a time separator according to the current culture</td>
</tr>
<tr>
<td>.</td>
<td>Indicates a decimal separator according to the current culture</td>
</tr>
<tr>
<td>,</td>
<td>Indicates a thousand separator according to the current culture</td>
</tr>
<tr>
<td>\</td>
<td>Indicates an escape character</td>
</tr>
<tr>
<td>{</td>
<td>Indicates the initial delimiter for repetition of masks</td>
</tr>
<tr>
<td>}</td>
<td>Indicates the final delimiter for repetition of masks</td>
</tr>
</tbody>
</table>

The appearance of the currency symbol is controlled by the DisplayMoney property, and each character to type is represented by a prompt. The default prompt is the underscore, but you can change it via the PromptCharacter property.

For dates, you can also use extra properties such as AcceptAMPM, Century, and even a custom user format in addition to the predefined formats defined by the MaskedEditUserDateFormat enumeration:

```csharp
public enum MaskedEditUserDateFormat
{
    None,
    DayMonthYear,
    DayYearMonth,
    MonthDayYear,
    MonthYearDay,
    YearDayMonth,
    YearMonthDay
}
```

Many of the settings that influence the formatting applied by the MaskedEdit extender descend from the current culture. The CultureName property indicates the culture to
apply. Note that this setting overrides the culture setting defined for the page through the UICulture attribute in the @Page directive.

While the masked extender provides dynamic formatting capabilities, an additional component—the masked validator—ensures that any text typed can be successfully parsed back to the expected type:

```xml
<act:MaskedEditValidator ID="MaskedEditValidator1" runat="server"
    ControlExtender="MaskedEditExtender1"
    ControlToValidate="TextBox1"
    IsValidEmpty="False"
    EmptyValueMessage="Number is required."
    InvalidValueMessage="Number is invalid" />
```

The MaskedEditValidator control is a custom validator that you optionally attach to the MaskedEdit extender so that the content of the edited TextBox is carefully verified. The validator ensures that the text matches the mask. The validator performs server and client validation, and it can be associated with a validation group, just like any other ASP.NET validator control.

The Text property of a masked TextBox returns formatted text. For a date, the property returns something like "05/04/2008"; for a number input field, the property returns text like "3,500.00". The currency symbol is not included in the Text property, even though it is shown to the user in the page.

How can you parse the value returned by Text into the logical data type—be it a date or a decimal? You can use the static Parse method on the DateTime and Decimal types, but you must pay due attention to the culture you use. For example, "05/04/2008" can be either the 4th of May (US culture) or the 5th of April (European culture).

The issue is that there’s no guaranteed matching between the culture used by the input page and the server page. The risk is that users type the date according to the European culture and have it processed on the server as a US culture data. Worse yet, the 3500 value entered using, say, Italian decimal and thousand separators in a numeric text box ("3.500,00") might throw an exception because the parser of the Decimal type defaults to the US culture where commas and the dot are reversed. You have to work around these issues programmatically.

The key fact to remember is that extenders default to the en-us culture unless the CultureName property is explicitly set. On the server, instead, the system defaults to the value of the UICulture property on the Page class. In your code-behind class, you first obtain a CultureInfo object that reflects the culture used for the user interface of the MaskedEdit extender. You can proceed as shown here:

```csharp
string culture = "en-us";
if (!String.IsNullOrEmpty(MaskedEditExtender1.CultureName))
    culture = MaskedEditExtender1.CultureName;
CultureInfo info = new CultureInfo(culture);
```
Next, you call the `Parse` method, specifying a format provider based on the selected culture:

```csharp
NumberFormatInfo numberInfo = info.NumberFormat;
DateTimeFormatInfo dateInfo = info.DateTimeFormat;
DateTime when = DateTime.Parse(txtBirthDate.Text, dateInfo);
decimal amount = Decimal.Parse(txtAmount.Text, numberInfo);
```

Figure 19-11 shows the behavior of a page that uses the it-IT culture in the masked editor but an en-US culture on the server. The preceding code snippet performs the trick of transforming dates correctly.

**FIGURE 19-11** The MaskedEdit extender in action

The **AutoComplete Extender**

Auto-completion consists of the program’s ability to predict the word the user is typing from the first few characters he or she has just entered. In Internet Explorer, for example, auto-completion keeps track of any text you type in the address bar and form fields and offers suggestions whenever you’re typing again in a similar control. This feature is entirely browser-led and can just be turned on and off for `<input>` and `<form>` tags by setting the `autocomplete` attribute to `off`. Note, though, that the `autocomplete` attribute is not a standard HTML attribute, although today nearly all browsers do recognize and support it.
The AutoComplete extender in the ACT provides a similar behavior for TextBox controls, but it makes the developer responsible for all the logic that provides possible words to the user. The extender creates a drop-down panel, much like a drop-down list, and positions it right at the bottom of the text box. Here’s how to associate an auto-complete extender with a text box:

```xml
<act:AutoCompleteExtender runat="server" ID="AutoComplete1"
    TargetControlID="TextBox1"
    MinimumPrefixLength="3"
    ServicePath="Suggestions.asmx"
    ServiceMethod="GetSuggestions" />
```

The extender is bound to a Web service or WCF service that actually provides the words to populate the drop-down list. The `MinimumPrefixLength` property instructs the control when to place a call to the Web service—for example, only when the user has typed at least the specified number of characters. The text already typed in will be used as input for the specified Web service method. The response is used to populate the drop-down list.

The `EnableCaching` Boolean property can also be turned on. If you do so, typing the same prefix more than once results in a single call to the Web service. Furthermore, depending on the way the Web service retrieves its data, you also can enable caching on the server to save some extra roundtrips to a database or another remote data store. Table 19-14 shows the full list of properties supported by the extender.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Animations</td>
<td>Sets animations to be played when the flyout is shown and hidden.</td>
</tr>
<tr>
<td>CompletionInterval</td>
<td>Gets and sets the number of milliseconds after which the extender gets suggestions using the bound Web service.</td>
</tr>
<tr>
<td>CompletionListCssClass</td>
<td>Indicates the CSS class used to style the completion list flyout.</td>
</tr>
<tr>
<td>CompletionListHighlightedItemCssClass</td>
<td>Indicates the CSS class used to style the highlighted item in the completion list flyout.</td>
</tr>
<tr>
<td>CompletionListItemCssClass</td>
<td>Indicates the CSS class used to style the item in the completion list flyout.</td>
</tr>
<tr>
<td>CompletionSetCount</td>
<td>Gets and sets the number of suggestions to get from the bound Web service. Default is 10.</td>
</tr>
<tr>
<td>ContextKey</td>
<td>String property, indicates any page or user-specific information to pass to the bound Web service.</td>
</tr>
<tr>
<td>DelimiterCharacters</td>
<td>Indicates one or more characters that the extender will use to tokenize the text-box content. The Web service will then use the last of these tokens to provide suggestions. Not set by default.</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>EnableCaching</td>
<td>Boolean property, indicates whether client-side caching is enabled. True by default.</td>
</tr>
<tr>
<td>FirstRowSelected</td>
<td>Boolean property, indicates whether the first option in the list will be automatically selected. False by default.</td>
</tr>
<tr>
<td>MinimumPrefixLength</td>
<td>Gets and sets the minimum number of characters in the text-box buffer that trigger the bound Web service. Default is 3.</td>
</tr>
<tr>
<td>ServiceMethod</td>
<td>Gets and sets the name of the method to invoke on the bound Web service.</td>
</tr>
<tr>
<td>ServicePath</td>
<td>Gets and sets the URL of the bound Web service.</td>
</tr>
<tr>
<td>UseContextKey</td>
<td>Boolean property, indicates whether the value of the ContextKey property should be used, if specified. False by default.</td>
</tr>
</tbody>
</table>

A Web service that works with the AutoComplete extender is an ASP.NET AJAX script service. It looks nearly the same as a regular ASP.NET Web service, except that its class must be decorated with the ScriptService attribute. If you employ a Web service that lacks the attribute, each request to the associated ASMX endpoint originates an HTTP 500 error.

```csharp
[ScriptService]
public class SuggestionService : WebService
{
    
}
```

The name of any public method on the class that is flagged with the WebMethod attribute can be successfully assigned to the ServiceMethod property of the extender. A method that provides suggestions must have the following signature:

```csharp
[WebMethod]
public string[] GetCustomerIDs(string prefixText, int count)
```

The first argument is the prefix text to generate suggestions. It matches the current content of the text box and its length is not smaller than the value of the MinimumPrefixLength property. The count parameter indicates how many suggestions are to be provided. The value of the count parameter comes from the value of the CompletionSetCount property. The return value is always packed as an array of strings. Because of the ScriptService attribute, any communication between the server and client occurs through JSON strings.

You can leverage any supported attributes on Web service methods that will make the call go faster. For example, the CacheDuration attribute on the WebMethod attribute forces the service to cache internally for the specified duration the response of the method call. Likewise, you can enable session state if it's strictly required by the logic the method implements. See Figure 19-12.
We'll return to the topic of script services and WCF services in the next chapter and discuss the full source for this example.

Adding Safe Popup Capabilities to Web Pages

Dialog boxes are plain windows and can work with or without modality. A modal window is a window that disables entirely any visual elements underneath it. The user can't take any action on the current application until he or she dismisses the modal window. The most popular Web counterpart of dialog boxes are popups.

The internal object model of most Web browsers already provides a native API to manage popups, and it supplies methods to open and close such windows with and without modality. Largely abused by some Web sites, popups are blocked by most client browsers and are no longer a valid option to drill down information over the Web. However, this doesn't mean that modal dialog boxes are definitely banned from Web applications.

It is essential to come up with a different implementation of dialog boxes that is as effective as modal popups and totally hassle-free for end users. The problem with classic popups is that they are just additional browser windows adorned with a different set of styles and fully controlled via script. A truly modal Web window, instead, provides you with the same modal
effect of classic popups, but it leverages the page's object model rather than the browser's object model.

In the ACT, you find a number of extenders to provide popup functionalities. The most compelling is the ModalPopup extender.

The ModalPopup Extender

The ModalPopup extender adds modality to a piece of markup—typically, a panel. Bound to a button control, it pops up the specified panel and disables the underlying page. Any clicking on anything other than the elements in the topmost panel is lost and never reaches the intended target. ModalPopup performs a smart trick by adding an invisible `<div>` tag to cover the entire browser window. This layer swallows any user action and stops it from reaching underlying controls. With clever CSS coding, you can add some nice effects, such as graying out anything underneath the top-most panel, as shown in Figure 19-13.

The modal popup extender just takes the markup generated by a server-side ASP.NET panel and shows and hides it as the user clicks on a linked HTML element. Initially styled to be hidden, the markup used for the dialog box is downloaded on the client when the host page is loaded and then shown and hidden on demand.

FIGURE 19-13 The ModalPopup extender in action
Part IV  ASP.NET AJAX Extensions

You start by defining a panel to provide the user interface, and then you add a button control to trigger the display of the dialog box:

```xml
<asp:Button runat="server" ID="btnViewMore" Text="More" />
<asp:Panel runat="server" ID="pnlViewCustomer">
    <div style="margin:10px">
        <h1>The service is not available in <span id="lblCountry"></span>.</h1>
        <asp:Button runat="server" ID="viewBox_OK" Text="OK" />
    </div>
</asp:Panel>
```

Next, you set up the extender and specify the target control ID and the popup control ID:

```xml
<act:ModalPopupExtender ID="ModalPopupExtender1" runat="server"
    TargetControlID="btnViewMore"
    PopupControlID="pnlViewCustomer"
    BackgroundCssClass="modalBackground"
    OkControlID="viewBox_OK"
    OnOkScript="allSet()" />
```

The target control ID of a modal popup extender is the ID of the server control that, when clicked, causes the dialog box to pop up. The popup control ID is the ID of the server control that provides the content for the dialog box.

The `OkControlID` property allows you to identify a button control in the popup panel to be used to dismiss the panel with an OK answer. When the panel is dismissed using the OK button, the associated `OnOkScript` JavaScript function, if any is present, is executed:

```javascript
function allSet()
{
    ...
}
</script>
```

The content of the popup panel can be initialized before display on both the client and the server. On the client, you register a handler for the `showing` event:

```javascript
function pageLoad(sender, args)
{
    $find("ModalPopupExtender1").add_showing(onModalShowing);
}
function onModalShowing(sender, args)
{
    $get("pnlViewCustomer").style.backgroundColor = "yellow";
}
```
On the server, you do it in the code associated with the Click event of the trigger button. However, in this case the popup panel must be wrapped in an UpdatePanel and brought up programmatically from the server:

```csharp
protected void btnEditText_Click(object sender, EventArgs e)
{
    // Initialize the controls in the panel used as the UI of the dialog box
    InitDialog();

    // The panel markup has already been served to the page. To edit it,
    // you need to wrap the panel's content in an UpdatePanel region and
    // update the panel once you make any changes
    popupPanel.Update();

    // Inject the script to show the dialog as the page is loaded in the browser.
    ModalPopupExtender1.Show();
}
```

There's another trick that contributes to making this code work. The ModalPopup extender is bound to an invisible button so that it can never be brought up via the user interface. If you bind the popup to a visible push button, the Click server event will never be fired and you have no way to initialize the control in the popup panel. For more information and details on Web dialog boxes, you might want to read the March 2008 installment of my "Cutting Edge" column in MSDN Magazine.

The **PopupControl Extender**

The PopupControl extender differs from ModalPopup because it can be dismissed by simply clicking outside. The PopupControl extender can be attached to any HTML element that fires the onclick, onfocus, or onkeydown events. The ultimate goal of the extender is to display a pop-up window that shows additional content, such as a calendar on a text box in which the user is expected to enter a date. The contents of the pop-up panel are expressed through a Panel control, and they can contain ASP.NET server controls as well as static text and HTML elements:

```html
<asp:panel runat="server" ID="Panel1">
...
</asp:panel>
<act:PopupControlExtender ID="PopupExtender1" runat="server"
    TargetControlID="txtInvoiceDate"
    PopupControlID="Panel1"
    Position="Bottom" />
```
The TargetControlID property points to the control that triggers the popup, whereas PopupControlID indicates the panel to display. The Position property sets the position of the panel—either at the top, left, right, or bottom of the parent control. Additional properties are OffsetX and OffsetY, which indicate the number of pixels to offset the popup from its position, as well as CommitProperty and CommitScript, which can be used to assign values to the target control.

The pop-up window will probably contain some interactive controls and post back. For this reason, you might want to insert it within an UpdatePanel control so that it can perform server-side tasks without refreshing the whole page. Typically, the popup will be dismissed after a postback—for example, after the user has selected a date in a Calendar control. The Calendar control in this case fires the SelectionChanged event on the server:

```csharp
protected void Calendar1_SelectionChanged(object sender, EventArgs e)
{
    PopupExtender1.Commit(Calendar1.SelectedDate.ToShortDateString());
}
```

The Commit method sets the default property of the associated control to the specified value. If you want to control which (nondefault) property is set on the target when the popup is dismissed, use the CommitProperty property. Likewise, you use the CommitScript property to indicate the JavaScript function to execute on the client after setting the result of the popup.

### Note
An extender can’t be placed in an UpdatePanel that is different than the control it extends. If the extended control is incorporated in an UpdatePanel, the extender should also be placed in the updatable panel. If you miss this, you get a runtime exception.

The **HoverMenu Extender**

The HoverMenu extender is similar to the PopupControl extender and can be associated with any ASP.NET control. Both extenders display a pop-up panel to display additional content, but they do it for different events. The HoverMenu, in particular, pops up its panel when the user moves the mouse cursor over the target control. The panel can be displayed at a position specified by the developer. It can be at the left, right, top, or bottom of the target control. In addition, the control can be given an optional CSS style so that it looks like it is in a highlighted state. (See Figure 19-14.)
The HoverMenu extender is good for implementing an auto-display context menu for virtually every ASP.NET control instance and for providing tips to fill in some input fields. In Figure 19-14, for example, when the user hovers the cursor over the text box, a list of suggestions appears to simplify the work:

```xml
<asp:TextBox ID="TextBox1" runat="server" />
<asp:Panel ID="Panel1" runat="server" CssClass="popupMenu">
    <asp:RadioButtonList ID="RadioButtonList1" runat="server" AutoPostBack="true"
        OnSelectedIndexChanged="RadioButtonList1_SelectedIndexChanged">
        <asp:ListItem Text="Dino Esposito"></asp:ListItem>
        <asp:ListItem Text="Nancy Davolio"></asp:ListItem>
        <asp:ListItem Text="Andrew Fuller"></asp:ListItem>
        <asp:ListItem Value="" Text="None of the above"></asp:ListItem>
    </asp:RadioButtonList>
</asp:Panel>
<act:HoverMenuExtender ID="HoverMenu1" runat="server"
    TargetControlID="TextBox1"
    HoverCssClass="hoverPopupMenu"
    PopupControlID="Panel1"
    PopupPosition="Right" />
```

The Panel1 control defines a list of radio buttons, each containing a suggestion for filling the text box. The HoverMenu extender targets the text box control and defines Panel1 as its dynamic pop-up panel. The PopupPosition property indicates the position of the panel with respect to the target control. Likewise, other properties not shown in the previous example code, such as OffsetX and OffsetY, define the desired offset of the panel. The PopDelay sets the time (in milliseconds) to pass between the mouse movement and the display of the
panel. The HoverCssClass can optionally be used to give the text box a different style when the hover menu is on. It is interesting to look at the CSS class associated with the panel:

```css
_popupMenu
{
    position:absolute;
    visibility:hidden;
    background-color:#F5F7F8;
}
_hoverPopupMenu
{
    background-color:yellow;
}
```

Just as for the PopupControl extender, to take full advantage of the HoverMenu extender you need to place extended controls inside of an UpdatePanel control. In this way, whenever the user clicks a radio button, the panel posts back asynchronously and fires the SelectedIndexChanged event on the server:

```csharp
void RadioButtonList1_SelectedIndexChanged(object sender, EventArgs e)
{
    TextBox1.Text = RadioButtonList1.SelectedValue;
}
```

The server-side event handler will then just update the text in the text box, as shown in Figure 19-14.

### The TabContainer Control

Multiple views are a common feature in most pages. They group information in tabs and let users click to display only a portion of the information available. In ASP.NET, the MultiView control provides an effective shortcut to this feature. But it requires a postback to update the page when the user selects a new tab. In the ACT, the TabContainer control provides a free AJAX version of the multiview control.

The TabContainer control includes a list of child TabPanel objects you can access programmatically via the Tabs collection. You add one `<TabPanel>` tag for each desired tab and configure it at will. Here's an example:

```xml
<act:TabContainer runat="server" ID="TabContainer1">
    <act:TabPanel runat="server" ID="TabPanel1" HeaderText="Your Tab">
        <ContentTemplate>
            <h3>Some text here</h3>
        </ContentTemplate>
    </act:TabPanel>
    ...
</act:TabContainer>
```
All tabs are given the same size, and you can control the size designation through the Width and Height properties of the container. The height you set refers to the body of tags and doesn't include the header.

You can add some script code to run when the user selects a new tab. You can wrap up all the code in a page-level JavaScript function and bind the name of the function to the OnClientActiveTabChanged property of the tab container. The following code writes the name of the currently selected tab to a page element (originally, an ASP.NET Label control) named CurrentTab:

```javascript
<script type="text/javascript">
    function ActiveTabChanged(sender, e) {
        var tab = $get('<%=CurrentTab.ClientID%>');
        tab.innerHTML = sender.get_activeTab().get_headerText();
    }
</script>
```

Note the usage of code blocks in JavaScript. In this way, the client ID of the label is merged in the script regardless of whether the page is a regular page or a content page (with a hierarchy of parent controls and naming containers). Figure 19-15 shows the control in action.

![Figure 19-15 The TabContainer control in action](image-url)
Conclusion

Partial rendering provides an excellent compromise between the need to implement asynchronous and out-of-band functionality and the desire to use the same familiar ASP.NET application model. As you've seen in this chapter, any ASP.NET page can be easily transformed into an ASP.NET AJAX page. You divide the original page into regions and assign each markup region to a distinct UpdatePanel control. From that point on, each updatable region can be refreshed individually through independent and asynchronous calls that do not affect the rest of the page. The current page remains up and active while regions are updated.

Partial rendering doesn't embody the loudly demanded change of paradigm for Web applications. It is limited to making the current Web model more efficient and effective by silently adding more script code and using this script code to hook up some standard browser procedures.

Partial rendering is the quickest way to AJAX and a profitable one too, as long as you are aware of its inherent limitations and know when to choose differently. A different aspect that AJAX-powered pages have when compared to classic ASP.NET pages is the use of more responsive and interactive controls.

No matter how many controls you have in your arsenal, you'll likely always be lacking just the one that is crucial for your current work. That's why the extensibility model of ASP.NET has been so successful over the years, and that's why so many component vendors crowd the market with excellent product offerings.

Anyway, always deriving new controls from existing ones might not necessarily be a wise strategy. A new control is required for a significant piece of server and client code that can be used to back up a good chunk of user interface. If you only need to filter the values in a text box, a custom text box control is hardly the best option. But until the arrival of extenders and the AJAX Control Toolkit, there was no other way out.

With control extenders, you define the concept of a “behavior” and work with it as a distinct entity, set apart from classic server controls. Extenders are server controls, but they work on top of bound controls and improve their overall capabilities by adding a new behavior.

Partial rendering and extenders in the ACT offer a powerful framework for authoring richer pages today, and it requires only limited new skills. True AJAX, though, requires an architectural switch. An AJAX application is articulated in two layers—the front end and back end. For the front end, you mostly have JavaScript and Silverlight in the near future. (I'll cover Silverlight in Chapter 21.) For the back end, you need services. And services are just the topic of the next chapter.
Just the Facts

- AJAX is a term that collectively refers to a set of development components, tools, and techniques for creating highly interactive Web applications that give users an overall better experience.

- AJAX pages work by making out-of-band calls to the Web server using the XMLHttpRequest object—a component designed to perform one key operation: sending an HTTP request either synchronously or asynchronously.

- AJAX raises the need for a more powerful JavaScript language. Entirely written in standard JavaScript, the Microsoft AJAX Library delivers a richer environment in which to write client-side code with a strong sense of object-orientation.

- Any ASP.NET AJAX page needs a reference to one ScriptManager control. The ScriptManager control manages and delivers script resources and enables partial-page rendering and Web service and page method calls.

- Partial rendering divides the page into independent regions, each of which controls its own postbacks and refreshes without causing a full-page update.

- You can safely use UpdatePanel controls from within master pages and user controls. UpdatePanel controls can be nested too.

- The mechanics of asynchronous postbacks keeps the displayed page up and running all the time. Subsequently, users must be prevented from clicking anywhere so that they don’t start new (and unwanted) operations.

- Partial rendering is only one possible way to approach AJAX. It preserves most of your current investments and is relatively cheap to implement. Partial rendering adds AJAX capabilities to your pages but doesn’t change its basic architecture.

- The AJAX Control Toolkit is a shared-source library of Web controls specifically designed to enhance ASP.NET control. It is not included in the ASP.NET 3.5 platform and should be downloaded separately.

- An ACT extender control represents a logical behavior that can be attached to one or more control types to extend their base capabilities. Extenders decouple controls from behaviors and make it possible to extend existing controls with new behaviors.