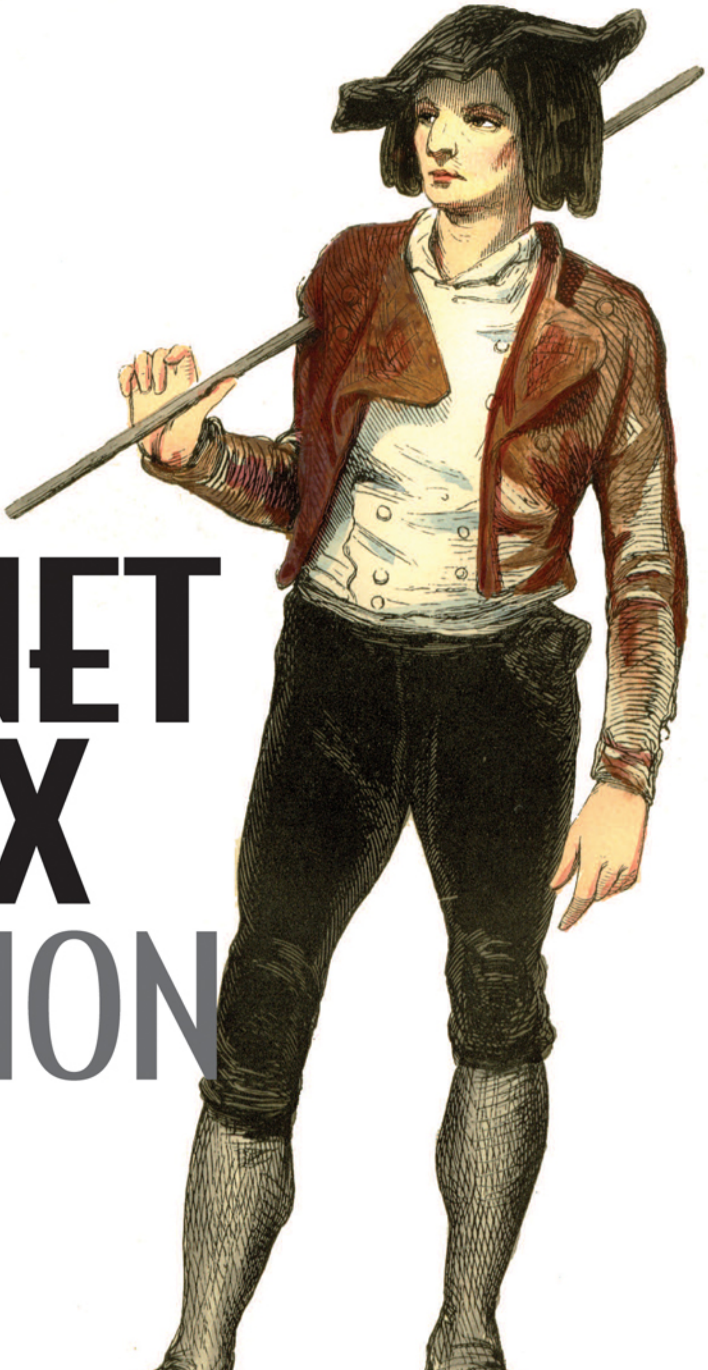
Alessandro Gallo
David Barkol
Rama Krishna Vavilala

ASP.NET AJAX IN ACTION



SAMPLE CHAPTER

MANNING

ASP.NET Ajax in Action
by Alessandro Gallo
David Barkol
and Rama Krishna Vavilala

Chapter 2

brief contents

2

First steps with the Microsoft Ajax Library

In this chapter:

- Overview of the Microsoft Ajax Library
- The Application model
- The abstraction API for working with the DOM
- JavaScript extensions

In the age of Ajax programming, web developers need to be more JavaScript proficient than ever. You must accomplish a long list of tasks in an Ajax-enabled page and coordinate activities on the client side. For example, you need the ability to access server resources, process the results quickly, and maintain smooth webpage interactivity. The need for programming patterns that build robust and maintainable code is also on the rise. In a nutshell, a consistent client-side programming environment that works on all modern browsers is essential.

This chapter is the first one dedicated to the Microsoft Ajax Library, which is written on top of JavaScript and constitutes the client portion of the ASP.NET AJAX framework. In the tour of the basic framework components in chapter 1, you began to write code using the library's syntax. This chapter will provide more examples and give you a comprehensive overview of the library's features.

2.1 A quick overview of the library

The Microsoft Ajax Library provides a rich set of tools for managing nearly every aspect of client development. The library isn't just a simple framework for sending asynchronous requests using the XMLHttpRequest object. Instead, one of its main goals is to bring to the client side many coding patterns familiar to .NET developers. Such .NET flavors include the possibility of exposing multicast events in JavaScript objects and leveraging a component model on the client side. The library also enhances the JavaScript language's type system and lets you write client code using object-oriented constructs like *classes* and *interfaces*. In addition, you can easily access local web services using JavaScript and deal with the ASP.NET application services, such as membership and profile, from the client side. Nonetheless, this is just a taste of the goodies provided by the library.

2.1.1 Library features

The Microsoft Ajax Library is rich in features, which we've grouped into logical categories. Because we can't explore all of them in a single chapter, the following list shows how the features are distributed in the book's various chapters:

- *Application model*—When you enable the Microsoft Ajax Library in a web page, an Application object is created at runtime. In section 2.2, you'll discover that this object takes care of managing the client lifecycle of a web page, in a manner similar to what the Page object does on the server side. The Application object hosts all the client components instantiated in the web page and is responsible for disposing them when the page is unloaded by the browser.

- *Components*—The Microsoft Ajax Library brings to the client side a component model similar to that provided by the .NET framework. You can create *visual* or *nonvisual* components, depending on whether they provide a UI. In chapter 8, which is entirely dedicated to the client component model, you'll see also how visual components can be associated with Document Object Model (DOM) elements.
- *JavaScript extensions*—As you'll see in chapter 3, the library leverages the object model provided by JavaScript by introducing an enhanced type system that supports reflection and object-oriented constructs like classes, interfaces, and enumerations. In addition, the built-in JavaScript objects have been extended to support methods commonly found in .NET classes.
- *Compatibility*—Section 2.3 covers the *abstraction API*, which is a set of client methods for writing code that runs smoothly in all the supported browsers. This API abstracts common operations performed on DOM elements, such as handling events and dealing with CSS and positioning.
- *Ajax*—The library isn't exempt from providing a communication layer for sending asynchronous HTTP requests using the XMLHttpRequest object. Chapter 5 is entirely dedicated to the networking layer.
- *Application services*—By using the Microsoft Ajax Library, ASP.NET developers can deal with the authentication, membership, and profile providers on the client side. You can interact with the providers through proxy services by writing JavaScript code in the page.
- *Partial rendering*—The UpdatePanel control, introduced in chapter 1, makes it possible to update portions of the page's layout without refreshing the whole UI. This mechanism, called *partial rendering*, is leveraged on the client side by an object called the PageRequestManager. In chapter 7, when we discuss what happens under the hood of the UpdatePanel, we'll explain how the Microsoft Ajax Library participates in the partial-rendering mechanism.

Some of the features in the ASP.NET Futures package are interesting, and we decided to cover them in this book. Chapter 11 is dedicated to XML Script, a declarative language—similar to the ASP.NET markup language—used to create JavaScript objects without writing a single line of JavaScript code. Chapter 12 talks about how to perform drag and drop using the Microsoft Ajax Library.

Before proceeding, let's establish a couple of conventions relative to the terminology we'll use throughout the book. JavaScript is an object-oriented language; but, unlike C# or VB.NET, it doesn't support constructs like classes and namespaces. Nonetheless, as you'll see in chapter 3, you can manipulate JavaScript functions in interesting ways to simulate these and other object-oriented constructs. For this reason, when talking about client JavaScript code, we often borrow terms such as *class*, *method*, *interface*, and others from the common terminology used in object-oriented programming. For example, when we talk about a *client class*, we're referring to a class created in JavaScript with the Microsoft Ajax Library.

We're ready to start exploring the library. The first step is learning how to load the library's script files in a web page.

2.1.2 *Ajax-enabling an ASP.NET page*

The Microsoft Ajax Library is organized in client classes contained in namespaces. The root namespace is called `Sys`. The other namespaces are children of the root namespace. Table 2.1 lists the namespaces defined in the library and the type of classes that they contain.

Table 2.1 Namespaces defined in the Microsoft Ajax Library. The root namespace defined by the library is called `Sys`

Namespace	Content
`Sys`	Base runtime classes, Application object
`Sys.Net`	Classes that belong to the networking layer
`Sys.UI`	Classes for working with components and the DOM
`Sys.Services`	Classes for accessing ASP.NET services like profile, membership, and authentication
`Sys.Serialization`	Classes for JSON serialization/deserialization
`Sys.WebForms`	Classes related to partial page rendering

The Microsoft Ajax Library consists of multiple JavaScript files loaded by the browser at runtime. These files are embedded as web resources in the System.Web.Extensions assembly, which is installed in the Global Assembly Cache (GAC) by the Microsoft ASP.NET AJAX Extensions installer.

As you already know from chapter 1, the library files are automatically loaded into an ASP.NET page as soon as you declare a ScriptManager control. Therefore, every Ajax-enabled ASP.NET page must contain a ScriptManager control:

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

Table 2.2 lists the script files that make up the Microsoft Ajax Library, along with the description of the functionality they provide.

Table 2.2 The Microsoft Ajax Library features are distributed across multiple JavaScript files.

Filename	Features
MicrosoftAjax.js	The core library that contains the JavaScript extensions, the type system, classes for the object-oriented patterns, the communication layer, classes for creating components, and classes for dealing with the browser's DOM
MicrosoftAjaxTimer.js	Contains the client timer component used by the Timer server control
MicrosoftAjaxWebForms.js	Contains classes for supporting the partial-update mechanism used by the UpdatePanel server control

The Microsoft Ajax Library is written in pure JavaScript, so it isn't tied to the ASP.NET framework. If you want to work with the library without using ASP.NET, you need to reference the script files with `script` tags in the web page. However, the script files in the ASP.NET AJAX installation directory don't include some resources files needed by the library at runtime. For this reason, you need to download the Microsoft Ajax Library package, which includes all the library files and the resource files; it's available for download at the ASP.NET AJAX official website (http://ajax.asp.net).

All the library files are provided in *debug* and *release* versions. The debug version facilitates the debugging of the script files. It contains comments and takes advantage of a number of tricks that make debuggers happy. For example, it avoids using anonymous JavaScript functions to show more informative *stack traces*. In addition, calls to functions are *validated* to ensure that the number and types of parameters are those expected. The debug version of a library file is slower and bigger than the release version; the release version is compressed, comments are removed, and validation doesn't take place. This results in faster and considerably shorter code.

Let's examine the options you have to load the desired version of a script file.

2.1.3 Script versions

You can load the desired version of a script through the ScriptManager control. You can also load debug and release versions of custom script files. Debug and release versions are distinguished by the file extension: The debug version has the extension .debug.js, and the release version has the normal .js extension.

To load either the debug or the release version, you have to set the `ScriptMode` property of the ScriptReference control that references the script file in the ScriptManager. For example, suppose you want to load the release version of a custom script file called MyScriptFile.js, stored in the ScriptLibrary folder of the website. Here's how the ScriptManager control will look:

```
<asp:ScriptManager ID="TheScriptManager" runat="server">
    <Scripts>
        <asp:ScriptReference Path=" ~/ScriptLibrary/MyScriptFile.js"
                   ScriptMode="Release" />
    </Scripts>
</asp:ScriptManager>
```

Because the `ScriptMode` property is set to `Release`, the script file loaded in the page is MyScriptFile.js. If you set the value of the property to `Debug`, the MyScript-File.debug.js file is loaded.

NOTE Regardless of whether you're loading the debug or release version, the name of the script file in the `Path` attribute must always be that of the release version.

The `ScriptMode` attribute can take one of the following values:

- `Auto`—The name of the script file to load matches the one specified in the `Path` property. This is the default value.
- `Inherit`—The ScriptManager control infers the name of the script file from the compilation mode of the website, as configured in the web.config file. If you're running in `debug` mode, the ScriptManager loads the file with the .debug.js extension. Otherwise, it loads the file with the .js extension.
- `Debug`—The ScriptManager loads the debug version of the script file.
- `Release`—The ScriptManager loads the release version of the script file.

In chapter 13, we'll explain some techniques used to develop a debug version of a custom script file.

After this quick overview of the Microsoft Ajax Library, let's examine some of the features in more detail. In the next sections, we'll discuss the foundations of the library: the Application model and the client page lifecycle.

2.2 The Application model

A web application is made up of pages. Because ASP.NET pages follow an object-oriented model, each page is modeled with an instance of the `Page` class, which encapsulates a hierarchy of controls. Controls follow the *page lifecycle*, which is a set of processing steps that start when the `Page` instance is created and consists of multiple, sequential stages. In the initial stages, like *Init*, controls are instantiated and their properties are initialized. In the final stages, *Render* and *Dispose*, the HTML for the page is written in the response stream, and all the controls and resources, as well as the `Page` instance itself, are disposed.

NOTE To learn more about the ASP.NET page lifecycle, check the MSDN documentation at http://msdn2.microsoft.com/en-us/library/ms178472.aspx.

Imagine that the web page has completed its lifecycle on the server side. The `Page` instance and the controls raised their events, and you handled them to inject the custom application logic. The HTML for the page is ready to be sent down to the browser. If you enabled the Microsoft Ajax Library, a new lifecycle starts on the client side. As soon as the browser loads the main script file, MicrosoftAjax.js, the client runtime creates a global JavaScript object—the Application object—and stores it in a global variable called `Sys.Application`.

This new object becomes the brains of a web page in the browser. Despite its name, it plays a role similar to the `Page` object on the server side. Once the `Page` object is done on the server side, the processing on the client side is delegated to `Sys.Application`, as illustrated in figure 2.1.

The introduction of a global Application object in the browser isn't meant to revolutionize the way you write the client code. The goal is to achieve consistency between the programming models used on both the server and client sides. The main objectives of `Sys.Application` are as follows:

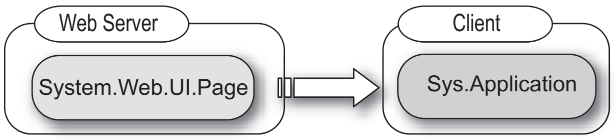


Figure 2.1
On the server side, an ASP.NET page is represented by an instance of the `Page` class. In a similar manner, on the client side, you have the global Sys.Application object.

- *Providing a centralized place to execute the client code*—This goal is reached by defining a custom page lifecycle on the client. As you'll see in a moment, the client page lifecycle starts when the browser loads the page and ends when the user navigates away from the page or the page is reloaded. When each stage in the lifecycle is entered, the Application object raises a corresponding event.
- *Hosting the client components instantiated in the page*—Once instantiated, client components become children of the Application object and can be easily accessed through the Application object. Also, they're automatically disposed by the Application object when the web page is unloaded by the browser.

Client components and the client-page lifecycle are the key concepts we'll dissect in the following sections. Let's start by illustrating the concept of a client component. Then, we'll focus on the client-page lifecycle and the events raised by the Application object.

2.2.1 Client components

Let's say you need a hierarchical menu for navigating the pages of a website. Whether it's written in C# or JavaScript—assuming it isn't poorly designed—you usually don't have to know anything about the logic used to render the menu. Instead, you only have to configure the menu and instantiate it in the page. If you also need the same menu in a different page, you perform similar steps to include and initialize it. The point is, the code should be packaged into a single, configurable object that can be reused in another application.

The primary tenet behind components is code reusability. Components implement a well-defined set of interfaces that allows them to interact with other components and to be interchanged between applications. Thanks to the base interfaces, the code encapsulated by components can change at any time without affecting the other processing logic.

The Microsoft Ajax Library provides specialized client classes that simplify the authoring of client components. The group of classes related to component development is called the *client component model* and closely mirrors the model in use in the .NET framework. In this way, you can write component-oriented client applications using JavaScript code.

We'll explore the nuts and bolts of client components in chapter 8. For now, it's enough to treat a client component as a black box that encapsulates reusable client logic and exposes it through methods, properties, and events, as shown in figure 2.2. You've already met your first component: the Application object introduced in the previous section. In the following section, we'll explain how the Application object and client components interact during the client-page lifecycle.

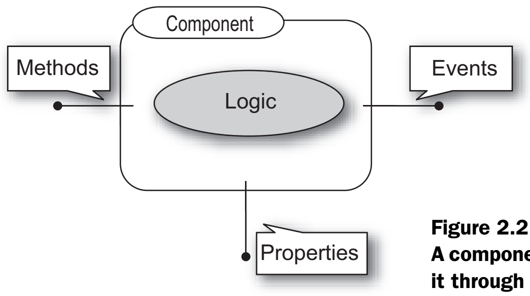


Figure 2.2
A component encapsulates some logic and exposes it through properties, methods and events.

2.2.2 Client-page lifecycle

Previously, we pointed out the Application object acts on the client side like the counterpart of the Page object. The Page object manages the lifecycle of a web page on the server side, and the Sys.Application object accomplishes the same task on the client side. The client lifecycle of a web page is much simpler than the server lifecycle of an ASP.NET page. It consists of only three stages: *init*, *load*, and *unload*. When each stage is entered, the Sys.Application object fires the corresponding event—`init`, `load`, or `unload`.

As shown in the activity diagram in figure 2.3, the client-page lifecycle starts when the browser loads a page requested by the user and ends when the user navigates away from the page. Let's examine the sequence of events in more detail.

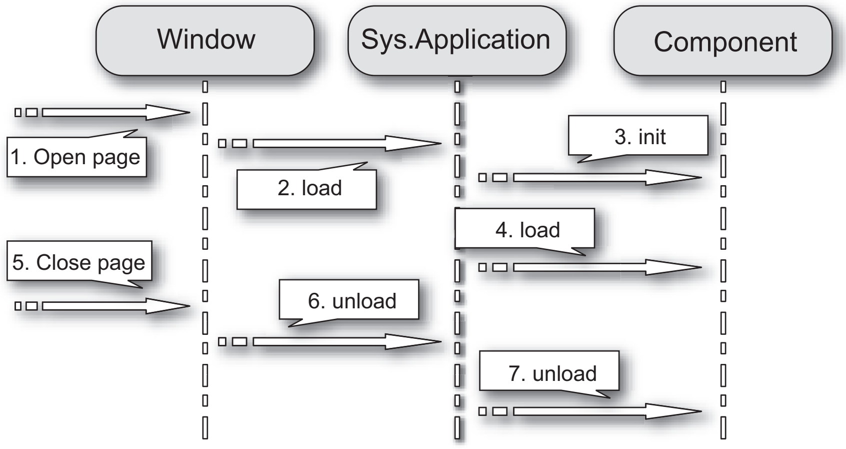


Figure 2.3 The client-page lifecycle starts when the browser loads a web page. The Sys.Application object is responsible for hooking up the events raised by the window object and, in turn, firing its own events. Client components are created during the init stage and disposed automatically in the unload stage.

When the browser starts loading a web page, the DOM's window object fires the `load` event. This event is intercepted by the Application object, which, in turn, starts initializing the Microsoft Ajax Library's runtime. When the runtime has been initialized, Sys.Application fires the `init` event. During the init stage, all the client components you want to use should be instantiated and initialized. As you'll discover in chapter 8, client components are instantiated using a special function called `$create` and are automatically hosted by the Application object.

After the creation of components is complete, Sys.Application fires the `load` event. This stage provides you with an opportunity to interact with the components created in the init stage. This is also the best place to attach event handlers to DOM elements and perform data access—for example, using the techniques for sending asynchronous requests that we'll illustrate in chapter 5.

When the user navigates away from the page or reloads it, the `unload` event of the window object is intercepted by Sys.Application, which in turns fires its own `unload` event. At this point, all the resources used by the page should be freed and event handlers detached from DOM elements.

The events raised by the Sys.Application object and, in general, by client components are different from the events raised by DOM elements. In chapter 3, we'll explain how to expose events in JavaScript objects. The event model used by the Microsoft Ajax Library is similar to the model used by the .NET framework: Events support multiple handlers and can be subscribed and handled in a manner similar to the events raised by ASP.NET server controls.

It's not difficult to deduce that one of the objectives of the Microsoft Ajax Library is bringing .NET flavors to the client side. The Application object, client components, events, and client-page lifecycle are the foundations of the Microsoft Ajax Library. They let ASP.NET developers use familiar development patterns even when writing JavaScript code. Before we go any further, let's take a moment to reinforce what you've learned by putting together a simple program.

2.2.3 "Hello Microsoft Ajax!"

This example illustrates how to write a simple program with the Microsoft Ajax Library. Because we've discussed the Application object and the client-page lifecycle, we'll show them in action in a page. The quickest way to start programming with the Microsoft Ajax Library is to create a new Ajax-enabled website using the Visual Studio template shipped with ASP.NET AJAX. See appendix A for instructions on how to install the Visual Studio template. The template creates a new website and adds a reference to the System.Web.Extensions assembly and a properly configured web.config file. In addition, it creates a Default.aspx page with a ScriptManager control already in it. To run the following example, open the Default.aspx page and insert the code from listing 2.1 in the page's `form` tag.

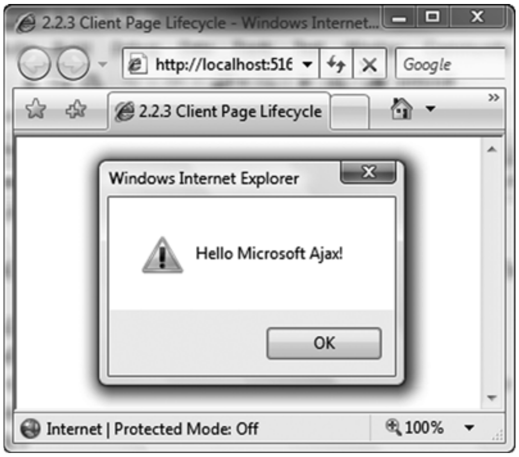
Listing 2.1 Code for testing the client-page lifecycle

```
<script type="text/javascript">
<!--
    function pageLoad() {
        alert("Page loaded!");
        alert("Hello Microsoft Ajax!");
    }

    function pageUnload() {
        alert("Unloading page!");
    }
//-->
</script>
```

The code in listing 2.1 is simple enough for a first program. It consists of two JavaScript functions, `pageLoad` and `pageUnload`, embedded in a `script` tag in the markup code of the web page. In the functions, you call the JavaScript's `alert` function to display some text in a message box on screen.

The names of the functions aren't chosen at random. Provided that you've defined them in the page, the Microsoft Ajax Library automatically calls the `pageLoad` function when the load stage of the client page lifecycle is entered. The `pageUnload` function is automatically called when the unload stage is reached.

When you run the page, the code in `pageLoad` is executed as soon as the load stage is entered. Figure 2.4 shows the example up and running in Internet Explorer. To see what happens during the unload stage, you can either press the F5 key or navigate away from the page.


2.2.3 Client Page Lifecycle - Windows Internet...
http://localhost:516
Google
2.2.3 Client Page Lifecycle
Windows Internet Explorer
OK
Internet | Protected Mode: Off
100%

Hello Microsoft Ajax!

Figure 2.4
The "Hello Microsoft Ajax!" program running in Internet Explorer

The pageLoad function

When you're using the Microsoft Ajax Library, the pageLoad function is the best place to execute the client code. Handling the window.load event isn't safe because the library handles it to perform the runtime initialization. It's always safe to run the code during the load stage of the client-page lifecycle, because the runtime initialization is complete, all the script files referenced through the ScriptManager control have been loaded, and all the client components have been created and initialized.

If you want to detect the init stage, you have to do a little more work. Declaring a `pageInit` function won't have any effect. Instead, you have to write an additional statement with a call to the `add_init` method of Sys.Application, as shown in listing 2.2.

Listing 2.2 Handling the `init` event of Sys.Application

```
Sys.Application.add_init(pageInit);

function pageInit() {
    alert("Entered the Init stage!");
}
```

The `add_init` method adds an event handler for the `init` event of Sys.Application. The event handler is the `pageInit` function you passed as an argument to the method. The Application object also has `add_load` and `add_unload` methods, which add event handlers to the `load` and `unload` events, respectively. However, the `pageLoad` and the `pageUnload` functions offer a way to execute code during the load and unload stages of the client page lifecycle.

The init stage is typically used to create instances of the client components you use in the page. However, we won't deal with it until chapter 8, where we'll explain the nuts and bolts of client components. The majority of the client code, including attaching event handlers to DOM elements and sending Ajax requests, can be safely executed in the `pageLoad` function.

Now that you've written your first program, let's focus on the client code a little more. Web developers use JavaScript primarily to access a web page's DOM. The DOM is an API used to access a tree structure built from a page's markup code. The following sections explore how to use the Microsoft Ajax Library to program against the browser's DOM.

2.3 Working with the DOM

When a browser renders a page, it builds a hierarchical representation (called the *DOM tree*) of all the HTML elements like buttons, text boxes, and images. Every element in the page becomes a programmable control in the DOM tree and exposes properties, methods, and events. For example, an `input` tag with its `type` attribute set to `button` is parsed into a button object with a `value` property that lets you set its text. The button can also raise a `click` event when it's clicked. The ability to manipulate DOM elements makes the difference between static and dynamic HTML pages. It's possible to change the behavior of the UI elements at any time, based on the user's inputs and interactions with the page.

But this is where life gets tricky. Almost all browsers implement the DOM programming interface differently. In some cases, there are differences between versions of the same browser. This means a dynamic page that works on one browser may stop working on another browser and complain about JavaScript errors. At this point, you're forced to duplicate the code to work around the browser incompatibilities.

The Microsoft Ajax Library addresses this serious problem by providing an *abstraction API* whose purpose is to abstract common operations made on DOM elements, such as handling their events and working with CSS. As we'll explain, the API frees you from having to know which functions are supported by the DOM implementation of a particular browser. It takes care of calling the correct function based on the browser that is rendering the page.

2.3.1 The abstraction API

The Microsoft Ajax Library lets you access the DOM in a manner independent from the browser that renders the page. The abstraction API consists of the methods exposed by two client classes: `Sys.UI.DomElement` and `Sys.UI.DomEvent`. The first one abstracts a DOM element, and the second represents the event data object that DOM event handlers receive as an argument.

Using this model, you prevent the code from dealing directly with the browser's API. Instead, you call methods defined in the Microsoft Ajax Library, which takes care of calling the correct function based on the browser that is currently rendering the page. Figure 2.5 illustrates this concept by showing how the DOM calls in a script file can be made through the `Sys.UI.DomElement` and `Sys.UI.DomEvent` classes.

For example, suppose you want to hook up an event raised by a DOM element. Instead of checking whether a browser supports an `attachEvent` rather than an `addEventListener` method, you can call the `addHandler` method of the

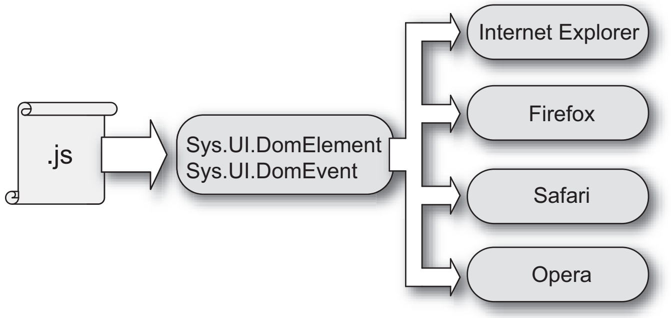


Figure 2.5 The Microsoft Ajax Library provides a common interface to different DOM implementations. The library translates the DOM calls made with the `Sys.UI.DomElement` class into browser-specific functions. `Sys.UI.DomEvent` offers a cross-browser object for representing event data.

`Sys.UI.DomElement` class. Then, it's up to the Microsoft Ajax Library to call the correct function based on the detected browser.

You know that Firefox passes the event object as an argument to the event handler, whereas Internet Explorer stores its custom event object in the `window.event` property. If you use the abstraction API, the same cross-browser event object is always passed as an argument to the event handler. Thanks to the cross-browser event object, you're also freed from the pain of dealing with different properties that describe the same event data.

To give you confidence using the API, let's work on an example that explains how to use some of the methods provided to handle an event raised by a DOM element. Later, we'll address CSS and positioning.

2.3.2 A dynamic, cross-browser text box

One of the main uses of JavaScript is to check and validate user input before a web form is submitted to the server. To perform this task, you often have to write client logic that limits the values a user can enter in a text field. As an example, suppose you want a user to enter some text in a text box. The requirements say that the text must contain only letters—no digits. To implement this task, you must access the text-box element, handle its `keypress` event, and filter the text typed by the user. Listing 2.3 shows how this task can be accomplished using the Microsoft Ajax Library. Notably, the resulting code runs in all the browsers supported by the library.

Listing 2.3 A text box that accepts only letters

```
<div>
    <span>Please type some text:</span>
    <input type="text" id="txtNoDigits" />
</div>

<script type="text/javascript">
```

```
<!--
    function pageLoad() {
        var txtNoDigits = $get('txtNoDigits');      Access text
                                                     box element
        $addHandler(txtNoDigits,                     Attach event
            'keypress', txtNoDigits_keypress);       handler
    }

    function pageUnload() {
        $removeHandler($get('txtNoDigits'),          Detach event
            'keypress', txtNoDigits_keypress);       handler
    }

    function txtNoDigits_keypress(evt) {
        var code = evt.charCode;
                                                     Handle keypress
        if(code >= 48 && code <= 57) {               event
            evt.preventDefault();
        }
    }
//-->
</script>
```

As you know, the code in the `pageLoad` function is executed as soon as the load stage of the client page lifecycle is entered. The function body contains calls to the `$get` and `$addHandler` methods.

Shortcuts

As we explained in chapter 1, functions prefixed with the character $ are aliases or shortcuts used to access methods with longer names. This saves you a little typing. Even if it seems like a minor detail, using shortcuts is useful for obtaining shorter code and smaller JavaScript files. Table 2.3 lists some of the shortcuts defined in the Microsoft Ajax Library together with the longer name of the associated method.

The first method called in `pageLoad` is `$get`, which gets a reference to a DOM element. `$get` accepts the ID of a DOM element and returns a reference to it. You can also pass a reference to a DOM element as the second parameter. In this case, the method searches an element with the given ID in the child nodes of the provided DOM element.

Table 2.3 Shortcuts for common methods defined in the abstraction API

Shortcut	Method Name	Description
`$get`	`Sys.UI.DomElement.getElementById`	Returns a reference to a DOM element
`$addHandler`	`Sys.UI.DomElement.addHandler`	Adds an event handler to an event exposed by a DOM element
`$removeHandler`	`Sys.UI.DomElement.removeHandler`	Removes an event handler added with `$addHandler`
`$addHandlers`	`Sys.UI.DomElement.addHandlers`	Adds multiple event handlers to events exposed by DOM elements and wraps the handlers with delegates
`$removeHandlers`	`Sys.UI.DomElement.removeHandlers`	Removes all the handlers added with `$addHandler` and `$addHandlers`

The second method, `$addHandler`, adds an event handler for an event raised by a DOM element. The first argument is a reference to the element that exposes the event you want to subscribe. The second argument is a string with the name of the event. The third argument is the JavaScript function that handles the event. The syntax of `$addHandler` is illustrated in figure 2.6. Note that the string with the name of the event must not include the prefix `on`.

You can remove the handler added with `$addHandler` at any time by passing the same arguments—the element, the event name, and the handler—to the `$removeHandler` method. It's a good practice to always dispose event handlers

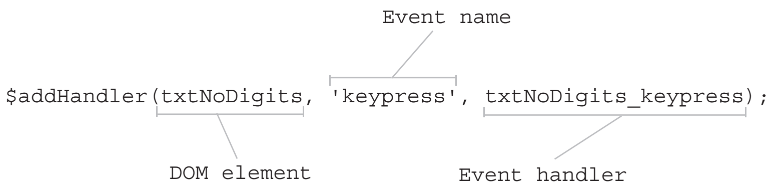


Figure 2.6 Syntax for the `$addHandler` method, used for attaching a handler to an event raised by a DOM element

added to DOM elements. You do so to prevent memory leaks in the browser that could slow the application and cause a huge performance drop. As in listing 2.3, a good place to use `$removeHandler` is in the `pageUnload` function, which is called just before the browser unloads the web page.

In the previous example, the handler for the text box's `keypress` event is the `txtNoDigits_keypress` function. The handler is called as soon as a key is pressed in the text field. As you can see, the first argument passed to the `txtNoDigits_keypress` function, `evt`, is an instance of the `Sys.UI.DomEvent` class. The main characteristic of this object is that it contains the event data and exposes the same properties in all the supported browsers.

NOTE The `$addHandler` method uses a private function to subscribe the event. When the private handler is executed, the browser-specific event object is converted into a `Sys.UI.DomEvent` instance. At this point, the original event handler passed to `$addHandler` is called with the `Sys.UI.DomEvent` instance as an argument, in place of the original event object.

Among the various properties of the event object, `charCode` returns the code of the typed character. The `preventDefault` method is invoked to avoid the execution of the default action for the subscribed event. In the example, this prevents characters whose code corresponds to a digit from being displayed in the text box. Table 2.4 lists all the properties of the cross-browser event object.

Table 2.4 Properties of the `Sys.UI.DomEvent` class, which provides a cross-browser object that contains the event data for DOM events

Property	Value
`rawEvent`	Underlying event-data object built by the current browser
`shiftKey`	True if the Shift key was pressed
`ctrlKey`	True if the Ctrl key was pressed
`altKey`	True if the Alt key was pressed
`button`	One of the values of the Sys.UI.MouseButton enumeration: leftButton, middleButton, rightButton
`charCode`	Character code for the typed character
`clientX`	During a mouse event, the x coordinate of the mouse location relative to the client area of the page
`clientY`	During a mouse event, the y coordinate of the mouse location relative to the client area of the page

Table 2.4 Properties of the `Sys.UI.DomEvent` class, which provides a cross-browser object that contains the event data for DOM events *(continued)*

Property	Value
`target`	Element that raised the event
`screenX`	During a mouse event, the x coordinate of the mouse location relative to the computer screen
`screenY`	During a mouse event, the y coordinate of the mouse location relative to the computer screen
`type`	Name or type of the event (like click or mouseover)
`preventDefault()`	Prevents execution of the default action associated with the event
`stopPropagation()`	Prevents the event from propagating up to the element's parent nodes

Changing the layout of DOM elements is one of the main tasks in dynamic HTML pages. For example, features like animations, scrolls, and drag and drop rely on the positioning and the style of DOM elements. Let's return to the `Sys.UI.DomElement` class to examine a group of methods related to CSS and positioning.

2.3.3 *CSS and positioning*

The `Sys.UI.DomElement` class provides a group of methods for performing common tasks related to CSS and positioning. If you pass a DOM element and a string with the name of a CSS class to the `addCssClass` method, you add the class to the list of CSS classes associated with an element. If you pass the same arguments to the `removeCssClass` method, you remove the class from the list of associated CSS classes.

If there's one thing that highlights the incompatibilities between the various implementation of the DOM, it's positioning. Due to the different ways in which the many parameters related to the box model are computed, making a UI look the same in multiple browsers can be a real challenge.

The box model

The box model describes the layout of DOM elements in a web page through a set of parameters implemented in CSS. You can read a good article about the box model at http://www.brainjar.com/css/positioning/.

The Microsoft Ajax Library tries to mitigate this issue by providing methods that take into account bugs and different algorithms for computing the position and bounds of DOM elements. To retrieve the location of an element, you write a statement like the following:

```
var location = Sys.UI.DomElement.getLocation(element);
```

The `getLocation` method takes a reference to a DOM element and returns an object with two attributes, `x` and `y`, which store the left and top coordinates of the element relative to the upper-left corner of the parent frame:

```
var top = location.y;
var left = location.x;
```

The `setLocation` method accepts the `x` and `y` coordinates (specified in pixels) and sets the element's position using the `left` and `top` attributes of its `style` object. In this case, the element's location is set based on the positioning mode of the element and its parent node. Consider the following statement:

```
Sys.UI.DomElement.setLocation(element, 100, 100);
```

If the parent node of `element` has a specific positioning mode (for example, `relative` or `absolute`), then its location—by CSS rules—is set relative to the parent element and not to the parent frame.

If you need to know the *bounds* of an element—its location and its dimensions—the `Sys.UI.DomElement.getBounds` method takes a reference to a DOM element and returns an object with four attributes: `x`, `y`, `height`, and `width`.

The main goal of the abstraction API isn't to increase differences by adding a new group of methods to learn and remember; the objective is to offer a consistent programming interface to the browser's DOM. When the Microsoft Ajax Library evolves and adds support for new browsers, the code written with the API will continue to work as before. This is possible because under the hood the library takes care of all compatibility issues.

It's time to leave the abstraction API, but your work with the DOM isn't over. Handling DOM events is one of the most common tasks when scripting against a web page. In the next sections, we'll introduce two useful methods for implementing callbacks and *delegates* in JavaScript. We'll show how you can use them to make DOM event-handling easy and straightforward.

2.3.4 Client delegates

Think of a button on an ASP.NET page or on a Windows Forms form. When the user clicks the button, the Button object realizes this and raises a `click` event, which implicitly means, "Hey, someone clicked me, but I'm just a Button. What should happen now?" In the .NET framework, objects use a delegate to invoke one or multiple methods responsible for processing the event.

NOTE To learn more about delegates in the .NET framework, browse to the following URL: http://msdn.microsoft.com/msdnmag/issues/01/04/net/.

To implement a similar pattern in JavaScript, the Microsoft Ajax Library provides a method called `Function.createDelegate`, which accepts two arguments and returns a new function—let's call it the *client delegate*. The first argument is the object that will be pointed to by `this` in the client delegate. The second argument is the function that you want to invoke through the client delegate.

When you invoke the client delegate, it calls the function you passed previously. The difference is that now, `this` points to the object you passed as the first argument to `createDelegate`.

To understand why a client delegate is useful, recall what happens when you add a handler to an event raised by a DOM element. In the handler, `this` always points to the element that hooked up the event, which determines the *scope* you can access. Using a client delegate lets you access—through the `this` keyword—a different object than the DOM element that hooked up the event.

Listing 2.4 demonstrates how to use a client delegate to handle the `click` event of a button element. In the event handler, you can access a variable stored in a different scope.

Listing 2.4 Using a client delegate to handle a DOM event

```
<input type="button" id="testButton" value="Click Me" />

<script type="text/javascript">
<!--
  function pageLoad() {
     this.test = "I'm a test string!";          ❶

     var clickDelegate =                        ❷
        Function.createDelegate(this, onButtonClick);

     $addHandler($get('testButton'), 'click', clickDelegate);   ❸
  }
  function onButtonClick() {
     alert(this.test);                          ❹
  }
//-->
</script>
```

❶ **test property of window object**
❷ **Create client delegate**
❸ **Handle click event with delegate**
❹ **Access the window object**

The ❷ `clickDelegate` variable stores a client delegate that ❸ invokes the `onButtonClick` function. In the function, `this` points to the object passed as the first argument to `createDelegate`. Because `pageLoad` is a global function, `this` points to the global object, which will be accessible in the ❹ `onButtonClick` function. To demonstrate this, you alert the value of the ❶ `test` variable, which was added to the `window` object in the first statement of the `pageLoad` function. If you try to pass the `onButtonClick` function—instead of the client delegate—to the `$addHandler` method, you'll notice that the `test` variable is `undefined`, because `this` now points to the button element that raised the `click` event.

The `Function.createDelegate` method is useful because you don't have to store in a global variable—or even in a DOM element—the context that you want to access in the event handler. Plus, the Microsoft Ajax Library provides two methods for subscribing to multiple DOM events, creating delegates for them, and disposing both the delegates and the handlers automatically. These methods are accessed through the `$addHandlers` and `$clearHandlers` shortcuts.

2.3.5 *$addHandlers and $clearHandlers*

The main advantage of using `$addHandlers` and `$clearHandlers` is to avoid the error-prone (and boring) job of creating client delegates, attaching handlers, and then performing the reverse task, all multiplied by the number of events you're subscribing. For example, the following statement automatically creates the client delegates for the `click` and `mouseover` events of a button element:

```
$addHandlers(buttonElement, { click:onButtonClick,
        mouseover:onMouseOver }, this);
```

The first argument passed to `$addHandlers` is the DOM element. The second parameter is an object for which each property is the name of an event, and whose value is the event handler. The last parameter is the *owner* of the handler, which determines the context under which the event handler is executed. Usually you pass `this`, which lets you access—in the handler—the object that subscribed to the event.

If you need to detach all the event handlers from the element, you can do so with a single statement:

```
$clearHandlers(buttonElement);
```

Note that the `$clearHandlers` method detaches all the event handlers attached with `$addHandler` and `$addHandlers`. It also disposes all the delegates created with the `$addHandlers` method.

NOTE Browsers can leak memory if event handlers aren't correctly detached from DOM elements. For this reason, it's important to always detach all the event handlers and dispose all the delegates when a client object is cleaned up or the whole page is unloaded.

You don't always need to switch the scope of an event handler to access the information you need. Often, it's enough to access a context object where you've stored only the references that may be useful during the processing of the event. In this case, a better approach is to handle a DOM event using a callback function.

2.3.6 Callbacks

The Microsoft Ajax Library provides a method called `Function.createCallback`, which you can use to create a callback function that an object can invoke at any time. The main purpose of `Function.createCallback` is to call a function with a *context* provided by the object that created the callback. The context is an object that, usually, contains application data that otherwise wouldn't be accessible in the method, because they belong to a different scope. Just like client delegates, callbacks are useful for processing DOM events. The code in listing 2.5 shows how you can access, in a DOM event handler, a custom object created in the `pageLoad` function.

Listing 2.5 Using a callback to handle a DOM event

```
<input type="button" id="myButton"
       value="Time elapsed since page load" />

<script type="text/javascript">
<!--
    function pageLoad() {
        var context = { date : new Date() };        ❶ context object

        var clickCallback =
            Function.createCallback(onButtonClick, context);    ❷ Create callback

        $addHandler($get('myButton'), 'click', clickCallback);
    }                                     Attach handler to click event ❸

    function onButtonClick(evt, context) {
        var loadTime = context.date;
        var elapsed = new Date() - loadTime;    ❹ Access event object and context

        alert((elapsed / 1000) + ' seconds');
    }
//-->
</script>
```

The ❶ context variable stores the time at which the pageLoad function was invoked. You want to access this information in the function that handles the button's click event. To do that, you ❷ create a callback that points to the onButtonClick function. In the Function.createCallback method, you specify the function to invoke and the context object. ❸ Then, you pass the callback to the $addHandler method. ❹ When the onButtonClick function is called, the context object is added to the list of arguments, just after the event object.

Callbacks and delegates are nice additions to the set of features you can leverage when programming against the DOM with the Microsoft Ajax Library. But in the Ajax age of web applications, interacting with the DOM is just a small portion of the tasks the client code should accomplish. You need features and tools that simplify everyday programming with JavaScript. In the following sections, we'll explore some of the classes that the Microsoft Ajax Library provides to increase productivity on the client side.

2.4 Making development with JavaScript easier

Since the first versions of JavaScript, developers have started writing libraries to leverage the base functionality provided by the language. Every new library or framework that sees the light—and lately, this seems to happen on a monthly basis—aims at increasing the productivity of JavaScript developers by offering a set of tools that makes it easier to design, write, and debug the client code. As you may have guessed, the direction taken by the Microsoft Ajax Library is to achieve the same goals by bringing some of the .NET coding patterns to the client side. Let's look at some of the client classes provided by the library, starting with the enhancements made to the built-in JavaScript objects. Later, we'll talk about browser detection and client-side debugging.

2.4.1 The String object

String manipulation is one of the most common tasks in everyday programming. JavaScript comes with a String object that contains various methods for dealing with strings. However, some frequently desired methods such as format and trim aren't in the list. The good news is that the Microsoft Ajax Library extends—at runtime—the String object to make it more similar to the counterpart class in the .NET framework. For example, one of the methods added to the client String object is format. You can use numbered placeholders like {0} and {1} to format strings using variables, just as you do on the server side:

```
alert(String.format("This code is running on {0} {1}",
    Sys.Browser.agent, Sys.Browser.version));
```

The Sys.Browser object can be used to get information about the browser that loaded a page. We'll return to browser detection in section 2.4.4. In the meantime, look at table 2.5, which lists the new methods added to the JavaScript String object.

Table 2.5 Methods added to the JavaScript String object

Method	Description
`endsWith`	Determines whether the end of the String object matches the specified string.
`format`	Replaces each format item in a String object with the text equivalent of a corresponding object's value.
`localeFormat`	Replaces the format items in a String object with the text equivalent of a corresponding object's value. The current culture is used to format dates and numbers.
`startsWith`	Determines whether the start of the String object matches the specified string.
`trim`	Removes leading and trailing white space from a String object instance.
`trimEnd`	Removes trailing white space from a String object instance.
`trimStart`	Removes leading white space from a String object instance.

An object familiar to .NET developers is the *string builder*. A string builder is an object that speeds up string concatenations because it uses an array to store the parts instead of relying on temporary strings. As a consequence, a string builder is orders of magnitude faster than the + operator when you need to concatenate a large number of strings.

2.4.2 Sys.StringBuilder

The `Sys.StringBuilder` class closely resembles the `System.Text.StringBuilder` class of the .NET framework. In JavaScript, it's common to build large chunks of HTML dynamically as strings and then use the `innerHTML` property of a DOM element to parse the HTML. Even if it isn't a standard DOM property, `innerHTML` is orders of magnitude faster than the standard methods for manipulating the DOM tree. Listing 2.6 shows how to use an instance of the `Sys.StringBuilder` class to format the URL of a web page and display it on a label. Instances of client classes are created with the `new` operator in the same way as JavaScript custom objects. We'll return to client classes and other object-oriented constructs in chapter 3.

Listing 2.6 Example using the `Sys.StringBuilder` class

```
<span id="urlLabel"></span>

<script type="text/javascript">
<!--
    function pageLoad(sender, e) {
        var sb = new Sys.StringBuilder();    <-- Create instance of StringBuilder

        sb.append('<h3>You are now browsing: ');
        sb.append('<b><i>');
        sb.append(window.location);
        sb.append('</i></b></h3>');

        var myLabel = $get('urlLabel');

        urlLabel.innerHTML = sb.toString();    <-- Inject HTML chunk in span element
    }
//-->
</script>
```

To add the various parts to the final string, you pass them to the `append` method. When you're done, you call the `toString` method to get the whole string stored in the `StringBuilder` instance. The `Sys.StringBuilder` class also supports an `appendLine` method that adds a line break after the string passed as an argument. The line break is the escape sequence `\r\n`, so you can't use it when building HTML. Instead, you write something like this:

```
sb.append('<br />');
```

You can use the `isEmpty` method to test whether a `StringBuilder` instance doesn't contain any text. The following `if` statement performs this check and, if the `StringBuilder` isn't empty, clears all the text using the `clear` method:

```
if(!sb.isEmpty()) {
    sb.clear();
}
```

When the number of strings to concatenate is larger, the string builder becomes an essential object to avoid huge performance drops. The `Sys.StringBuilder` class relies on an array to store the strings to concatenate. Then, it uses the `join` method of the String object to perform the concatenation and obtain the final string.

Arrays are probably one of the most used data structures in programming. JavaScript provides a built-in Array object that the Microsoft Ajax Library extends with methods commonly found in the .NET `Array` class.

2.4.3 The Array object

In JavaScript, an *array* is an ordered collection of values that can be of different types. The built-in Array object exposes many methods to deal with arrays, but the Microsoft Ajax Library extends it so it looks and feels similar to the Array object of the .NET framework. Table 2.6 lists the new methods added to the Array object, together with their descriptions.

Table 2.6 Extension methods added to the JavaScript Array object

Method	Description
`add`	Adds an element to the end of an Array object
`addRange`	Copies all the elements of the specified array to the end of an Array object
`clear`	Removes all elements from an Array object
`clone`	Creates a shallow copy of an Array object
`contains`	Determines whether an element is in an Array object
`dequeue`	Removes the first element from an Array object
`forEach`	Performs a specified action on each element of an Array object
`indexOf`	Searches for the specified element of an Array object and returns its index
`insert`	Inserts a value at the specified location in an Array object
`parse`	Creates an Array object from a string representation
`remove`	Removes the first occurrence of an element in an Array object
`removeAt`	Removes an element at the specified location in an Array object

All the new methods added to the Array type act as *static* methods. As we'll clarify in chapter 3, one of the ways to extend an existing JavaScript object is to add a method directly to its type. As a consequence, the new method can be called directly on the type rather than on a new instance. This is similar to static methods in C# and shared functions in VB.NET; in practice, the usage is the same in JavaScript. In listing 2.7, you create an array and play with some of the new methods introduced by the Microsoft Ajax Library.

Listing 2.7 Some of the new methods added to the Array object

```
<script type="text/javascript">
<!--
    function pageLoad(sender, e) {
        var arr = new Array();

        Array.add(arr, 3);
        Array.addRange(arr, [4, 5, "Hello World!"]);

        Array.removeAt(arr, arr.length - 1);

        var sum = 0;
        Array.forEach(arr,
           function(item) { sum += item; });

        alert(sum);

        Array.clear(arr);
    }
//-->
</script>
```

Add items

Remove last item

Compute sum of items

Clear array

The first thing that .NET developers may notice is that methods have familiar names. The code in listing 2.7 could have been written using the standard methods of the built-in Array object. On the other hand, the Microsoft Ajax Library combines or renames them to achieve, wherever possible, consistency between server-side and client-side classes.

As we said previously, the new methods added to the `Array` object are static. For example, the `addRange` method is called as `Array.addRange` and accepts two arguments: the array instance on which you're working, and an array with the elements you want to add to the instance. The reason for having static methods is that the JavaScript `for/in` construct, if used to loop over the elements of an array, would return any methods added to the `Array` object. This happens because the `for/in` construct loops through the properties of a generic object. On the other hand, static methods aren't returned if you use `for/in` to loop over an array instance.

An interesting new method is `Array.forEach`, which loops through an array and processes each element with the function passed as the second argument. The example uses `Array.forEach` with a simple function that computes the sum of the integers in the array.

Often, web applications have a much wider scope than a company's intranet. Many are websites that can be browsed by virtually any corner of the world. However, different cultures have different ways to represent data such as dates, numbers, and currencies. It's important for applications to be aware of these differences and take *globalization* and *localization* into account.

2.4.4 Globalization

On the server side, ASP.NET does a great job at globalization by providing a group of objects that store information about different cultures. For example, you can rely on specific CultureInfo objects to set a culture for the web application. These objects contain all the settings relative to a particular culture, such as date and number formats.

Such infrastructure isn't available on the client side, where you usually have to elaborate your custom strategy or, worse, renounce it to implement globalization. Luckily, the Microsoft Ajax Library makes enabling globalization on the client side a piece of cake. If you set the `EnableScriptGlobalization` property of the `ScriptManager` control to `true`, the CultureInfo object relative to the current culture (set on the server side) is serialized using the JSON data format and then sent to the browser at runtime.

The serialized object is stored in the `Sys.CultureInfo` variable and contains two child objects: InvariantCulture and CurrentCulture. These objects contain the settings relative to the invariant culture and the current culture as set on the server side. The serialized CultureInfo object is used in conjunction with a group of new methods added to the Date and Number objects by the Microsoft Ajax Library.

The Date object provides formatting capabilities and localized parsing. You can format date instances by passing a format string to the `format` method, like so:

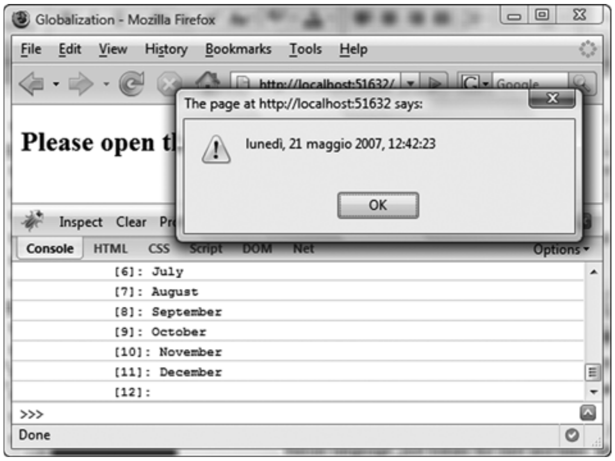
```
var date = new Date();
var formatString =
    Sys.CultureInfo.CurrentCulture.dateTimeFormat.LongDatePattern;

alert(date.format(formatString));
```

This code snippet uses the CurrentCulture object to obtain a standard format string, but you could also use a custom format string.

NOTE If you need more information about standard and custom format strings for date and numbers, visit http://msdn2.microsoft.com/en-us/library/427bttx3.aspx and http://msdn2.microsoft.com/en-us/library/97x6twsz.aspx.

The `format` method uses the Sys.CultureInfo.InvariantCulture object to produce a string with the formatted date. If you want to use the `CurrentCulture` object, you have to pass the format string to the `formatLocale` method:

```
alert(date.formatLocale(formatString));
```


Globalization - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:51632/
Inspect Clear
Console HTML CSS Script DOM Net Options
OK
Done

The page at http://localhost:51632 says:

lunedì, 21 maggio 2007, 12:42:23

```
[6]: July
[7]: August
[8]: September
[9]: October
[10]: November
[11]: December
[12]:
>>>
```

Figure 2.7
A `Date` instance can be formatted using the current culture as set in an ASP.NET page on the server side.

Figure 2.7 shows how a `Date` instance is formatted in the Italian culture (`it-IT`). Similar rules are used for parsing strings that contain date representations. The `Date.parseInvariant` method parses a string representation of a date using the InvariantCulture object. On the other hand, the `Date.parseLocale` method takes advantage of the CurrentCulture object.

The `Number` object has been extended with similar methods. You have `format` and `localeFormat` methods to format `Number` instances using the invariant culture or the current culture, as set on the server side. Similarly, the `parse` and `parseLocale` methods are responsible for parsing a string with a number representation using the desired culture settings.

NOTE You can browse the MSDN documentation for globalization and localization at http://msdn2.microsoft.com/en-us/library/c6zyy3s9.aspx.

In addition to globalization, the ASP.NET AJAX framework provides support for localization. You can think of localization as the translation of a page into a particular culture: Client resources like JavaScript files and strings can now be localized from the client, not just the server. Chapter 4 will walk you through how this is attainable.

Now that you know how to deal with the enhanced JavaScript object, let's focus on the common tasks performed in everyday programming with JavaScript. Since web developers started scripting against web pages, browser detection has played a major role, especially due to the incompatibilities in the JavaScript implementations of different browsers.

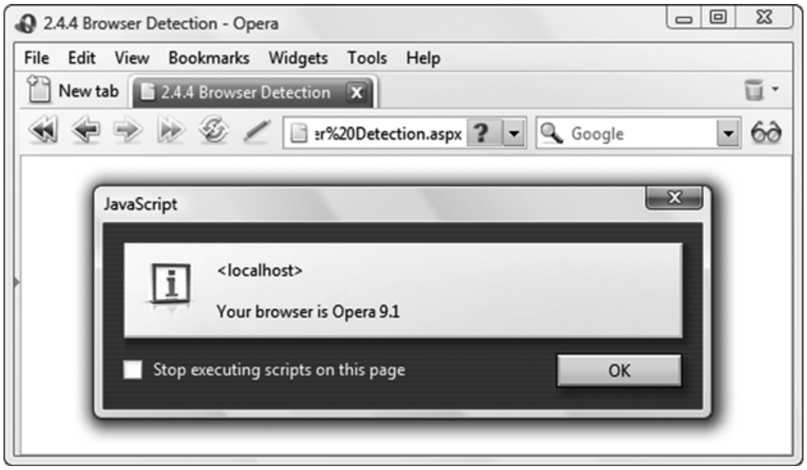
2.4.5 Browser detection

The Microsoft Ajax Library extracts information about the browser that is rendering the page from the DOM's navigator object. This information is stored in an object called Sys.Browser, which you can use to perform browser detection on the client side. To see browser detection in action, the code in listing 2.8 displays a message with the name and the version of the detected browser.

Listing 2.8 Using the Sys.Browser object to perform browser detection

```
<script type="text/javascript">
<!--
  function pageLoad(sender, e) {
    var browser = String.format("Your browser is {0} {1}",
       Sys.Browser.name, Sys.Browser.version);

    alert(browser);
  }
//-->
</script>
```

The `name` and `agent` properties of the Sys.Browser object contain the browser's name and the current version. Figure 2.8 shows how this information is displayed in a message box in the Opera browser.


2.4.4 Browser Detection - Opera
File Edit View Bookmarks Widgets Tools Help
New tab
2.4.4 Browser Detection
er%20Detection.aspx
Google
JavaScript
Stop executing scripts on this page
OK

<localhost>

Your browser is Opera 9.1

Figure 2.8 You can use the Sys.Browser object to perform browser detection at runtime.

Sometimes it's useful to take an action only if a particular browser is detected. In this case, you can test against the object returned by Sys.Browser.agent. For example, add the following statement in listing 2.8, after the call to the `alert` function:

```
if(Sys.Browser.agent == Sys.Browser.InternetExplorer) {
    alert('This message is displayed only on Internet Explorer!');
}
```

As you can easily verify, the message box in this code is displayed only in Internet Explorer. You can also test the `Sys.Browser.agent` property against `Sys.Browser.Firefox`, `Sys.Browser.Opera`, and `Sys.Browser.Safari`.

Performing browser detection is a small step toward the goal of writing client code that runs smoothly and behaves as expected in all the browsers you're targeting. Ajax web applications have introduced the need to dedicate even more time to debugging. The following sections will examine the tools available for debugging JavaScript code. We'll also discuss how the Microsoft Ajax Library can help you improve your debugging experience and handle errors.

2.4.6 Debugging

Debugging JavaScript code has never been the most exciting experience for a web developer. On one hand, browsers don't have embedded debuggers and often provide cryptic error messages that give no clue what went wrong. On the other hand, JavaScript continues to lack a real development environment, and almost all the debuggers are available as external tools. Given this situation, it shouldn't come as a surprise that one of the preferred debugging techniques has always been displaying messages on screen by calling the `alert` function at the right time in the code.

The situation is slowly getting better. All the modern browsers have a JavaScript console that logs client errors, and interesting tools are being developed. Among these, Firebug for Firefox and Web Development Helper for Internet Explorer have proven to be must-have tools for Ajax developers. These tools integrate with the browser and let you inspect and debug web pages directly in the browser. The next version of Visual Studio (codename "Orcas") will provide a real development environment for JavaScript, with features such as improved debugging and IntelliSense in script files. This book's Resources section points you to the URLs where you can download some of these tools. Also, Appendix B contains instructions on how to install them, together with an overview of their features.

The Microsoft Ajax Library offers support for code debugging through a class called `Sys.Debug`. This class exposes methods for logging messages to the browser's console and dumping client objects. To log a message to the console, you can call the `Sys.Debug.trace` method anywhere in the code, passing a string with the message as an argument:

```
Sys.Debug.trace("I'm a debug message.");
```

You can also dump an object by passing it to the `Sys.Debug.traceDump` method. An object dump displays all the properties of an object, together with their values, and can be helpful during debugging. The following example logs to the console the properties of the object returned by the `getBounds` method of the `Sys.UI.DomElement` class:

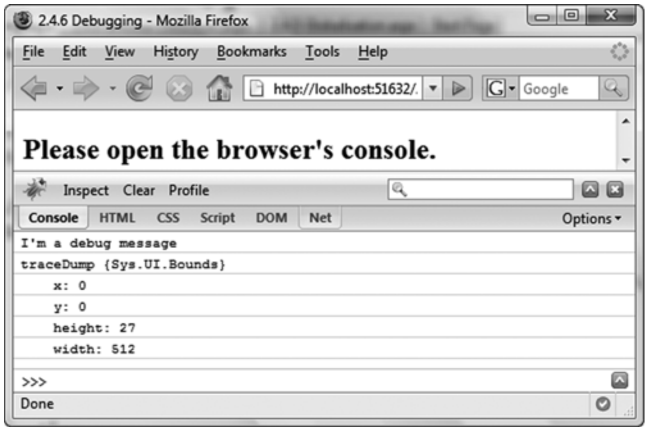
```
Sys.Debug.traceDump(Sys.UI.DomElement.getBounds(document.body));
```

Figure 2.9 shows the logged messages in the Firebug console of the Firefox browser. If you prefer to see the messages directly on the page area rather than in the JavaScript console, declare a `textarea` element with an ID of `TraceConsole`, like so:

```
<textarea id="TraceConsole" rows="30" cols="50"></textarea>
```

In this way, all messages passed to `Sys.Debug.trace` are displayed in the `textarea` element.

If you consider the two main tasks performed by Ajax applications—updating portions of the page layout and performing data access in the background—it's clear that a relevant part of the application logic consists of sending asynchronous HTTP requests to the server. For this reason, there's an increasing need for Ajax developers to monitor what happens during data transfers and to inspect the content of the requests and responses sent during the life of a web page.


2.4.6 Debugging - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:51632/
Google
Inspect Clear Profile
Console HTML CSS Script DOM Net
Options
Done

Please open the browser's console.

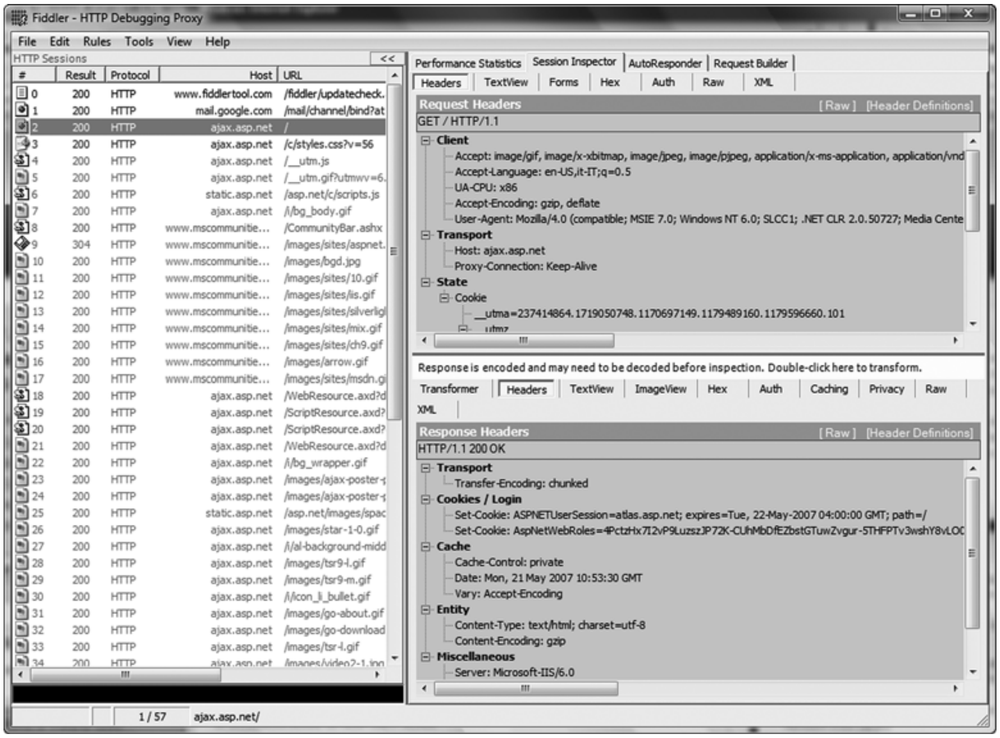
```
I'm a debug message
traceDump {Sys.UI.Bounds}
    x: 0
    y: 0
    height: 27
    width: 512
>>>
```

Figure 2.9 Debug messages logged to the Firebug console in Firefox

For example, a response that never arrives may suggest that you have problems related to server availability or to network latency. On the other hand, receiving a response with a status code of 500 indicates that something went wrong on the server. In chapter 5, we'll develop techniques for detecting errors during HTTP transactions; but in some cases you need to inspect the headers or the payload of a request or a response.

HTTP debugging is provided by some of the tools recently available to Ajax developers. Fiddler, Firebug for Firefox, and Web Development Helper for Internet Explorer offer this kind of functionality. Figure 2.10 shows a request traced using the Fiddler tool.

Web Development Helper was developed by a member of the ASP.NET team and provides specific functionality for inspecting the content of a response sent during partial rendering. You'll see this tool in action in chapter 7, where we'll go under the hood of the `UpdatePanel` control and the partial rendering mechanism.


Fiddler - HTTP Debugging Proxy
File Edit Rules Tools View Help
HTTP Sessions
Performance Statistics
Session Inspector
AutoResponder
Request Builder
Headers TextView Forms Hex Auth Raw XML
Transformer Headers TextView ImageView Hex Auth Caching Privacy Raw XML
[Raw] [Header Definitions]

#	Result	Protocol	Host	URL
0	200	HTTP	www.fiddlertool.com	/fiddler/updatecheck.
1	200	HTTP	mail.google.com	/mail/channel/bind?at
2	200	HTTP	ajax.asp.net	/
3	200	HTTP	ajax.asp.net	/c/styles.css?v=56
4	200	HTTP	ajax.asp.net	/__utm.js
5	200	HTTP	ajax.asp.net	/__utm.gif?utmwv=6.
6	200	HTTP	static.asp.net	/asp.net/c/scripts.js
7	200	HTTP	ajax.asp.net	/i/bg_body.gif
8	200	HTTP	www.mscommunitie...	/CommunityBar.ashx
9	304	HTTP	www.mscommunitie...	/images/sites/aspnet.
10	200	HTTP	www.mscommunitie...	/images/bgd.jpg
11	200	HTTP	www.mscommunitie...	/images/sites/10.gif
12	200	HTTP	www.mscommunitie...	/images/sites/iis.gif
13	200	HTTP	www.mscommunitie...	/images/sites/silverligl
14	200	HTTP	www.mscommunitie...	/images/sites/mix.gif
15	200	HTTP	www.mscommunitie...	/images/sites/ch9.gif
16	200	HTTP	www.mscommunitie...	/images/arrow.gif
17	200	HTTP	www.mscommunitie...	/images/sites/msdn.gi
18	200	HTTP	ajax.asp.net	/WebResource.axd?d
19	200	HTTP	ajax.asp.net	/ScriptResource.axd?
20	200	HTTP	ajax.asp.net	/ScriptResource.axd?
21	200	HTTP	ajax.asp.net	/WebResource.axd?d
22	200	HTTP	ajax.asp.net	/i/bg_wrapper.gif
23	200	HTTP	ajax.asp.net	/images/ajax-poster-
24	200	HTTP	ajax.asp.net	/images/ajax-poster-
25	200	HTTP	static.asp.net	/asp.net/images/spac
26	200	HTTP	ajax.asp.net	/images/star-1-0.gif
27	200	HTTP	ajax.asp.net	/i/al-background-midd
28	200	HTTP	ajax.asp.net	/images/tsr9-l.gif
29	200	HTTP	ajax.asp.net	/images/tsr9-m.gif
30	200	HTTP	ajax.asp.net	/i/icon_li_bullet.gif
31	200	HTTP	ajax.asp.net	/images/go-about.gif
32	200	HTTP	ajax.asp.net	/images/go-download
33	200	HTTP	ajax.asp.net	/images/tsr-l.gif
34	200	HTTP	ajax.asp.net	/images/video2-1.jpg

1 / 57 ajax.asp.net/

Request Headers

```
GET / HTTP/1.1
Client
    Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-ms-application, application/vnd
    Accept-Language: en-US,it-IT;q=0.5
    UA-CPU: x86
    Accept-Encoding: gzip, deflate
    User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.0; SLCC1; .NET CLR 2.0.50727; Media Cente
Transport
    Host: ajax.asp.net
    Proxy-Connection: Keep-Alive
State
    Cookie
        __utma=237414864.1719050748.1170697149.1179489160.1179596660.101
        __utmz
```

Response is encoded and may need to be decoded before inspection. Double-click here to transform.

Response Headers

```
HTTP/1.1 200 OK
Transport
    Transfer-Encoding: chunked
Cookies / Login
    Set-Cookie: ASPNETUserSession=atlas.asp.net; expires=Tue, 22-May-2007 04:00:00 GMT; path=/
    Set-Cookie: AspNetWebRoles=4PctzHx7I2vP9LuzszJP72K-CUhMbDfEZbstGTuwZvgur-5THFPTv3wshY8vLOC
Cache
    Cache-Control: private
    Date: Mon, 21 May 2007 10:53:30 GMT
    Vary: Accept-Encoding
Entity
    Content-Type: text/html; charset=utf-8
    Content-Encoding: gzip
Miscellaneous
    Server: Microsoft-IIS/6.0
```

Figure 2.10 Fiddler is a free tool that lets you debug HTTP traffic.

Appendix B contains walkthroughs for configuring these tools and getting them up and running in a few minutes.

We'll end our discussion of debugging by returning to code. If debuggers do their best to help developers spot errors, we should ask what we can do—as programmers—to make debugging code easier. A good practice is to always raise meaningful and well-documented errors, as we'll explain in the next section.

2.4.7 Typed errors

In the .NET framework, you can raise either built-in or custom errors by leveraging the `Exception` class. For example, if you try to access an element beyond the bounds of an array, an exception of type `IndexOutOfBoundsException` is thrown at runtime. The purpose of this and the other typed exceptions is to carry information about the errors that occur at runtime.

To work with and detect exceptions, a programming language usually provides a special construct. For example, C# and VB.NET let you wrap a portion of code in a `try-catch` block. The `catch` block detects exceptions that are raised in the code encapsulated in the `try` block.

On the client side, JavaScript supports the `try-catch` construct and uses the built-in Error object as the base type to provide information about errors that occur at runtime. The Microsoft Ajax Library extends the Error object to make it possible to create typed exceptions on the client side. For example, to signal that a function lacks an implementation, you do the following:

```
function doSomething() {
    throw Error.notImplemented();
}
```

The `notImplemented` method throws a *client exception* that you can detect using a `try-catch` block. In general, an error can be raised anywhere in the code using the JavaScript `throw` keyword. To catch the error, you have to wrap the call to `doSomething` with a `try-catch` block:

```
function pageLoad() {
    try {
        doSomething();
    }
    catch(e) {
        alert(e.message);

        Sys.Debug.traceDump(e);      <-- Dump error object to console
    }
}
```

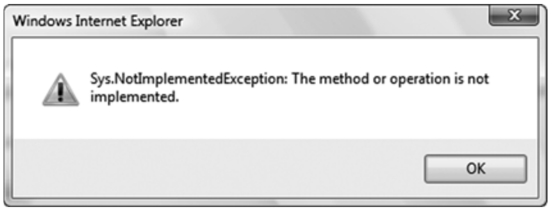
Every Error object captured in a `catch` block has a `message` property that indicates the nature of the error. Figure 2.11 shows the exception message displayed with a call to the `alert` function. If you run the previous example with the browser's console opened, you can also see a dump of the Error object captured in the `catch` block. Interestingly, the Error object also has a `stack` property that returns a string with the stack trace.

Note that, in the message box in figure 2.11, the error raised by calling `Error.notImplemented()` is reported as an exception of type `Sys.NotImplementedException`. The information about the exception type is stored as a string in the `name` property of the Error object. As a consequence, you can use the string returned by the `name` property to identify a particular exception in a `catch` block, as in the following code:

```
try {
    doSomething();
}
catch(e) {
    if(e.name == "Sys.NotImplementedException") {
        // Handle this particular exception.
    }
}
```

NOTE To browse the list of exception types defined by the Microsoft Ajax Library, consult the official documentation topic at http://ajax.asp.net/docs/ClientReference/Global/JavascriptTypeExtensions/ErrorTypeExt/default.aspx.

You can create custom exception types using the `Error.create` method. This method accepts two arguments: a string with the name of the exception type that the custom error represents, and a custom object whose properties are added to the Error object returned by `Error.create`. As a general rule, the custom object should contain at least a `name` property with the name of the exception type. Listing 2.9 shows an example of a custom error type created with the Microsoft Ajax Library.



Sys.NotImplementedException: The method or operation is not implemented.

Figure 2.11 An exception message displayed using typed errors

Listing 2.9 A custom error type created with the `Error.create` method

```
<script type="text/javascript">
<!--
    Error.myCustomError = Error.create('This is my custom exception
        message.',
      {
        name : 'Sys.MyCustomException',
        additionalInfo : 'Additional information about the error.'
      }
    );

    function pageLoad() {
        try {
            throw Error.myCustomError;
        }
        catch(e) {
            Sys.Debug.traceDump(e);

            alert(e.name + '\r\n' + e.message +
                '\r\n' + e.additionalInfo);
        }
    }
//-->
</script>
```

Custom properties attached to error object

Dump error object to console

Format error info

This listing defines a `Sys.MyCustomError` exception by using the `Error.create` method. The first argument passed to the method is the exception message. The second argument is an object with two properties: `name` and `additionalInfo`. The `name` property always contains a string with the exception type. The `additionalInfo` property is a custom property that should contain additional information about the error. If you need to add more properties, you can do so by expanding the object passed to the `Error.create` method.

The method returns a new function that you store in the `Error.myCustomException` property. This function raises the client exception. You call it in the `pageLoad` function, in the `try` block. In the `catch` block, you access the Error object and display the exception message in a message box onscreen. You dump the contents of the Error object in the browser's console. Note that you're able to access the `additionalInfo` property supplied in the custom object.

2.5 Summary

The Microsoft Ajax Library isn't just a library for performing Ajax requests. Instead, it provides a full featured framework for easily writing complex JavaScript applications. In this chapter, we have given a high-level overview of the library's features and explained the Application model, together with the concepts of client components and client page lifecycle.

One of the goals of the Microsoft Ajax Library is to make possible writing code that runs without incompatibilities in all the supported browsers. We have explored the compatibility layer, which is implemented with an abstraction API that turns library calls into browser-specific calls. The abstraction API takes into account DOM event handling, CSS, and positioning. Furthermore, the Microsoft Ajax Library allows you to easily create callbacks and client delegates in order to handle DOM events.

The Microsoft Ajax Library extends the built-in JavaScript objects to make them more similar to their .NET counterparts. The String object now offers formatting capabilities, and arrays can be easily manipulated. Furthermore, the Date and Number objects are enhanced to support globalization and localization. The library also provides a set of objects to perform common tasks in JavaScript applications, from fast string concatenations to browser detection and support for debugging.

In the next chapter, you will see how the Microsoft Ajax Library makes it easier to program in JavaScript using object-oriented constructs such as classes, interfaces, and enumerations.