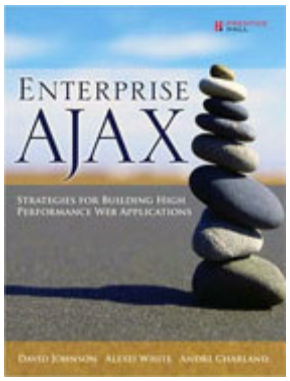
Enterprise Ajax

Sample Chapters

Strategies for Building High
Performance Web Applications



AJAX enables web pages to work faster and more intuitively. It allows Web applications to look and feel more like traditional applications. Web applications using AJAX have already started to replace standard desktop applications, generating a growing interest in how AJAX can actually make in-roads in the enterprise. **Enterprise AJAX** addresses this trend by going beyond the basics to show real results from AJAX performance benchmarking, user interface interaction testing, and customer implementations.

This book contains valuable AJAX architectural patterns as well as case studies from real enterprise applications and examples based around both .NET and Java, the two dominant enterprise application frameworks. The authors highlight important enterprise AJAX techniques and present them in the context of the Gang of Four design patterns. The topics in this book are extensions of traditional enterprise development patterns, but amplified to leverage the power of AJAX.

- Provides a concise introduction to AJAX fundamentals
- Looks at the details of some of the popular AJAX frameworks
- Shows how to build component-based AJAX applications
- Helps understand the role of AJAX in the future of the enterprise by looking at competing and complementary technologies
- Discusses advanced topics such as declarative JavaScript programming

Enterprise AJAX is geared towards developers of large scale systems who want to try out this new development methodology, whether to migrate legacy HTML interfaces to AJAX or to build new applications not possible before AJAX. At the same time, information architects, user-interface designers, and usability engineers will turn to this book to increase the performance of existing user interfaces and to ensure distributed applications run as intended.

Authors:

Dave Johnson - CTO / Co-Founder, Nitobi

Alexei White – Product Manager, Nitobi

Andre Charland – President / Co-Founder, Nitobi

We're releasing this sample chapter so can get a sneak peak at our upcoming book and provide us with feedback and ideas! Please email us at enterpriseajax@nitobi.com with any thoughts you have!

Chapter 3
Ajax in the Web Browser

In the previous chapter we have learned about all the core technologies that comprise the Ajax "stack". We also learned a little bit about how these components of Ajax fit into the traditional *Model-View-Controller (MVC)* paradigm which will enable us to think critically about the architecture of an Ajax application in a common and familiar way. In this chapter we are going to learn how to integrate Ajax into our web applications.

We'll first discuss how to get your Ajax code up and running using several different browser dependent techniques. Once we know how to bootstrap our JavaScript code we will take a closer look at implementing a MVC pattern in the web browser. We will see how to start with a simple Model for storing data, then look at how to render the data as a View and finally introduce user interaction with the Controller. This will include further investigation into JavaScript inheritance as well as a how to build a re-usable cross-browser event module – which is likely the most important feature of any Ajax application. Furthermore, we will learn to recognize and leverage some fundamental software development patterns such as the Observer pattern.

To frame the ideas we develop in this chapter we will be creating a sample Ajax application that we can use for managing customer and order information. By the end of this chapter you should have the skills to build a simple Ajax application that follows the tenets of MVC. Although following the MVC pattern will not be simple, the effort we put towards designing our software according to MVC should pay dividends when it comes time to test our code and even more so when our thoroughly tested code can be re-used in the future.

Component-Based Ajax

The canonical examples of Ajax applications are the mapping application Google Maps and the popular photo sharing site Flickr. While these two examples are superlative applications, they are not applications that are necessarily familiar territory to the average business user. Although today much of the hype around Ajax has to do with consumer-facing applications, the true calling of Ajax is likely going to be in enhancing the functionality and user experience in critical business-line applications used within the enterprise.

In the enterprise, customer relationship management (CRM), enterprise resource planning (ERP) and business intelligence (BI) are just a few of the types of applications that get used on a daily basis and can often have sluggish, and unintuitive user-interfaces. However, this does not have to be the case. One common thread throughout all of these types of business applications is that they are built from various basic user-interface components. These basic user-interface components include datatables for viewing large numbers of records, web forms for editing data, charts for visualizing data, and data search or filtering capabilities. All of these aspects of a user interface can be considered modules of an application that can be used independently in a larger web application framework such as JSF or ASP.NET. An example of one of these core components is the ASP.NET DataGrid control, which can be used to view and edit tabular data - this data could be anything ranging from customer records to support incidents to product sales information.

Focusing on these common business use cases around searching, listing and editing data, in this chapter we are going to lay a foundation from which we can build Ajax based components that can be used as modules in a larger web application architecture.

While the ASP.NET DataGrid control is heavily dependent on all the plumbing provided by the .NET framework on the server and is thoroughly unusable, Ajax based components, can be server agnostic and depend only the technologies available in the web browser. Data in an Ajax application or component, as long as it is provided in the expected data format, can be served from any type of server whether it is running PHP, Java or Ruby on Rails. Similarly, building and using Ajax components doesn't mean that you have to throw away current web application architecture, development tools, technologies, and methodologies. On the other hand, choosing a "single-page" approach to Ajax – one in which the web page never refreshes but instead relies solely on XHR request to access data on the server – can require a significant amount of work to convert an existing application over to use Ajax. A single-page approach to Ajax can be prudent if an application is being re-built from the ground up, however, if you want to preserve as much of a current web application as possible a component based approach could be more advantageous since it lends itself well to incrementally introducing Ajax functionality into an application.

Incremental Ajax

Today many web applications already use pre-built components in their development whether it's through .NET Web Controls, or JavaServer Faces. Even if only partially, incorporating Ajax techniques for any number of the components currently in use in your web applications is possible and can result in a better user-interface and happier end-users. The component approach allows developers to make incremental changes to an application and introduce Ajax functionality only when it is beneficial to do so. Small parts of an application can be enhanced with Ajax while leaving the majority of an application in its legacy technology. To move to a completely Ajax based architecture requires careful planning and rethinking of the role of the server because the main challenge of moving to Ajax component based user-interfaces is changing from primarily server based programming to primarily client based programming (in JavaScript, DHTML and CSS). Having said that, the client-side functionality of an Ajax component can quite easily be encapsulated in server side code either by hand or by using any one of the handful of server based Ajax solutions. For example, an Ajax enhanced JSF tree control can be built such that the current knowledge of Java programmers can be leveraged, thus easing server integration even further.

Impact on the Server

The role of the server changes significantly when we start to look at Ajax based applications. Traditionally the server was responsible for rendering the View and streaming this up to the client as well as responding to events that occur on the client – this second point is the most salient. When using Ajax the server is still responsible for rending some aspects of the View, such as `<script>` elements to include the appropriate JavaScript libraries and any custom HTML tags such as `<DOJO:button/>` to have a button rendered on the client by the Dojo framework. The important thing to recognize is that any events occurring on the client are no longer required to trigger an HTTP request to the server to have a new View generated on the server and streamed up to the client. With an Ajax component, events that occur on the client, such as when an item in a Listbox is selected, do not necessitate an entire page refresh but instead can directly update other components on the web page or request additional data from the server behind the scenes. If we think of this in terms of MVC, one can see that all three pieces of the MVC pattern can exist on the web page itself rather than needing to span the network from the client all the way to the server as we will see shortly. These changes can significantly reduce workload on the server and

also help you design the server architecture in a more modular way that can be adapted to any RIA in the future.

Ajax makes it possible to create entire web applications as well as small components in applications with real value, as opposed to just cool technology – even though there are plenty of opportunities to bring innovative ideas into even the stodgiest of businesses. Let's look at how we can build a component-based Ajax application starting with the HTML document itself.

HTML Standards

Ajax has a lot to do with standards. Most of the technologies are supported by the W3C and many developers appreciate the fact that they are working standards based technologies. This is a big reason that proprietary technologies such as Adobe Flash do not get the same mind share amongst Ajax developers as it possibly should. However, given the current web browser landscape (and that of the near future), working in a standards-based world can still be something of a challenge. For the most part, when we talk about problems with standards adherence, we are speaking of Internet Explorer. Many areas of Internet Explorer are based on proprietary interfaces defined by Microsoft before there were any web standards that could apply. The Internet Explorer DOM Events and CSS styling implementations are different from the W3C and can cause a few headaches. Having said that, we have already come across a few areas where the foresight of Microsoft has resulted in de facto standards (`innerHTML`) or even techniques that have been adopted by the W3C (XHR). Let's take a look at some of the important differences between W3C HTML standards adoption in different web browsers.

Document Type Definitions

One of the first things that any web developer does when assembling a web page is chooses a document type definition or DOCTYPE for short. The first advantage of specifying a DOCTYPE on your web pages is that you can validate the contents of the page using a validation service like that provided by the W3C (http://validator.w3.org/) to ensure that the contents adhere to the given standard and don't have any broken markup – which can be a nasty bug when it comes to Ajax and dynamic DOM manipulation. Validation is often ignored by developers, however, it can be a good practice to help improve your web pages. Producing valid HTML or XHTML can have other side advantages such as improved search engine optimization (at least they make sure you have your `<title>` tags) to faster HTML parsing and better cross-browser performance. It can also help with accessibility (but not necessarily as we will see in Chapter 10). The most significant impact of specifying a particular DOCTYPE is to indicate to the web browser how to interpret and parse the HTML content. Depending on the DOCTYPE, CSS will be interpreted and HTML rendered in different ways. Although it is something most commonly discussed in conjunction with Internet Explorer, most web browsers support two modes of operation, "quirks mode" and "standards mode". In fact, most browsers even have a third "almost standards" mode (arguably what most people call standards mode for Internet Explorer) but we won't worry about that too much since there are so few and rarely noticed differences between this and regular standards mode.

To determine which mode of operation to work in, web browsers support DOCTYPE switching. What that means is that they change how they interpret and render the HTML and CSS contents of a web page based on the page DOCTYPE declaration. For example, if there is no DOCTYPE specified in a web page all web browsers operate in quirks mode. There are essentially six primary DOCTYPEs that one needs be concerned with as listed below.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
"http://www.w3.org/TR/html4/frameset.dtd">

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
```

The first three listed are the strict, transitional, and frameset versions of the HTML 4.01 DOCTYPE and the last three are the respective versions for XHTML 1.0. The main difference between HTML 4.01, which is often referred to as "tag soup", and XHTML 1.0 are list below:

	HTML 4.01	XHTML 1.0
Document should be well formed	Customers 	Customers
Attribute names should be lower-case and in quotes	Customers	Customers
Script elements need CDATA	<script TYPE=text/javascript> …</script>	<script type="text/javascript"> <![CDATA[…]]></script>

While many developers are tempted by the allure of "proper" XML based HTML tags it is often only met with problems. The main problem to note is that XHTML content should be, according to the specification, delivered with the `application/xhtml+xml` mime-type. The problem is that Internet Explorer does not yet recognize this mime-type. There has been one improvement in Internet Explorer 7 in that placing an XML prolog (`<?xml version="1.0" encoding="UTF-8" ?>`) before the DOCTYPE declaration no longer causes the browser to revert to quirks mode as it did in Internet Explorer 6. By specifying the XML prolog we can get Internet Explorer 7 to think that the content is XHTML rather than just HTML. It is a large step on the part of Microsoft on their path to fully support XHTML. XHTML documents do provide some advantages such as making it more straightforward to include other XML based languages (with namespaces) such as SVG and invalid XML will throw an error in the web browser. Using XHTML is still a bit optimistic and will only really become a reality when Internet Explorer adopts the `application/xhtml+xml` mime-type.

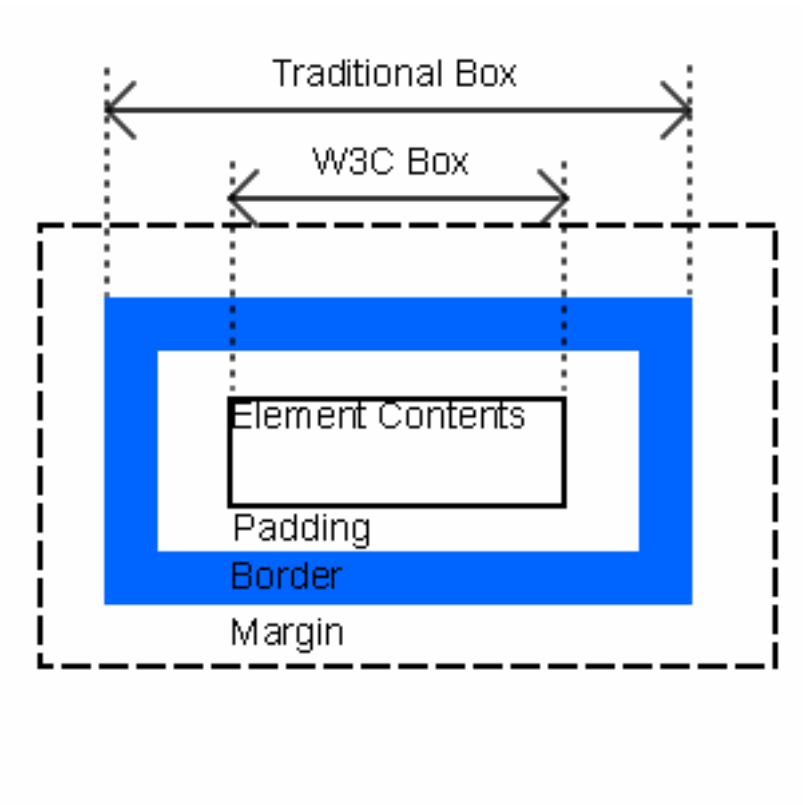
For most browsers, except for a small few like Konqueror [http://hsivonen.iki.fi/doctype/], all of the HTML and XHTML DOCTYPE definitions listed above will switch the browser into either almost or pure standards mode – which for the most part are the same. The "strict" DOCTYPE will put browsers like Firefox, Opera, and Safari into standards mode and Internet Explorer into

almost standards mode while the “transitional” DOCTYPE will put most browsers into almost standards mode.

Box Models

While this discussion of DOCTYPEs and XHTML may seem all well and good, we are going to cut the chase. For most browsers, such as Firefox, quirks mode has very few differences when compared to standards mode. Internet Explorer does not fare so well in this respect and is really the issue that lies at the heart of DOCTYPE switching. In Internet Explorer there is one major difference between quirks and standards mode and that is the CSS box model. The CSS box model has to do with what the dimensions of an HTML element represent. In the case of the W3C standard, the width and height of an HTML element corresponds to the width and height of the *contents* of the element, which is used in both quirks and standards mode by all browsers other than Internet Explorer. On the other hand the traditional model is still employed by Internet Explorer in quirks mode where the width and height of an HTML element corresponds to the width and height of the outside of the element – the outside of the element includes both the element padding and the element border. Depending on the situation, either of these points of view can be useful. In some cases one may want to be aligning elements inside other elements – in which case the W3C model is useful – and in other situations one may want to align elements next to each other – in which case the outer dimensions are important and the Traditional model makes more sense.

Figure 3.1
Elements of the Box Model

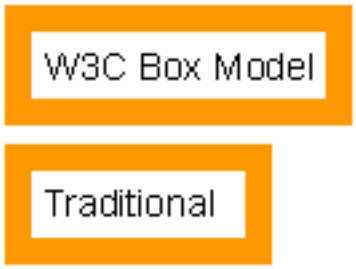


The point is that if you want to reduce your Ajax-induced box model headaches, you should adopt a strategy for dealing with the box model problem. There are essentially two options.

1. Use standards mode in Internet Explorer by specifying one of the DOCTYPES listed above, in which case web pages will use the W3C box model like all other browsers.
2. Use quirks mode in Internet Explorer by having no DOCTYPE and force other browsers to use the Traditional box model with the `box-sizing` CSS rule.

The CSS3 specification defines a rule called `box-sizing`, which can take on two values, `border-box` and `content-box` corresponding to the Traditional and W3C models respectively. Currently, Opera and Firefox both support this and therefore allow one to choose the box model depending on the situation. In Firefox the syntax for this property is slightly different with `-moz-` prepended to the rule and it also supports a `-moz-box-sizing:padding-box` value which should be self explanatory. An example of two boxes in the W3C and Traditional models are shown below with no margin, a 10 pixel border, 5 pixels of padding, and a specified width of 100 pixels. However, one can see that the W3C box is actually 130 pixels to the outside of the orange border (100 pixels *content width*, 10 pixels padding, and 20 pixels border) while the Traditional box is 100 pixels *total width* to the outside of the orange border irrespective of any padding or border widths.

Figure 3.2
Traditional Versus W3C Box Model



The box model is just one of, although the most important by far, the many differences between quirks and standards mode operation. Now that you have a feel for the implications of your DOCTYPE lets look at how to start our Ajax code in the browser.

Bootstrapping Ajax Components

The Ajax component life cycle begins as soon as the web page is first loaded. It is at this time that we need some way of initializing our Ajax components. Compared to a .NET Forms or Java application, bootstrapping an Ajax application is a little more difficult. There are a few different nuances that one must be familiar with when instantiating their application. The most important thing one needs to consider is the fact that any inline JavaScript, that is any JavaScript not contained inside a function, will execute as soon as the script engine finds it. What that means is that the JavaScript could execute prior to the loading of other parts of the web page that the JavaScript code could be referencing. For example, if some inline JavaScript at the start of a web page attempts to access an HTML element using the `$(elementId)` method then it will likely return `null` not because the element with that ID doesn't exist but because the HTML content of the web page has not had time to be parsed and loaded. There are several different ways to instantiate your Ajax application in which you can be certain that all the resources your application may require have been loaded and parsed. The most common way of ensuring this is to use the `window` or `<body> onload` event.

The Onload Event

The `onload` event of the browser `window` object should be familiar to many of you that have used JavaScript before, though for the benefit of those who haven't seen this and for the sake of a general review for the rest of you, let's consider this brief refresher. The `onload` event, a member of the global `window` object, allows one to specify a JavaScript function that is to be executed after the entire page, including the HTML tags, images, and scripts, have been downloaded to the browser. Unlike most events, the event handler for the `window.onload` event can be specified using explicit attachment through JavaScript or alternatively it can be specified explicitly in the HTML content; since this event is fired when the HTML DOM has been loaded and parsed, we cannot attach the event using the DOM event system. The syntax for specifying the `onload` event through JavaScript looks something like this:

```
window.onload = init;
```

where `entAjax.init` is a JavaScript function reference that will be executed once the page has loaded. There are two drawbacks to using this method for your Ajax application. The first drawback is that by setting the `window.onload` event we might be overwriting the event as defined by a different component on the page. This problem of inadvertently overwriting some event or object that someone else has already defined can is becoming increasingly important with the growing focus on JavaScript mashups – merging two online applications such as maps and photos to create a new application – as we will discuss later. In order to bootstrap multiple Ajax components using `window.onload`, we need to create a single proxy JavaScript function in which all of the Ajax components for our page are bootstrapped. Commonly a function named `init()` is used for this purpose as shown below where we attach a handler to the `onload` event, though there is not much we can do quite yet:

```
<html>
  <head>
    <script type="text/javascript">
function init() {
  alert($("myComponent").innerHTML);
}
// Set the onload event function reference to our Init function
window.onload = init;
    </script>
  </head>
  <body>
    <div id="myComponent">My first component goes here.</div>
  </body>
</html>
```

The second drawback of the `onload` event is that our Ajax components will only be loaded after the *entire* page has been downloaded. This means the browser has to wait for all resources (including externally linked Images, CSS and JavaScript), to be downloaded to the client. There is a potential for things to break if there is an error in downloading the page, or if the user abandons the download by clicking on the "Stop" browser button. With the `onload` approach, there is a small chance that our Ajax components may take a long time to activate, or in the worst case, never be bootstrapped at all.

Being a Good Neighbour

If your JavaScript is running on a web page that is also running other components or JavaScript frameworks, you have to be careful to play nicely with the document events. Since almost all JavaScript depends on the `onload` event to start the bootstrapping process, it can be a bad idea to blindly overwrite the `window.onload` event, which may be being used by other JavaScript code already. This can happen most often when two people are working on different code or components for a single web page. Below is a simple web page with two included JavaScript files that might come from different authors, each written to instantiate a different component on the web page like a customer list and product list.

```
<html>
  <head>
    <script type="text/javascript" src="customers.js">
    <script type="text/javascript" src="products.js">
  </head>
  <body>
    <div id="customerList">HTML element for Customers.</div>
    <div id="productList">HTML element for Products.</div>
  </body>
</html>
```

The first include (customers.js) might look like this:

```
var example1 = {};
example1.init = function() {
  alert($("myCustomers").innerHTML);
}
window.onload = example1.init;
```

While the second JavaScript include, for the purposes of this example, looks pretty much identical but it refers to the other HTML element.

```
var example2 = {};
example2.init = function() {
  alert($("myProducts").innerHTML);
}
window.onload = example2.init;
```

Like good programmers, in this example we have put each of the initialization functions in their own namespace; otherwise, we would have had bigger problems on our hands with functions being overwritten. Instead we just have the problem that the second JavaScript file included completely overwrites the `window.onload` event meaning that the first initialization function will never be called. It is always best practice to assign methods to common event handlers in a non-destructive manner. Non-destructive event handler assignment, or function assignment in general, can be achieved by copying the old function and creating a new anonymous function encapsulating both the old and new function. For example both of the includes should have attached their event handlers to the `window.onload` event like this:

```
var example2 = {};
example2.init = function() {
  alert($("myProducts").innerHTML);
}
// Store any previously defined onload event handlers
var oldLoader = window.onload || function() {};
// Create closure that calls example2.init and previously defined meth-
ods
window.onload = function() {
  example2.init();
  oldLoader.call(this);
}
```

What we have done is saved a reference to the old `onload` handler in the `oldLoader` variable and then set the `window.onload` event to an anonymous function, which calls both our new `onload` handler called `LoadMyComponent` as well as the old `onload` event handler. The only reason that this works is due to the magic of JavaScript closures, unless of course this code is running in the global scope and `oldLoader` is a global variable. Either way, the `oldLoader` variable, although defined outside of the anonymous function, is still accessible when the anonymous function is executed after the page has finished loading.

Mixing Content and Functionality

That almost concludes our discussion of the `onload` event. However, just for completeness we should probably mention that there is one approach to using the `onload` event that should be avoided. Like any other HTML event the `onload` event can be specified as an attribute on the HTML `<body>` element such as:

```
<body onload="example1.init();">
```

Notice that, unlike attachment of the event handler through JavaScript, we have actually written a proper call to the handler function with the trailing brackets – when specifying events directly on HTML elements the contents of the attribute should be valid JavaScript code that is evaluated by the web page. Although tempting, this is generally frowned upon since not only does it blur the lines between the presentation and the functionality of the web page but the `onload` event on the `<body>` element will override any other `onload` events specified on the `window` object, making it very difficult for other JavaScript components to bootstrap themselves.

Browser Tricks

`onload` is the standard, run of the mill way of bootstrapping an Ajax application and, in general, client-side JavaScript. As we already mentioned, the `onload` event can take a long time to fire if there are large images and the like to be downloaded by the web page. If one wants to make an application load up slightly faster and provide the end-user with a correspondingly improved experience, what we really want to achieve when bootstrapping our Ajax application is to have the application load as soon as possible such that the end-user spends as little time waiting as possible. Of course, the problem here is that Ajax components may depend on the web page DOM or HTML elements to be accessible (ie loaded and parsed by the web browser) before they can be loaded. If those dependent HTML elements are not parsed when the Ajax component is initialized then we can run into problems. This means that we have to take advantage of some

browser tricks to make our JavaScript run immediately after the DOM has been loaded and parsed but just prior to other extraneous resources like images are loaded. This saves time for our end-users and improves the entire user-experience.

Fortunately, there are several well-known methods that we can use to eliminate these problems from our applications and we will look at them now.

Script Placement

The easiest but most fragile method of getting your scripts to run sooner is by placing the actual `<script>` element in the web page after any HTML elements that are required for the Ajax program to be initialized. While it is a simple and fully cross-browser compliant solution, most people deride it due to its blatant dependency on the precise placement of the `<script>` element, something that has no relation to the display of the web page, in amongst the HTML that is considered to be the MVC View. For example, below is perfectly acceptable inline JavaScript since the JavaScript reference to the HTML DOM is positioned inline after the HTML element to which it refers.

```
<html>
  <head></head>
  <body>

    <!--This HTML element is accessible to scripts below, not above-->
    <div id="CustomerName">John Doe</div>

    <script>
var customer = $("CustomerName");
// This is just fine inline JavaScript
customer.style.color = "blue";
    </script>
  </body>
</html>
```

DOMContentLoaded

Mozilla-based browsers provide a handy, and mostly undocumented, event called `DOMContentLoaded`. The `DOMContentLoaded` event allows us to bootstrap our Ajax components after the DOM has loaded and before all the other non-DOM content has finished loading. Using the DOMContentLoaded event is much like any other event and is registered like this:

```
if (document.implementation.createDocument){ //Check for FirefoxMozilla
  document.addEventListener('DOMContentLoaded', Example.init, false);
}
```

This is a very simple way to provide a slightly faster application. Of course keep in mind this only works in Mozilla based browsers and another solutions needs to be found for other browsers.

Defering Scripts

As might be expected, Internet Explorer requires that we find a different way to achieve the same result. Internet Explorer supports a few proprietary features to help us ensure our scripts load when we want them to. The first is the `DEFER` attribute which can be used on `<script>`

elements. When a `<script>` element has a `DEFER` attribute on it, Internet Explorer delays running the JavaScript referred to by or contained in the `<script>` element until the full web page DOM has been parsed and loaded. In essence, the `DEFER` attribute is completely analogous to placing the `<script>` element at the very end of the document. There is one problem that we need to be aware of, the `DEFER` attribute is of course ignored by non-IE browsers, which means that your code will execute immediately as the page is still loading. Luckily, there is another trick that we can employ in Internet Explorer to help with this problem. Internet Explorer supports conditional comments that enable one to specify HTML content that is conditionally uncommented depending on some condition. An example of a conditional comment that includes a certain JavaScript file if the web browser is Internet Explorer is shown below:

```
<!--[if IE]><script defer src="ie_onload.js"></script><![endif]-->
```

To any non-Internet Explorer browsers the previous HTML code will simply be commented out and thus not load and run, however, in Internet Explorer the conditional comment will be recognized and will evaluate to true. The result of this conditional evaluating to `true` is that the contents of the comment will actually be loaded with the DOM and the `<script>` element with the `DEFER` attribute to delay processing. Another option here is to use conditional compilation, which essentially allows code to be invisible to all browsers except for Internet Explorer like this:

```
<html>
  <head>
    <script type=”text/javascript”>
/*@cc_on @*/
/*@
  alert("You are running Internet Explorer");
@*/
    </script>
  </head>
  <body>…</body>
</html>
```

The script above will show the alert if run in Internet Explorer. The `@cc_on` statement turns on conditional compilation and the `/*@` indicates the beginning of a conditional script block with `@*/` indicating the end of a conditionally compiled script block.

Quirky Results

Just to make sure we cover all our bases, another method that many people try is using the `document.onreadystatechange` event in Internet Explorer. Sadly, this has been shown to behave rather unreliably, working on some occasions and not others, sporadically therefore we suggest people steer clear of this event. Instead, to make sure the DOM has loaded in any browser then you can have a code loop using `setTimeout()` which checks if a certain DOM element is available by using `$()`.

```
function domReady(nodeId)
{
  // start or increment the counter
  this.n = typeof(this.n) == 'undefined' ? 0 : this.n + 1;
  var maxWait = 60;
```

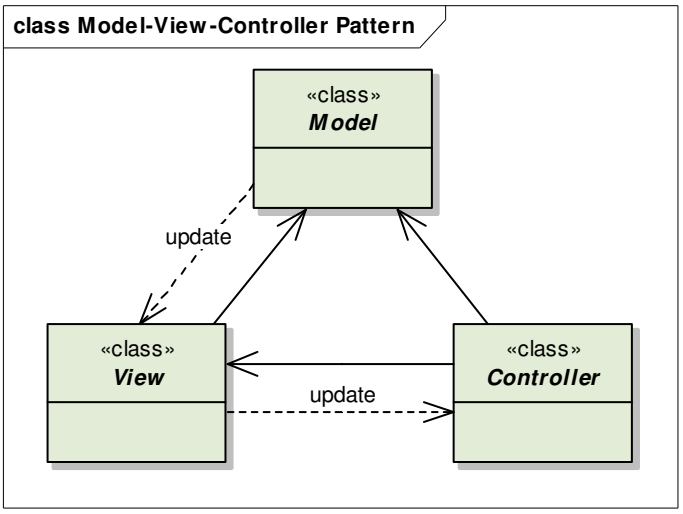
```
  if (typeof($) != null && $(nodeId) != “undefined")
  {
    alert("The DOM is ready!");
  }
  else if (this.n < maxWait)
  {
    setTimeout(function(){domReady(nodeId)}, 50);
  }
};
domReady(“myDomNodeId”);
```

This is undoubtedly the most straightforward way of determining when the DOM is ready. The only trick is that it must know the DOM node that needs to be loaded and cannot take into account deeply buried dependencies between JavaScript components and other HTML elements. A surefire way to load your JavaScript in a hurry is to have a known HTML element placed immediately prior to the closing `<body>` tag. Now that we have looked at some of the issues surrounding getting our Ajax applications off the ground, let’s continue by discussing how we can leverage a purely client-side Model-View-Controller pattern in our Ajax applications.

Model–View–Controller

Going forward, we will focus on the MVC pattern as a guideline for building our Ajax components and applications. The MVC pattern is usually utilized in the context of the client and server with the View being implemented on the client and Controller and certainly the Model being encapsulated on the server. However, once we start dealing with Ajax based applications much of the three aspects of the MVC pattern can be implemented on the client, with the exception in most cases of Model persistence.

Figure 3.3
An MVC Based Architecture



The diagram above is a simple example of an MVC based architecture. The solid lines represent what could be thought of as explicitly coded arbitrary method invocations while the dashed lines are method invocations on objects that are assumed to implement a particular abstract Model, View or Controller interface. The important thing to notice here is that the Controller usually has explicit connections to both the Model and the View. That is to say that the Controller will have access to specific methods that can be called on the Model or View; likewise, the View will usually call methods of the Model. On the other hand, the View and Model should ideally have no connections to the Controller or the View respectively. Of course this is an ideal view of MVC and we will be fairly liberal in its application when the rubber hits the road. Let's take a look at how MVC can align with Ajax based development and construct a simple Model, View, and Controller.

View

Since the dawn of the Internet, the web browser has been the traditional territory of the MVC View so we will begin our look at Ajax MVC there. The View is responsible for displaying the user-interface of our Ajax component by using the HTML DOM and CSS as well as laying the groundwork for the wiring that will allow users to interact with the application. One can think of the View as being an automated teller or cash machine (ATM); through what is supposed to be the simple interface of the ATM one is able to interact with their bank, a large and most times highly complex entity. Through the ATM interface one can perform various operations on your bank account like finding out what the balance is, depositing money into it, and withdrawing money from it. All of these are intricate and complicated processes yet presented to the user in what is meant to be an easy and understandable manner. If we look more closely at the MVC architecture, there are several specific tasks that the View is generally responsible for, such as:

Rendering the Model

Requesting updates from Model

Sending user-gestures to the Controller

Allowing the Controller to select a View

When an Ajax component is initially rendered in our application, the View is responsible for doing the work of presenting the data in our Model to the user. The actual process of rendering the data from the Model requires the data to be merged with some sort of HTML based template to result in the data being formatted in a user-friendly manner. Generation of the merged data with the HTML formatting can be achieved through a few different means. The simplest way to create the View is by concatenating a string of data and HTML markup, which is subsequently inserted into the web page by using the `innerHTML` property of some HTML element.

```
<html>
  <head>
    <script type="text/javascript">
var example = {};
example.init = function() {
  var s = [];
  var customers = ["Jim","Bob","Mike"];
  for (var i=0; i<customers.length; i++) {
    s.push("<div class=\"customer\">"+customers[i]+"</div>");
  }
  $("customerList").innerHTML = s.join("");
}
window.onload = example.init;
```

```
    </script>
  </head>
  <body>
    <div id="customerList"></div>
  </body>
</html>
```

This is the most common and straightforward mechanism for building the HTML of an Ajax application on the client. Alternatively, the HTML can be generated on the client using various templating techniques such as pure JavaScript, JavaScript and DOM manipulations, or using Extensible Stylesheet Language Templates (XSLT) – though XSLT is currently only available in the latest builds of Safari (build 15836), which won't be in wide use until the next version of OS X is released. One other alternative is to generate the HTML on the server and send View HTML snippits rather than Model data up to the client, which certainly reduces the difference between traditional web applications and Ajax applications; however, rendering the View on the server and passing HTML snippits over the wire can significantly hinder any advantages that Ajax provides. We will look at many of these different approaches to generating the View as we re-factor our Ajax component architecture throughout the book.

Once the View has been rendered, using which ever method we choose, the View can also explicitly request updates from the Model. In an Ajax application these updates can occur quite frequently thus ensuring that the data presented to the end-user in the View is valid and relevant at all times; this sort of up-to-date information in the View is mostly unheard of in a traditional application and provides significant value to the end-user. When data from the Model is required on the client, it may be requested from the Model on the client where the data has been preloaded or it may be the case that the data needs to be requested from the server using the `XMLHttpRequest` object. In either case, as we will discuss in a moment, the Model should be abstracted to the extent that when writing the client JavaScript code for an Ajax application it not matter care if the data is on the server or the client.

Requesting updates from the Model may occur as an intrinsic part of an application – such as when the application initially loads – or as a result of gestures by the user during the lifetime of an application. Gestures by the user are captured by the View and subsequently connected from the View to the Controller of our MVC architecture using DOM Events as, once again, we will see very shortly. The connection between the View and Controller is clearly a very important aspect of the MVC pattern. It is a wise idea to make the connection between the View and Controller as general as possible since this enables efficient division of work in terms of splitting up the user-interface and application logic. Furthermore, by connecting the View to the Controller in a generic fashion one is able to make changes to the Controller with much more confidence that it will not break the application. One could even replace a Controller with a completely new one and have few changes to the application design. Determining these interfaces between the components of the MVC architecture is paramount as it increases the code reusability as well as makes changing any component of architecture less error prone and much easier. Of course the other important thing to remember when discussing efficient separation of the MVC architecture is the benefits we find in terms of testing. By loosely coupling the various concerns testing is made much easier and gives us peace of mind when making changes to your application. We will discuss some of these advantages such as testing later.

Finally, the View must provide an interface that allows the Controller to select or make a certain View active. This is certainly important since there are many situations where the Controller must take action on some gesture from a user, such as clicking on a button, which results in a new View being created or a visible View becoming active. We will see in a moment where this sort of interface is required in the context of the interactions between the View and Controller.

Controller

Once we have implemented the View of our component, which is likely the most straightforward aspect, we need to create a Controller that acts as, in our ATM example, the actual ATM computer. The Controller of the ATM is responsible for choosing things like which operations to present to the user – ie which View to present – and taking certain actions depending upon the operation that the user selects through the View. If a user wants to check their account balance they might see a list of options on the ATM screen and when the "Check Balance" button is pressed a message is sent to the Controller with information about the action in a generic manner. In response to this request, the Controller retrieves the appropriate data from the Model and presents it in a different View. So the Controller essentially defines how our application's interface behaves and changes in response to user-gestures. In a more general sense, the Controller is responsible for:

Defining application behavior

Mapping user-gestures to model updates

Selecting or updating the view in response to user-gestures

Every time a user interacts with an application the Controller is responsible for deciding what to do, it essentially defines the behavior of the application. Whenever any user-gestures are made they are handled by the Controller. When any Views need changing they are directed by the Controller. When our Ajax component is loaded it is the Controller that decides which View to present and with what data from the Model. Using JavaScript for the Controller, one can create the logic that determines how the application will behave under different conditions and in response to various user-gestures.

In response to user-gestures, the Controller also acts as a conduit for data to flow from the View down to the Model where it can be persisted to a database and also trigger updates to other Views. Depending on the application, the Controller will need to be prepared to receive various types of event requests from the View that can change information in the Model related either to application state or actual data that the application is manipulating. In an Ajax application the user-gestures result in events being fired through the DOM Event model. As we saw in Chapter 2, the DOM Event model enables one to connect events such as mouse clicks and key presses from different DOM elements to different event handlers in the Controller layer of the MVC architecture.

In particular, user-gestures often result in changes to the View meaning that the Controller needs to be able to either change the active View or enable a completely different one depending on the user interaction.

Model

So far so good. The only problem is we have little idea of where the data for our application actually comes from. At this point, we know that when the user at the ATM wants to find their account balance they need to press the appropriate button in the View which then alerts the Controller that some action is to be taken. When this happens, the Controller needs to retrieve the actual account balance from the Model, which in the case of an ATM is likely some mainframe computer system. That is the place where our application's business logic lives, where money changes hands, and checks and balances ensure that everything is done correctly. When our Controller is asked to show the account balance it will make the appropriate request for that data from the Model. Once the data is returned from the Model we can then use the Controller to update the View with the returned data. When all is said and done, the responsibilities of the Model can be distilled down to the following areas of an Ajax component:

Encapsulating application state

Exposing application API

Notifying the View of changes

Paramount to any Ajax component is the management of the component state. The Model is responsible for providing an interface to through which application state can be manipulated and queried. This can include both pure data, such as the user account balance, as well as *metadata* about the application, like transient information about the current session. We will explore the idea of an MVC Meta-Model later.

Since the Model encapsulates the data of our application it also needs to be available to be queried about that data – otherwise it would not be all that useful. The Controller generally needs to be able to perform a few common operations on data such as create, read, update, and delete, which are commonly referred to as CRUD (Create, Read, Update, Delete). All of these operations exposed by the Model are leveraged by an application to enable a well defined separation of the View and Controller from the Model. At the same time, having the Model as a clearly separate entity from the other aspects of an application allows other parts of an application to both access and manipulate the Model data and metadata available from the Model in other contexts. For example, the same API exposed by the Model on the mainframe computer storing your bank details can be used to access your account information from either an ATM machine or through an internet banking interface.

Although in the ATM analogy the Model likely has little knowledge of the View, in Ajax applications it is fairly important that the Model has the ability to notify the View of changes to the Model that may have occurred. This is particularly beneficial when you have several different Views displaying data from the same Model; in this case changes to data initiated through one View can automatically be propagated to all the other Views that are interested in the Model information. This is a much more difficult goal to achieve in a traditional application where the updates to the View from the Model be accompanied by a lengthy page refresh for the end-user. We will later see some techniques for separating explicit bindings between the Model and View which can better enable us to stick to a cleanly separated MVC architecture.

Ajax MVC

Now that everyone has a common foundational idea of the MVC pattern and how it might look when applied to a real world situation. Let's now take a closer look at how the MVC pattern might be leveraged in the context of building an Ajax component that can be re-used throughout your enterprise applications.

Ajax Model

While reviewing the various aspects of the MVC pattern it was useful to start from the more tangible View and work back to the obscure idea of the Model. However, when we start applying MVC to an Ajax component it is more instructive to start from the Model and move up to the View. Doing things in this order usually makes it easier to write our code in a test-driven approach - where we write the tests before the code is written and the code should ensure that the tests pass - and we can build on each layer with the next. Logically, the first thing that we need when developing an Ajax application is access to our data. In fact, retrieving data from the server is the only part of an Ajax component that actually uses "Ajax" – the majority of the what is referred to as Ajax is actually just DHTML. We will start with a solely JavaScript Model with no connection to the server that implements our CRUD functionality. The basic functionality of any Model requires the ability to maintain a list of data records on the client much like a MySQL ResultSet or an ADO RecordSet – this is the simplest sort of Model we can create that contains almost no domain information. A list of records can be preserved in any data format such as XML or Plain Old JavaScript Objects (POJSO), however, there is some common functionality in a sim-

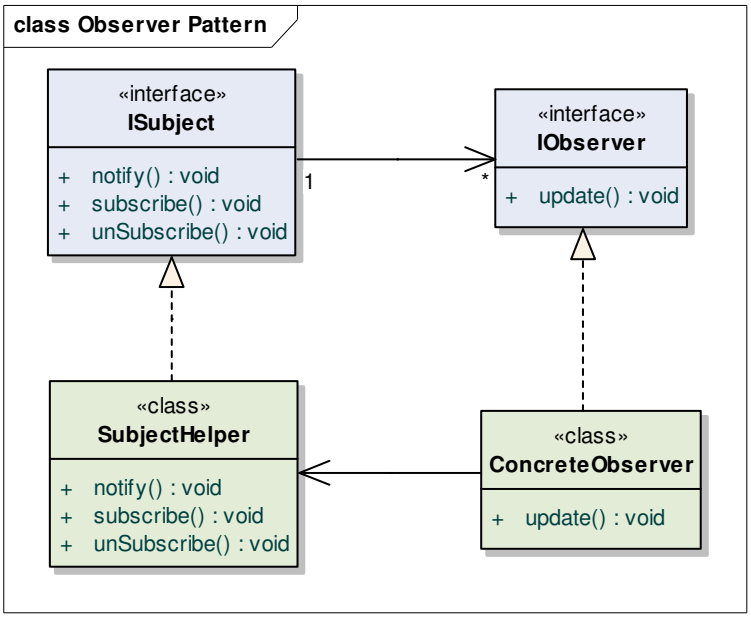
ple Model that is independent of the storage format. To fit a basic MVC Model into the Observer pattern there are some basic aspects we need to consider. Most importantly, a basic Model requires events for each of the important CRUD operations. Below is our `DataModel` class that defines a simple Model for us.

```
entAjax.DataModel = function()
{
  this.onRowsInserted = new entAjax.SubjectHelper();
  this.onRowsDeleted = new entAjax.SubjectHelper();
  this.onRowsUpdated = new entAjax.SubjectHelper();
}

entAjax.DataModel.prototype.insert = function(items, index) { }

entAjax.DataModel.prototype.read = function() { }

entAjax.DataModel.prototype.update = function(index, values) { }

entAjax.DataModel.prototype.remove = function(index) { }
```

Here we have created, according to the class definition guidelines we outlined in Chapter 2, a new JavaScript class called `DataModel` that represents our basic Model with basic events instantiated in the constructor and method stubs for the four intrinsic data operations – CRUD. In reality, the `DataModel` class should be an `abstract` class since it lacks any definition of the CRUD methods, yet since there is no simple way to indicate abstract classes in JavaScript this will have to do for now. At any rate, the `DataModel` provides a good basis from which we can build more specialized models for various types of data storage.

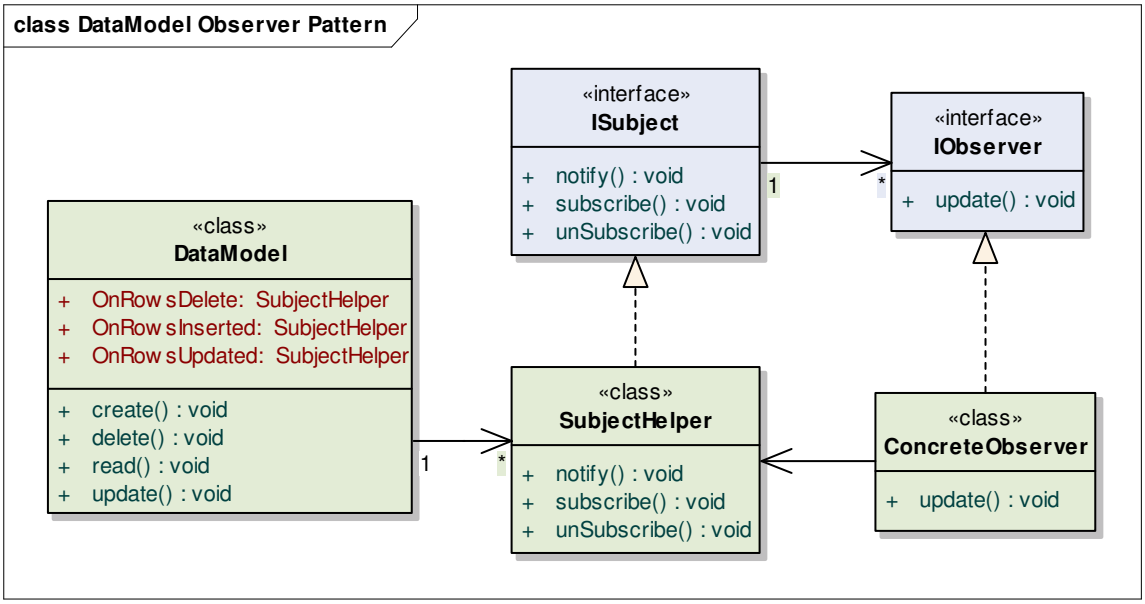
One will notice that the events (onRowsDelete etc) created as properties of the `DataModel` class and are of type `SubjectHelper`. The `SubjectHelper` class is an important part of the Observer pattern as described below.

Figure 3.3
Class Diagram of the Observer Pattern



In this incarnation of the Observer pattern rather than having a `ConcreteSubject` that implements the `ISubject` interface, as is usually the case, we have a `SubjectHelper` class that implements the `ISubject` interface. By approaching the Observer pattern in this way one can have multiple `SubjectHelper` classes – or more specific classes that inherit from `SubjectHelper` – associated with a single `Subject` as described below in the case of the `DataModel` class:

Figure 3.4
Observer Pattern Elaborated



There are several benefits to using the `SubjectHelper` class. Not only does it help to decouple the domain logic from the implementation of the Observer pattern but it also enables one to create specific helpers for more granular subjects. Our `DataModel` is a perfect illustration of

this idea. Our DataModel domain object has several operations that can occur such as data inserts, deletes, and updates. For each of these operations there may be different observers interested in a subset of these events. With the `SubjectHelper` in the Observer pattern an observer can subscribe to only the specific subjects that it wants to be notified about rather than having all observers subscribe to the DataModel object itself, which results in *all* observers being notified of every event that occurs irrespective if they are interested in the particular event or not. As one can imagine, having all observers subscribe to the domain object itself rather than the specific events that concern them can create significant overhead at run-time. The `ISubject` interface from our UML model will look something like this in JavaScript:

```
entAjax.ISubject = function()
{
  this.observers = [];
  this.guid = 0;
}

entAjax.ISubject.prototype.subscribe = function(observer)
{
  var guid = this.guid++;
  this.observers[guid] = observer;
  return guid;
}

entAjax.ISubject.prototype.unSubscribe = function(guid)
{
  delete this.observers[guid];
}

entAjax.ISubject.prototype.notify = function(eventArgs)
{
  for (var item in this.observers)
  {
    var observer = this.observers[item];
    if (observer instanceof Function)
      observer.call(this, eventArgs);
    else
      observer.update.call(this, eventArgs);
  }
}
```

`ISubject` has only three methods. All the observers for a particular subject are kept in the `Observers` object hash and observers can be added or removed through the `subscribe()` and `unSubscribe()` methods respectively. The `notify()` method is used to iterate over the collection of observers and call the `update()` method on each of them. We have also slightly augmented the familiar Observer pattern by enabling one to specify a custom method on the observer object to be called rather than requiring calling the `update()` method. This will be useful in a moment for specifying global methods as event handlers.

Although we prefix the `ISubject` class with `I` to convey that it is an interface, given the lack of support for interfaces in JavaScript it becomes more of a pseudo interface or abstract class. Given the dynamic nature of JavaScript the interface can also be used leveraged more as a means to achieve multiple (single-level) inheritance. It is best not to get too hung up on the se-

mantics since JavaScript is such a malleable language you can pretty much achieve anything depending on the amount of effort you want to devote.

```
entAjax.SubjectHelper = function()
{
  this.observers = {};
  this.guid = 0;
}
```

This will be the first chance we get to take advantage of our JavaScript inheritance model. In fact, we are going to define a simple way of providing an interface that also enables the use of default values. While we cannot ensure the child classes implement the methods of an interface, we can at least ensure that the methods to be implemented have some default realization in the interface itself. Admittedly, it is a bit of a hack but at the same time it does provide us with the means to achieve both classical inheritance and interfaces. The way that we implement interfaces could actually be thought of more as multiple, single-level inheritance.

For our inheritance to work, immediately after we define our class we have to call entAjax.extend with the sub-class and the parent class as the two parameters. In this would look something like this:

```
entAjax.implements = function(klass, interface)
{
  for (var item in interface.prototype)
  {
    klass.prototype[item] = interface.prototype[item];
  }
}

entAjax.SubjectHelper = function()
{
  …
}

entAjax.implement(entAjax.SubjectHelper, entAjax.ISubject);
```

We can now fill in the CRUD methods of our abstract `DataModel` class by creating a new `SimpleDataModel` class that inherits from the `DataModel` class that implements the details of the method stubs for managing our data in a particular way. How we store the data depends on a lot of issues, most prominent of which are what the data will be used for and what the target end-user browser is. To begin with, we will store the data using arbitrary objects in a JavaScript `Array`. This makes our CRUD operations pretty simple and means it should be interoperable with data formatted as JSON from either the server or the client. The `SimpleDataModel` class will implement the CRUD methods and ensure that the corresponding events are fired for those methods. `SimpleDataModel` looks something like this:

```
entAjax.SimpleDataModel = function()
{
  // Call the base constructor to initialize the event objects
```

```
  entAjax.SimpleDataModel.baseConstructor.call(this);
  // List of records in our Model
  this.Items = [];
}
// Inherit from DataModel
entAjax.extend(entAjax.SimpleDataModel, entAjax.DataModel);

entAjax.SimpleDataModel.prototype.insert = function(items, index)
{
  this.Items.concat(items);
  this.onRowsInserted.notify({“source”:this, “items”:items});
}

entAjax.SimpleDataModel.prototype.read = function(query)
{
  return this.Items;
}

entAjax.SimpleDataModel.prototype.update = function(index, values)
{
  var item = this.Items[index];
  for (var field in values)
  {
    item[field] = values[field];
  }
  this.onRowsUpdated.notify({“source”:this, “items”:[item]});
}

SimpleDataModel.prototype.remove = function(index)
{
  var item = this.Items.splice(index, 1);
  this.onRowsDeleted.notify({“source”:this, “items”:[item]});
}
```

The CRUD operations are fairly basic and use native JavaScript methods on the `Array` object such as `concat()` and `splice()` to keep things simple and fast. We are keeping the `SimpleDataModel` agnostic of the type of data that is stored in the array, yet will soon look at some of the improvements we can make by enforcing more strongly typed data “rows”. The important thing to notice in the `SimpleDataModel` is that in each of the `create()`, `update()`, and `remove()` methods we have called the `notify()` method of the `onRowsCreated`, `onRowsUpdated`, and `onRowsDeleted` properties respectively. In the context of MVC, notifying observers of changes to data gives us the ability to notify a View of a change in the Model data. To notify a View of a change in the data we need at least two things, a list of events on which the Model sends out notifications and a list of objects that have registered themselves to be notified. These are both managed by the `SubjectHelper` class.

To take advantage of the Observer pattern we need to define the events for which other objects may be interested in listening. In the case of the Model, some events might be things like `RowInserted`, `RowDeleted` and so on. The nature of these is completely up to the application architect and highly dependent on the application. In the case of inserting data into the Model, the `OnRowsInserted` event is fired – by firing we just mean that the `notify()` method is called. At the end of the `insert()`, `update()`, and `remove()` methods we have

added one line of code that calls the `notify()` method of the respective `SubjectHelper` such as:

```
this.onRowsInserted.notify({"source":this, "items":items});
```

The arguments that are passed to the `notify()` method are subsequently provided to the call to `update()` on the observer. This allows the observer to receive some context about the event that occurred. In the case of the `create()` method on the `SimpleDataModel` class we want to actually provide the event handler with information about the specific data that was created such that it can take some action on that data. Depending on one's development methodology, it may be prudent to create a `RowsCreatedEventArgs` class, for example, rather than using a struct as the data structure for passing information to the observer. Usually the most compelling reasons for using a JavaScript class are documentation and ease of programming. The other option is to not pass information about the event to the observer's `update()` method and instead leave it up to the observer to request information through a method on the `SubjectHelper` such as `getData()`. This can be a good approach if there is a large amount of data associated with an event that is not used by the majority of the observers.

This is a very simplified model. The most obvious omission is the fact that the data is stored in a JavaScript array with no connection to the server for persistence and therefore will only exist for the duration of the application. Although we have presented an entirely JavaScript based Model, we could just as easily have used an alternative way of storing the data on the client such as an XML document. XML can be a good option if your server environment provides or consumes XML based data or your application requirements identify data interoperability as a high priority. XML can be also leverage XSLT, which can simplify the process of building the Ajax View computationally, technically, and even in terms of development workflow. Furthermore, XML based data is easily and efficiently manipulated (think grouping and pivoting) using XSLT on the client. Still, no matter how we store the data on the client it doesn't solve the problem of our data being destroyed as soon as the user closes the web browser or visits a different web site; by re-factoring this code, we will soon integrate some Ajax functionality for loading and saving the data on the server and, in some cases, even on client. At the very least, this simple Model gives us a good framework so we can start manipulating data in the web browser JavaScript sandbox and it should be good enough to connect it up to the DOM based View through our event driven Controller. Once we do get to the point where the Model spans the network from the client to the server, we will look at how to take advantage of some more important design patterns such as ActiveRecord, which has been popularized of late with the rising interest in the Ruby on Rails platform. Making use of these well known patterns like MVC, Observer, and ActiveRecord can be a key differentiator when it comes to building a high-performance, usable Ajax application.

To bring the ideas behind the Model together let's look at a short example. We start by creating a `Listener` class that implements the `IObserver` interface – in reality all that means is that it implements an `update()` method. An instance of the `Listener` class is then subscribed to the `OnRowsCreated` event of a `SimpleDataModel`. Now when we manually call the `create()` method on the instance of the `SimpleDataModel` class, the `update()` method on the subscribed observers is executed.

```
// Create a simple class to listen for updates
Listener = function() {}
// When it receives an update just call alert
Listener.prototype.update = function(eventArgs)
{
```

```
  alert(eventArgs.type + ' - ' + eventArgs.data.getFullName());
}
// Create an instance of the listener
var CustomerListener = new Listener();

// Create a new Model with no data in it
var CustomerModel = new entAjax.SimpleDataModel();

// Register the CustomerListener object to hear about changes to data
CustomerModel.OnRowsCreated.subscribe(CustomerListener);

// Finally insert a new Customer and we should see our alert
CustomerModel.create(new Customer('John', 'Doe');
```

Now that we have some data to work with, we can take a closer look at how to build up a data driven MVC View for an Ajax application.

Ajax View

Once we have the Model, whether entirely on the client or spanning the client and server, we can render the information that exists in the Model using the View. In the traditional n-tier web architecture that cleanly separates the presentation, business logic and data access components of an application, the View is generated on the server and the HTML is streamed up to the client for presentation. User-gestures propagate from the View to the Controller with a full HTTP POST request back to the server where any client-side information is processed and a new View is created and streamed to the client. Views are usually generated using some sort of scripting language such as PHP or something more full featured like Java, which in turn will likely use some other templating technologies like XSLT, Velocity (Java), or Smarty (PHP). When we consider the View of MVC in Ajax there are essentially two options. Most commonly, changes to the View are made entirely on the client using either client side templating or DOM manipulation. Performing all the View changes on the client is the essence of Ajax and can be leveraged to create a truly rich user-interface. A bit less common, and usually reserved for situations where complicated logic or leveraging of legacy resources is taking place, is the generation of small parts of the View on the server which are retrieved behind the scenes and placed directly into the DOM with little or no logic on the client. Portals might often use this architecture when there are small pieces of data coming from many disparate data sources. In Chapter 6 we will explore some of these finer points of Ajax architectures.

As an introduction we will limit ourselves to doing some basic JavaScript gymnastics. We already mentioned that there are several options to consider when building the View. Choosing the right option for building the View means taking into account various factors such as performance on both the server and client, server load, maintainability, testability, and developer skill-set. From that list, performance is paramount; one of the main reasons for moving to an Ajax architecture to begin with is due to the inherent performance issues with the traditional post-back based web application. Being an interpreted language, JavaScript tends to be slow – and depending on the application, the speed of the JavaScript interpreter can become a major bottleneck. The most obvious solution for building the View is to use the methods provided by the DOM for manipulating DOM elements. We looked at a few of these methods in Chapter 2 such as `document.createElement()`, `document.appendChild()` plus several others. We have also already looked at an example of creating some HTML using the DOM methods back in the previous chapter and even using string concatenation to build an HTML fragment earlier in this chapter. For example, if we wanted to create a View of some information about a Customer using the DOM rather than string concatenation we could do something such as this:

```
var aCustomerList = CustomerData.read();
var iCustomers = aCustomerList.length;
for (var i=0; i<iCustomers; i++)
{
  var dCustomerDiv = document.createElement('DIV');
  var sCustomerName = aCustomerList[i].getFullName();
  var dCustomerName = document.createTextNode(sCustomerName);
  dCustomerDiv.appendChild(dCustomerName);
  document.body.appendChild(dCustomerDiv);
}
```

Note that since JavaScript is not a compiled language and developers are not always editing the code in an integrated development environment (IDE) with code autocomplete (intellisense), it can be helpful to prepend your variable names to identify variable types as we have done here. In this code listing we have used the standard DOM methods, however, one can also opt to use the often overlooked `<table>` specific DOM methods when building tabular structures. For the most part `<table>` tags are deprecated in favour of using `<div>` elements and CSS for layout. Two advantages to using `<table>` elements are that it can be very fast for rendering and most software for web page accessibility, such as JAWS from Freedom Scientific, uses the `<table>` element markup to glean information about the data in a web page – more on this in Chapter 8. Although using the standard DOM API is quite intuitive when approaching the problem from an XML mindset, there are a few noteworthy alternatives. The most common way, as we have mentioned, is to use the de facto standard HTML element `innerHTML` property. `innerHTML` is the fastest way of getting large amounts of data into a web page and tends to be one of the easiest. Using the DOM API is, in general, slower than using `innerHTML` for manipulating the DOM (we will show some benchmarking results later). Given than using `innerHTML` is the fastest approach and that it takes a string value, the real question regarding generating the View becomes one of how to create the HTML string that the `innerHTML` property is set to? As you could probably guess the quick and dirty way is to build a string by concatenating strings together to build your HTML. If we want to create a list of customer names and insert them into the DOM, we can change our previous bit of DOM manipulation code to look something more like this:

```
var sCustomerList = "";
var aCustomerList = CustomerModel.read();
var iCustomers = aCustomerList.length;
for (var i=0; i<iCustomers; i++)
{
  sCustomerList += '<div>'+aCustomerList[i].getFullName()+'</DIV>');
}
$('CustomerList').innerHTML = sCustomerList;
```

Not only does this end up being less code but, accordingly, considerably faster. Although most JavaScript frameworks have some sort of simple API for string-building, unlike many other programming languages there is no optimized string-building functionality in JavaScript so we just have to settle for the simple string concatenation.

At this point most developers will be bemoaning the fact that string concatenation is the preferred way of building HTML fragments. Luckily, to avoid both ugly string concatenation and the

dog slow DOM API, we can take advantage of some templating techniques. From the point of view of keeping clear separation in your development workflow and code, using templates is certainly the most enticing option. Templating can be done with varying degrees of complexity and performance. A basic templating scheme can be conjured using some special syntax and – as most templating techniques take advantage of – regular expressions. Continuing on with the theme of building a list of customer names, let's first define a basic template that we can use to show each customer name in our list. The syntax will designate replacement values using `{$ObjectPropert}` to indicate that the ObjectProperty of the current JavaScript execution context should be placed in the template at that position. So for our Customer list it might look something like this:

```
<div>${firstName} ${lastName}</div>
```

To apply the template to some data we can use a function that looks something like this:

```
function Render(oCustomer, sTemplate)
{
  while ((match = /\$\{(.*?)\}/.exec(sTemplate)) != null)
  {
    sTemplate = sTemplate.replace(match[0], oCustomer[match[1]]);
  }
  // Return the filled in template
  return sTemplate;
}
```

This will allow us to create basic templates that use search and replace for the various properties on a given object. Notice that we actually used a regular expression to find and replace the appropriate information in the template – like most languages JavaScript too wields the power of regular expressions. We could re-factor this to include things like calling methods on the object, calling other global methods, and conditionals or looping based on the data. Although it may seem useful to have conditionals and looping in the templates, it can make the building and maintenance of the templates more difficult. Furthermore, by maintaining the logic external to the HTML templates, our data and presentation are still well separated. We could also make multiple templates for a given part of the user-interface and use JavaScript to evaluate some conditional value. For example, we can apply a different template to our customer data depending on if the customer has a positive or negative balance like this:

```
function Render(oCustomer, sPositiveTemplate, sNegativeTemplate)
{
  var sTemplate = sPositiveBalanceTemplate;
  if (oCustomer.balance > 0) {
    sTemplate = sNegativeBalanceTemplate;
  }
  while ((match = /\$\{(.*?)\}/.exec(sTemplate)) != null)
  {
    sTemplate = sTemplate.replace(match[0], oCustomer[match[1]]);
  }
  // Return the filled in template
  return sTemplate;
};
```

Using this technique there is far less template debugging and the templates are devoid of any program logic making them more re-usable and easier for a designer to build in isolation of the rest of the application – all they need to know is that there will be cases where Customer names will be rendered in two different ways, one indicating when the Customer has an outstanding balance and one when they don't. Still, there is something to be said for using a more full featured templating system like either JSON templating (JSONT) or XSLT.

Ajax Controller

Now that we have looked at a basic Model and some fundamentals of creating the View, we need to glue those together and make an application with which an end-user can actually interact. To glue everything together in an MVC Ajax application is the Controller. Due to the nature of the Controller, responding to users-gestures and orchestrating the Views and Model, the Controller is highly dependent on the DOM Events API. We discussed much of the DOM Event model back in Chapter 2 and showed how to attach events to DOM elements in a cross-browser friendly way. There was one very important consideration that we did not take into account and that is the fact that in Internet Explorer there is a well known memory leak. The memory leak is most commonly associated with the attachment of DOM Events in Internet Explorer. In certain circumstances, a circular loop between the DOM and JavaScript can be created when attaching event handlers; this occurs when an anonymous function or closure is used as the event handler and in the execution scope that the anonymous function captures is a reference to the HTML element to which the anonymous function is being attached. The idea is outlined in the code below:

```
<html>
  <head>
    <script type="text/javascript">
var example = {};
example.init = function() {
  var customers = $("customerList");
  customers.onclick = function() {this.style.fontWeight = "bold"};
}
window.onload = example.init;
    </script>
  </head>
  <body>
    <div id="customerList">
      <div>Jim</div><div>Bob</div><div>Mike</div>
    </div>
  </body>
</html>
```

In the `example.init` function we get a reference to the HTML element with Id "`customerList`" and then set the `onclick` property of that HTML element to be an anonymous function; that anonymous function now captures the local scope of the `example.init` function, which includes the `customers` variable that points to the same HTML element that the anonymous function has now been attached to – thus we have a circular reference. It is not so much that this is a problem in and of itself, but there is a problem if this circular reference is not destroyed before the page reloads because otherwise that memory is lost due to the fact that circularly referenced JavaScript and DOM objects are immune to the Internet Explorer garbage collection algorithm. This problem is present in both IE 6 and IE 7 so we need a work around. Strictly

speaking what we really need is a common and unobtrusive approach for managing the problem. Although it may seem like a pain to deal with, and despite the fact that "it's just JavaScript" this memory leak can actually get rather out of hand in a complex application. Enterprise applications in particular are often used for long periods at a time and even small memory leaks can start to build up and hinder performance greatly. The recommended approach to managing this problem is to keep track of all the HTML elements that have event handlers attached to them and subsequently detach the event handlers when the web page is unloaded.

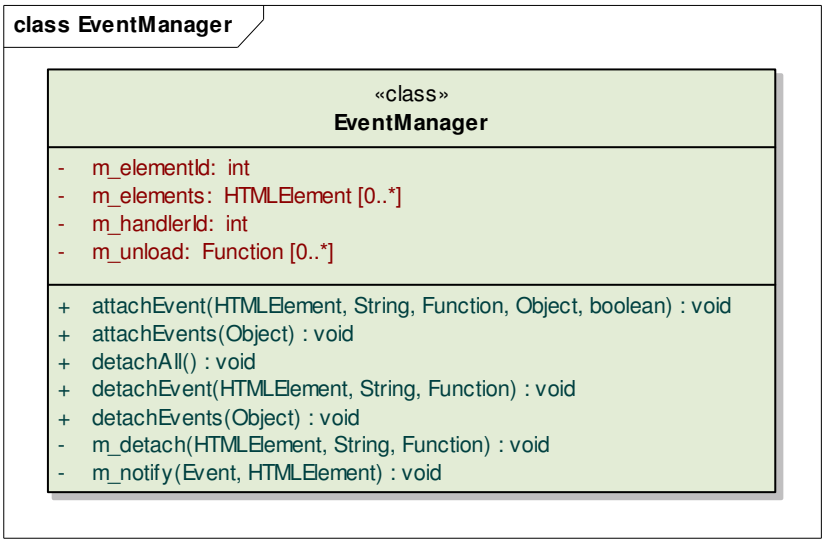
To help ease our event management pains we will create an event object following the *Singleton pattern* through which events can be attached and detached to HTML elements in a cross-browser environment. Rather than going through the trouble having a formal Singleton `getInstance` method on an event manager class we will take advantage of working with JavaScript and create a single `EventManager` object that exists for the duration of the existence of the web page. The general approach we take with event management is to keep track of all attached events in JavaScript and only attach a single event handler to a given element for any given event, rather than attaching each event handler to the HTML element explicitly; that single event handler is a method on the `EventManager` object, which has the responsibility of delegating the particular event to each of the event handlers that we are manually managing.

This approach to events has several important advantages. Although working around the problems with Internet Explorer garbage collection is a primary objective of our event management strategy, there are several other significant goals that approach to events will help us achieve.

1) Attachment of event handlers in a cross browser way.
2) Enabling event capturing.
3) Providing access to a global `Event` object.
4) Providing access to the element on which the event fired.
5) Providing access to the element on which the event was handled.
6) Preventing the Internet Explorer memory leak.

If we wrapped the event management in a "proper" Singleton class rather than using static methods and properties in the `entAjax` namespace, an `EventManager` class definition would look like this:

Figure 3.5
Traditional Versus W3C Box Model



Of most importance is the private `m_elements` array that contains references to all of the HTML elements to which event handlers have been registered. This array is where we manually manage all of the elements as well as the associated events and event handlers rather than explicitly attaching each event handler to the element. The attachment process sets a special expando property on the HTML element that contains all the information required to manually manage the events. JavaScript code for event attachment and notification is shown below:

```
// Singleton object.
entAjax.EventMangager = {};

entAjax.EventManager.attachEvent =
  function(element, type, handler, context, capture)
{
  // Increment our unique id to keep it unique
  var handlerGuid = this.handlerId++;
  var elementGuid = this.elementId++;

  // Check if the handler already has a unique identifier or not
  if (typeof handler.ea_guid != "undefined")
    handlerGuid = handler.ea_guid;
  else
    handler.ea_guid = handlerGuid;

  // Check the expando ea_guid property on the HTML Element is defined
  if (typeof element.ea_guid == "undefined")
  {
    element.ea_guid = elementGuid;
    // Add element to private elements array
    this.m_elements[elementGuid] = element;
  }

  // Expando ea_events contains registered events for the element
  if (typeof element.ea_events == "undefined")
    element.ea_events = {};

  // Check if event type is already in the ea_events expando
```

```
  if (element.ea_events[type] == null)
  {
    element.ea_events[type] = {};

    // Browser checking for IE / W3C
    if (element.addEventListener)
    {
      // W3C / MOZ event attachment
      element.addEventListener(type, function () {
          entAjax.EventManager.m_notify.call(this, arguments[0], ele-
ment)
        }, capture);
    }
    else if (element.attachEvent)
    {
      // IE event attachment
      element['ea_event_'+type] = function () {
        entAjax.EventManager.m_notify.call(this, window.event, ele-
ment);
      };
      // Detach will need to be used to avoid memory leaks!
      element.attachEvent('on'+type, element['ea_event_'+type]);

      // Support event capture as well as bubble
      if (capture) element.setCapture(true);
    }
  }
  // Add the handler to the list, track handler _and_ context
  element.ea_events[type][handlerGuid] = {
    'handler': handler,
    'context': context};
}
```

There is a lot of code there to digest. As we have mentioned, the event management is intended attach an event handler to the HTML element using the browser specific `element.attachEvent()` or `element.addEventListener()` methods only the first time a certain event type is used. In those instances when an event handler is first attached to an element for a certain event type, the specified handler is the static `entAjax.EventManager.m_notify()` method, which is responsible for actually executing all of the *real* event handlers that we are managing manually in the `m_elements` array. The `m_notify()` method looks like this:

```
/**
 * Global event handler responsible for delegating events
 * @param {Event} eventObj Browser specific event object
 * @param {HTMLElement} element The element handling the event
 */
entAjax.EventManager.m_notify = function(eventObj, element)
{
  // Set the global entAjax.Event object to the event object
  entAjax.Event = eventObj;
  // Object augmentation to track element that handled the event
  entAjax.Event.handlerElement = element;
```

```
  if (!nitobi.browser.IE)
  {
    entAjax.Event.srcElement = eventObj.target;
    entAjax.Event.fromElement = eventObj.relatedTarget;
    entAjax.Event.toElement = eventObj.relatedTarget;
  }

  for (var handlerGuid in element.ea_events[e.type])
  {
    var handler = element.ea_events[e.type][handlerGuid];
    if (typeof handler.context == "object")
      // Call handler in context of JavaScript object
      handler.call(handler.context, eventObj, element);
    else
      // Call handler in context of element on which the event was
fired
      handler.call(element, eventObj, element);
  }
}
```

When the event is actually fired by some end-user interaction `entAjax.EventManager.m_notify()` method handles the event and subsequently delegates the event to all the interested handlers. Since the `m_notify()` method orchestrates the calling of attached event handlers we are able to ensure the order in which the event handlers are called - something not guaranteed in most browsers - and to specify any arguments we like. This indirection is what allows us to circumvent the differences in event handling between various browsers and gives us a little more flexibility we didn't even ask for. So, for example, if we were to attach two event handlers to some HTML element that are to be fired on the `onclick` event, the `entAjax.EventManager.m_notify()` method would be attached to the element as the `onclick` event handler and each of the event handlers we wanted to attach would be stored in the `ea_events['onclick']` expando property on the HTML element itself. Subsequently, when the end-user clicks on the element the `m_notify()` method will call each handler that is defined in the `ea_events['onclick']` expando property on the HTML element, as one can see in the `m_notify()` method above. If we were to serialize the HTML element after two different `onclick` event handlers have been attached it would look like this:

```
<div
  onclick="entAjax.EventManager.m_notify(event, this)"
  ea_guid="7"
  ea_events="{'click':
               {'0':
                 {'handler':Function, 'context':Object},
                '1':
                 {'handler':Function, 'context':Object}
               },
              'mouseover':{…}
             }" … >
</div>
```

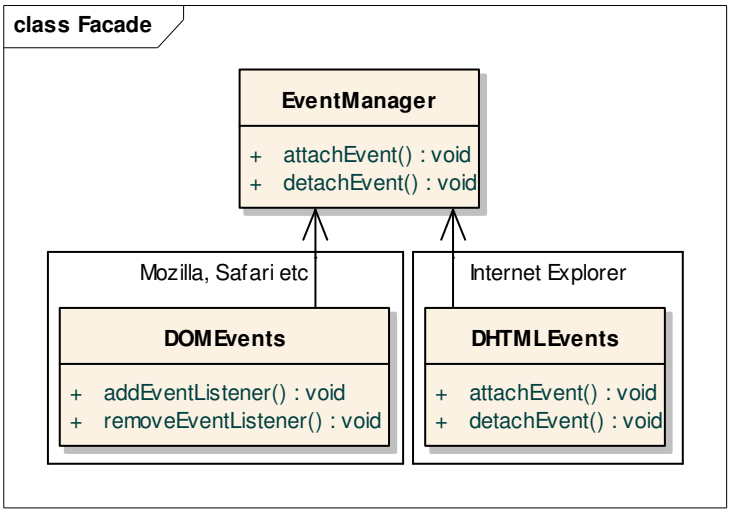
Now we can look at how this approach to attachment solves our six main problems. To en-

able us to attach event handlers to HTML elements in a cross-browser friendly way (point #1) we have encapsulated a check for the browser type in the static `attachEvent()` and `detachEvent()` methods of the `entAjax.EventManager` object - effectively using the *Façade pattern*. We just needed to use the `attachEvent()` method in Internet Explorer and `addEventListener()` in other browsers like Firefox, Safari and Opera (although Opera supports both methods).

The Façade Pattern

One of the most commonly implemented software design patterns in JavaScript is the Façade pattern. This is essentially when we create a new API or interface for the purpose of making easier or simplifying another one. We often use Façade's in JavaScript to mask the complexities of cross-browser development. For example, it's far better to simply call a single event method using a Façade than to check for what browser is running the application and calling the respective function each time.

Figure 3.6
Traditional Versus W3C Box Model



Event capturing as we discussed in Chapter 2 is available in both the Internet Explorer and W3C event models, although event capture does work differently in the different browsers. Again we have used a Façade to hide the details of event capture by providing a boolean argument in the attach method that will enable capture if true and by default does not enable capture. Event capture can be very useful in some situations as we will see later. We have also enabled #2 by using the `setCapture()` function in IE and using the support for defining capture events in `addEventListener()` in the other web browsers. Despite it being a very useful and sorely misunderstood event handling technique, event capture is largely rendered ineffective due to differences between the implementations in Internet Explorer and Firefox browsers.

Although it is not part of the W3C Event specification, Internet Explorer provides access to the event information through the global `window.Event` object. By intercepting and delegating

the events as we have done, we were also able to create our own global event object that is essentially the same as the global `Event` object property in IE. Furthermore, we also pass the `Event` object as the first parameter to the event handler method, which is how the `Event` object is to be accessed according to the W3C DOM Event specification. Handling the Event object in this manner means that it will be familiar to developers accustomed to working with either Internet Explorer or W3C compliant based browsers, while at the same time we prevent any collisions with other Ajax libraries such as Microsoft's Atlas in which the event model uses the `window.event` property in both Internet Explorer and Firefox. Although that does fix many of our problems with the `Event` object, we have not yet taken a close look at the differences between the Event object methods and properties in the Internet Explorer and W3C compliant models.

Closely related to the Event object is how various event properties are accessed. One of the important objects to have access to is the element which fired the event (point #4). This is easily handled through the `Event` object `srcElement` or `target` properties in Internet Explorer and most other browsers respectively. These discrepancies on object properties can generally be handled in one of two ways. While we have used the *Façade pattern* for event management, we could have also extended the native objects in both Gecko and KHTML / WebKit based browsers such that they would support the Internet Explorer `attachEvent()` method directly on objects of type `HTMLElement`. Both Firefox and recent builds of Safari (again build 15836) support the modification of the `prototypes` of native browser objects such as the `Event` object or the `HTMLElement` object. To extend the `Event` object in browsers supporting getters and setters one could use the special `__defineGetter__` method on the class's `prototype` property such as:

```
Event.prototype.__defineGetter__("srcElement", function () {
   var node = this.target;
   while (node.nodeType != 1) node = node.parentNode;
   return node;
}
```

There are other options available to a developer when creating a cross browser `Event` object include augmenting the native object with additionally properties, or copying all the properties from the native `Event` object to an entirely custom object.

Although it is a simple task to get a reference to the DOM element that triggered an event, accessing the element that handled the event – i.e. the DOM element to which the event handler was attached to - is not possible in Internet Explorer when using the native `attachEvent()` method. To have access to the DOM element that handled the event (point #5), which can be a convenient thing to know in many situations, we pass a reference to the element that handled the event as the second parameter to the event handler method. Alternatively, we have also augmented the `Event` object with the `handlerElement` property which is a reference to this element as well. Having no reference to the element that handled the event is an oft stated complaint about the Internet Explorer event model. In Firefox, on the other hand, one can usually access the element that handled the event through the `this` keyword in the handler function – ie the handler function is executed in the context of the HTML element. However, this is not always desirable, particularly when working in an object oriented environment where often times the handler is not a global function but instead a method to be executed in the context of the object to which the method belongs. Luckily our event management approach encapsulates the details of that too.

The only thing left is to ensure that our event attachment will not leak any memory (point #6) - this will require a little more code. If we were to make a simple web page with an event handler connected to an HTML element using our `entAjax.EventManager.attachEvent()` method we would still leak memory like a sieve in Internet Explorer due to the circular loops we have created (through closures and expando properties) between the HTML DOM and JavaScript runtimes. To prevent this infamous memory leak, we need to be sure that we detach the event handlers from the HTML elements right before the web page is unloaded. Of course, a conscientious developer could always write code to detach or unregister their events when their application unloads, however, this is rarely the case. By attaching events through our custom static methods the elements that have event handlers attached to them are managed internally such that when the web page `unload` event is fired all the event handlers can be detached automatically.

The actual detaching of an event is quite simple and performed in the private `m_detach()` method. Of course, we use the term private quite loosely in this case; we have gone ahead and commented it as private and named it using the "`m_`" prefix that is commonly used to indicate private members. When an event handler is detached we need to do some house keeping to ensure that when (and only when) the last event handler for a given event type is removed from an element the `entAjax.EventManager.m_notify()` handler is also detached from the HTML element using the native `detachEvent()` or `removeEventListener()` method for the corresponding web browsers. Upon further thought, one will likely notice that we can run into problems with the `onunload` event because developers may also want to register their own `onunload` events in addition to the important `onunload` event handler that cleans up all of the attached events. Any `onunload` events that are registered through our event interface would be in line to get detached and garbage collected like any other event before ever being called – essentially throwing out the baby with the bath water. To get around this we manage `onunload` events separately from the other events and keep track of them in the `entAjax.EventManager.m_unload` array. The `window.onunload` event is set to trigger the detaching of all the managed events like this:

```
window.onunload = entAjax.EventManager.detachAll();
```

Of course setting the `onunload` event in this manner can be destructive but we can get around that as we will see in a moment. To prevent interference with the `onunload` event, it is also a good idea to stick to using the `onbeforeunload` event for any custom events, which is another de-facto standard introduced by Microsoft.

Aspect Oriented

We will take this opportunity to interject an example of how to use Aspect Oriented Programming (AOP) to set the `window.onunload` event without destroying any other event handlers to which it already refers. Page or application wide events like `onload` and `onunload` can be touchy subjects when it comes to JavaScript frameworks and components. Various components, frameworks, and developers take different approaches to dealing with global events like onload. If you expect your code to be running on a web page with code from different development teams or component vendors, the rule thumb is to preserve functionality wherever possible. The flip side of this is that an end-developer using your code might also be less than benevolent and do things destructively that could potentially cause big headaches for you. The `onload` event is one of these major flashpoints. As we saw earlier in this chapter, using `onload` can be paramount for bootstrapping your Ajax application. Of course if your component relies on `window.onload`, you should be aware that not only will other components or frameworks want to

use that same event, but the end-developer will likely use either `window.onload` or even the explicit `onload` attribute on the `<body>` element to run their own JavaScript code. First of all, if we are using the `window.onload` event then we will want to set it in a non-destructive manner. Problems surrounding the `onload` event in a mash-up environment is a good reason to look at some of the more advanced bootstrapping techniques we have mentioned.

AOP can help us to add event handlers to the `window.onload` event while preserving any previous function to which the property previously pointed. The idea behind AOP is that we can add functionality to an object dynamically at runtime in a manner similar to the *Decorator pattern*, however, with much less up-front design; given the nature of the JavaScript language this is exceedingly easy. We will create some static methods in the `nitobi.lang` namespace that will take two methods as arguments and have the second method called before, after, or around the first method every time the first method is called – this is often referred to adding "advice" to a "join point" in AOP. Since JavaScript is dynamic, explicit join points for advice do not need to be defined during application design and instead can be added in a more ad hoc fashion. For example, in the case of a function that is assigned to the `window.onload` property, we would want to take that function and ensure that it gets called as well as any other functions that we want to attach to that event. To do this we take advantage of JavaScript closures and associative arrays once more:

```
/**
 * Attaches a function to be called after another function
 * @param {Object} srcObj The context of the source function
 * @param {String} srcFunc The source function
 * @param {Object} aspObj The context of the advice function
 * @param {Function} aspFunc The function we want to attach
 */
nitobi.lang.addAfter = function(srcObj, srcFunc, aspObj, aspFunc)
{
  var oldFunc = srcObj[srcFunc];
  srcObj[srcFunc] = function()
  {
    oldMethod.apply(srcObj, arguments);
    aspObj[aspFunc].apply(aspObj, arguments);
  }
}

nitobi.lang.addAfter(window, "onunload", myObj, myObj.myFunc);
```

AOP is just one of the unique capabilities of JavaScript due to the fact that it is a dynamic language. It can be a very useful tool for dynamically changing class or instance functionality at runtime as well as a host of other things like making the Decorator pattern easier or even moot.

Aspect Oriented Programming

Aspect oriented programming refers to a programming approach that is used to address situations where there are so-called cross cutting concerns. In particular this arises in object-oriented programming where encapsulation groups like things into various levels of packages and classes

– which is all well and good when we want to create a class hierarchy of animals say. However, cross cutting concerns are those aspects of a program that span horizontally across vertically grouped classes. The canonical example of this is logging. We may want some sort of logging functionality added to two sets of classes in different inheritance hierarchies. Logging functionality can be added across this class hierarchy using aspect oriented programming.

Aspect orient programming generally requires a significant amount of "helper" code; however, JavaScript makes it relatively easy to achieve.

Summary

Throughout this chapter we have looked at some of the ground work one needs to get their Ajax application off the ground as well as the three main aspects of an Ajax application from the point of view of the Model-View-Controller pattern. Now that we have laid the groundwork we will be able to build upon a solid MVC foundation in the next chapter when we tackle building an Ajax user-interface component based on everything we have learned here.

Resources

Window.onload problems and solutions - Dean Edwards Blog : http://dean.edwards.name/weblog/2005/09/busted/

Cross browser JavaScript resources
http://webfx.eae.net/

About Firefox and Quirks Mode Behaviour
http://developer.mozilla.org/en/docs/Mozilla_Quirks_Mode_Behavior

Model View Controller
http://en.wikipedia.org/wiki/Model-view-controller

Internet Explorer Memory Leak Patterns
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/IETechCol/dnwebgen/ie_leak_patterns.asp