vào dây

E B V N

Khoi Ngu?n Tri Th?c.

Th?ng C?o VÁ EBVN | Ch?nh S?ch Share | Ch?nh S?ch Mua B?n | I I s T E b O o K Update 30-1-06

?
?
***** T ? C h ? c E B V N :**

EBVN l? t? ch?c d?u ti?n v? duy nh?t c?a Vi?t Nam c? th? t? khai th?c c?c ngu?n ebook cho ri?ng m?nh.
Nh?ng ebook ch?ng t?i d? dang v? s? khai th?c lu?n lu?n newest, hotest v? !!!!!!.
Ð? ch?nh l? l?i kh?ng d?nh v? cung l? kim ch? nam d? EBVN t?n t?i ho?t d?ng.
?
V?o th?i di?m n?y, do l?c lu?ng c?n qu? m?ng n?n EBVN ch? t?p trung khai th?c hai m?ng c?ng ngh? h?ng d?u v? c? tuong lai h?a h?n nh?t hi?n nay. Ð? l? Dot net d?i di?n cho b?n c?ng c? l?p tr?nh. V? Artificial Intelligence d?i di?n cho b?n thu?t to?n.
**** Dot Net : bao g?m Dot net Framework, C#.net, ASP.net, VB.net, VC++.net.**
**** Artificial Intelligence : bao g?m AI, Genetic Programming, Neural Networks, Fuzzy Logic . . .**
V?i d?i ngu th?nh vi?n c?n r?t tr?, EBVN dang c? tham v?ng m? r?ng kh? nang khai th?c ebook v? chi?u s? u (ch? ? l? kh?ng c? chi?u r?ng), m? r?ng t?m khai th?c v?i hai c?ng ngh? tr?n.

Tuy nhi?n, EBVN v?n c?n dang trong qu? tr?nh h?nh th?nh l?c lu?ng "Ebook-Finder". V? v?y, n?u b?n n?o c? c?ng ch? hu?ng v?i ch?ng t?i xin h?y g?i ngay email t?i d?i tru?ng n2tuan (n2tuan@gmai.com) d? c? th? "join" v?o EBVN. Trong email, b?n c?n dua ra list m?t s? ebook b?n d? bought du?c.
EBVN s? r?t vui m?ng d?n nh?n nh?ng member m?i nhi?t t?nh, nang d?ng. Nh?ng mem m?i s? du?c c?p ngay m?t Account lo?i "Safari Max" b?n SAFARI.com trong th?i gian m?t nam, v? t?t nhi?n l? s? du?c truy c?p v?o ngu?n c?a EBVN.

E B V N

Khoi Ngu?n Tri Th?c.

Th?ng c?o vÁ EBVN | *Ch?nh s?ch share* | Ch?nh s?ch mua b?n | l I s T E b O o K Update 30-1-06

?
?
" Con ngu?i s?ng kh?ng th? thi?u du?c b?n b? ".
EBVN kh?ng th? t?n t?i n?u ebook c?a m?nh khai th?c du?c l?i ch? d?nh cho c?c mem c?a t? ch?c. Đ?ng th?i EBVN cung mu?n g?p m?t ph?n n?o d? v?o s? ph?t tri?n c?a n?n c?ng ngh? th?ng tin nu?c nh?. V? v?y, ch?ng t?i s? c? g?ng share cho c?c b?n nhi?u nh?t c? th?.
?
*********************************** O K * Đ? Y L ? Q U Y Đ? N H *******************************
Ch?ng t?i s? c?p nh?t c?c ebook m?i nh?t ngay khi c? th? tr?n website c?a t? ch?c. N?u c?c b?n c?n ebook n?o th? h?y g?i ngay email d?n cho ch?ng t?i ? d?a ch? : ebvn.ebvn@gmail.com v? ch? d?i.
N?u s? lu?ng ngu?i y?u c?u vu?t qua con s? " 100 " th? CH?C M?NG B?N V? NH?NG NGU?I REQUEST THIS EBOOK v? c?c b?n d? l? ch? nh?n c?a ebook m?nh c?n. Link download s? du?c ch?ng t?i g?i qua du?ng email ho?c ngay trong topic m? ch?ng t?i post ? ddth.com.
CH? ?: n?u c? new ebook, th?nh vi?n c?a EBVN s? post l?n forum ddth.com v? c?p nh?t ngay tr?n website.
****

E B V N

Khoi Ngu?n Tri Th?c.

Th?ng c?o vÁ EBVN | Ch?nh s?ch share | *Ch?nh s?ch mua b?n* | l I s T E b O o K Update 30-1-06

?
?
Ð? t?o ngu?n kinh ph? m?t ph?n b?i du?ng cho c?c member xu?t s?c c?a t? ch?c v? ph?c v? cho nh?ng d? d?nh s?p t?i c?a EBVN, EBVN m? th?m d?ch v? b?n ebook. D?ch v? n?y nh?m d?p ?ng nhu c?u c?a nh?ng chuy?n gia tin h?c dang th?c s? c?n ebook cho c?ng vi?c c?a m?nh. Gi? m?i cu?n ebook l? 10.000 VND.

C?c b?n h?y g?i email cho ch?ng t?i ? d?a ch? : ebvn.ebvn@gmail.com. Trong thu ghi r? cu?n s?ch b?n c?n. Ch?ng t?i s? g?i email l?i cho b?n v?i th?ng tin :
*** S? di?n tho?i d? b?n c? th? li?n h? v?i ch?ng t?i. B?n c? th? g?i di?n tru?c khi g?i ti?n**
*** Th?i h?n b?n s? n?p ti?n v?o t?i kho?n c?a EBVN.**

Sau d? trong th?i h?n d? n?u, b?n h?y g?i ti?n v?o t?i kho?n cho ch?ng t?i theo th?ng tin du?i d?y.

S? t?i kho?n ATM c?a c?ng ty:
Ng?n h?ng N?ng Nghi?p v? Ph?t Tri?n N?ng Th?n Vi?t Nam **AGRIBANK**
Ch? t?i kho?n: **Ng? Ng?c Tu?n.**
S? t?i kho?n: **2727-2715-0004-6365.**

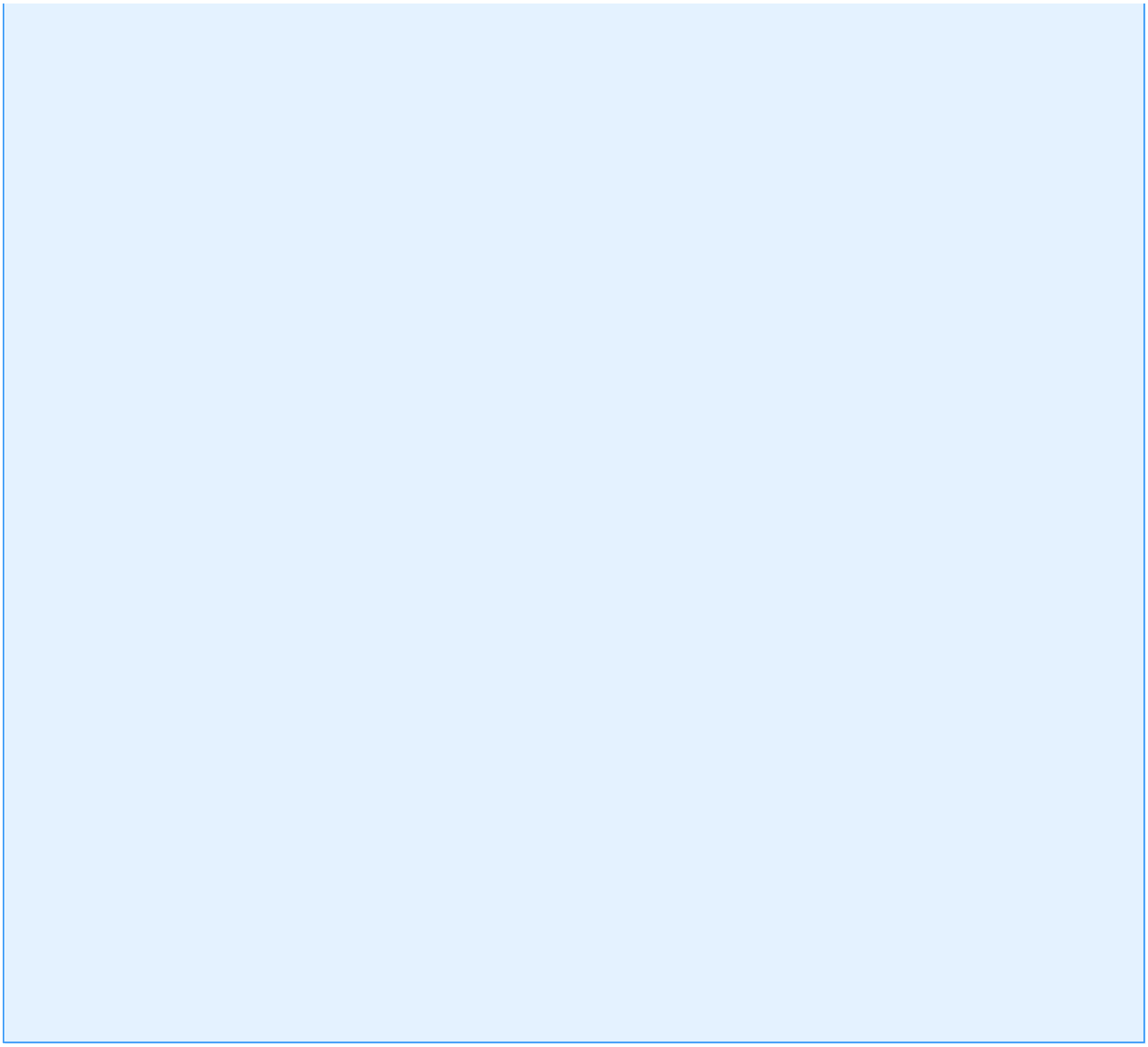
Ngo?i ra, EBVN hi?n dang n?m m?t s? account lo?i Safari Max c?a h?ng Safari. Safari l? h?ng kinh doanh s?ch online l?n tr?n th? gi?i. H?ng n?y c? m?i h?p t?c v?i r?t nhi?u publisher l?n nhu O'Reilly, Syngress, Addison Wesley, Microsoft Press, Sams Publishing. Ch?ng t?i s½ b?n cho c?c b¡n vÛi gi? t?nh theo sÑth? ng c?n l?i c?a account. 1 th?ng tuong duong v?i 5.000 VND.
Ð? bi?t th?m th?ng tin v? t? s?ch online Safari n?y, b?n h?y v?o website c?a h?ng:
www.oreilly.safari.com
?

C?m On C?c B?n Ð? ?ng H? E B V N.

E B V N

Khoi Ngu?n Tri Th?c.

List s?ch du?c c?p nh?t b?i EBVN.?
N?u b?n dang c?n ebook n?o du?i d?y xin g?i email d?n:
EbVn.eBvN@gmail.com
b?n s? c? co h?i s? h?u cu?n ebook m?nh c?n.



Ajax Hacks
By Bruce Perry
.....
Publisher: O'Reilly
Pub Date: March 01, 2006
ISBN: 0-596-10169-4
Slots: 0

EBVN

K h o i N g u ? n T r i T h ? c

Overview

Ajax, the popular term for Asynchronous JavaScript and XML, is one of the most important combinations of technologies for web developers to know these days. With its rich grouping of technologies, Ajax developers can create interactive web applications with XML-based web services, using JavaScript in the browser to process the web server response.

Taking complete advantage of Ajax, however, requires something more than your typical "how-to" book. What it calls for is Ajax Hacks from O'Reilly. This valuable guide provides direct, hands-on solutions that take the mystery out of Ajax's many capabilities. Each hack represents a clever way to accomplish a specific task, saving you countless hours of searching for the right answer.

A smart collection of 100 insider tips and tricks, Ajax Hacks covers all of the technology's finer points. Want to build next-generation web applications today? This book can show you how. Among the multitude of topics addressed, it shows you techniques for:

Using Ajax with Google Maps and Yahoo Maps

Displaying Weather.com data

Scraping stock quotes

Fetching postal codes

Building web forms with auto-complete functionality

Ajax Hacks also features a number of advanced hacks for accelerated web developers. Discover how to

create huge, maintainable bookmarklets, how to use client-side storage for Ajax applications, and how to call a built-in Java object from JavaScript using Ajax. The book even addresses best practices for testing Ajax applications and improving maintenance, performance, and reliability for JavaScript code.

The latest in O''Reilly's celebrated Hacks series, Ajax Hacks smartly complements other O'Reilly titles such as Head Rush Ajax and JavaScript: The Definitive Guide.

Ajax Hacks
By Bruce Perry
.....
Publisher: O'Reilly
Pub Date: March 01, 2006
ISBN: 0-596-10169-4
Slots: 0

EBVN

K h o i N g u ? n T r i T h ? c

Table of Contents

Chapter 1. Ajax Basics

Ajax Basics

Previous

E B V N
We are Vietnames

Introduction: Farewell to Page Refreshing

Remember when users called the Internet the "world wide wait"? Way back in the Neolithic era of the web? With some applications, the web has not really changed that much. Fill out form, click button, web page goes away, wait, page refreshes, correct mistake, click, wait, wait...You've been stuck in this netherworld before.

A number of recent web sites, however, such as many of the cool mapping applications that have evolved of late, require much greater responsiveness in the way they interact with users. These applications require small pieces of the web page to change instantaneously, often based on server information, rather than have the entire page "go away" with every click, with the new page only reappearing in your browser view when the server's response is finally complete.

For example, if you have ever used Google Maps, the way you can drag outer-lying regions into your view conveys the impression that you have all of the maps stored locally on your computer, for your effortless manipulation. Imagine how unpopular this application would be if every time you tried to "drag" the map the page would disappear for a few (long) moments while the browser waited for another server response. The application would be so sluggish that no one would use it.

It's Not a Floor Wax

A blend of well-known technologies and a nifty JavaScript tool forms the basis of a snappier and more powerful application model for the web. Before you run from a new acronym this one's easy, Ajax. Yet, it is neither a floor wax nor a desert topping. It stands for Asynchronous JavaScript and XML.

Ajax is a blend of a number of standard technologies that people are familiar with:

- JavaScript, a programming language that adds dynamic scripting to a web page. The code can be embedded right in there with the page to allow the page to implement cool new behaviors with a technique called "client-side scripting." This technique is almost as old as the web itself.
- `XMLHttpRequest`, an object or sub-set of JavaScript code that can connect with a server using the HTTP protocol. A lot of the Ajax magic is propelled by this piece of code, which all of the major browsers such as Mozilla Firefox, Internet Explorer 5, Safari, and Opera 7.6 support. The asynchronous part of Ajax derives from this object's characteristics.[1]
- Extensible Markup Language (XML), a standard method of describing data with a meta language, "information about information." The `XMLHttpRequest` object can handle the server response in standard XML format as well as plain text.
- HTML and Cascading Style Sheets, which control what the user sees on a web page. Web developers can use JavaScript to make dynamic changes to the visual interface by programming HTML elements and CSS styles.
- The Document Object Model (DOM), a model that represents a web page as a set of related

objects that can be dynamically manipulated, even after the user has downloaded the page. The web-page view is structured as a "tree" hierarchy made up of a root node and its various "branches." Each HTML element is represented by a node or branch, which are accessible by JavaScript. We show a lot of DOM programming in these hacks, a lot!

- Extensible Style sheet Language and Transformation (XSLT), a templating technology for controlling the display of information that originates in XML format.

Ajax is far from new, as these are relatively old technologies. Microsoft issued the first implementation of a JavaScript object that could query servers, as in the`XMLHttpRequest` object (although Microsoft's object had a different name), with version 5.0 of their Internet Explorer browse (as of this writing, IE is on version 6).

What is new are the plethora of web applications that use Ajax and represent a new model of interacting with Internet users. Examples of these applications are Google Maps, Google Mail, a collaboration suite called Zimbra, an interesting personal search-engine tool called Rollyo (http://www.rollyo.com/) as well as one of the first interactive web maps, this one of Switzerland (see http://map.search.ch/index.en.html). This number of Ajax applications is growing very rapidly. Wikipedia has published a short list: http://en.wikipedia.org/wiki/List_of_websites_using_Ajax.

Handle With Care

Of course, Ajax is not for everyone (particularly those desert topping fans!). Since Ajax technology can dynamically alter a web page that has already been downloaded, certain tools near and dear to may users, such as creating bookmarks for browser views and the "back" browser button, are interfered with. For example, in the absence of fancy scripting solutions, the dynamic changes you make with DOM in an existing web page cannot be linked to with a URL that you can send to your friends or save for later. We have included`Fix the Browser Back Button` and `Handle Bookmarks and Back Buttons in AJAX Applications with RSH` help shed light on these issues, if only to provide some hackable solutions.

A number of the cool Ajax tips described in this book alter the way web widgets behave, like select lists, textareas, text fields, and radio buttons that submit their own data and talk to servers behind the scenes. Browser users know these widgets by heart. Ajax-powered widgets should be first and foremost usable, and always avoid confusing and irritating your web users.

XMLHttpRequest

At the center of many of these hacks is the`XMLHttpRequest` object, which allows JavaScript to fetch bits of server data while the user is happily playing with the rest of your application. This object has its own API, which we will summarize in this introduction.

Hack 1 involves setting up the request object in JavaScript. Once the object is initialized, it has several methods and properties that your own hacks can use. Mozilla Firefox's request object has properties and methods not shared by the other major browsers[2]. Table 1-1 and 1-2 show the properties and methods supported by the request objects defined by most of the major browsers, however, such as Firefox, Internet Explorer 5.0 and later, Safari 1.3 and 2.0, Netscape 7, and Opera's latest releases (such as Opera 7.6).

Table 1-1. XMLHttpRequest properties

Property Name	Type/Description	
`onreadystatechange`	Callback function; set this to a function that will be called whenever `readyState` changes.	
`readyState`	Number; 0 means uninitialized, `open()` has not yet been called; 1 means loading, `send()` has not been called; 2 means loaded, `send()` has been called and headers/status are available; 3 interactive, `responseText` holds partial data; 4 means completed.	
`responseText`	`string`; the plain text of the response.	
`responseXML`	DOM Document object; an XML return value.	
`status`	Response status code, such as 200 (Okay) or 404 (Not Found).	
`statusText`	`string`; the text associated with the HTTP response status.	

Table 1-2. XMLHttpRequest methods

Method name	Return value/Description	
`abort()`	`void`; cancels the HTTP request.	
`getAllResponseHeaders()`	`string`; returns all of the response headers in a pre-formatted string. See Hack #?.	
`getresponseHeader(string header)`	`string`; returns the value of the specified header.	
`open(string url,string asynch)`	`void`; prepares the HTTP request and specifies whether it is asynchronous or not.	
`send(string)`	`void`; sends the HTTP request.	
`setHeader(string header,string value)`	`void`; sets a request header, but you must call `open()` first!	

Detect Browser Compatibility With The Request Object

Use JavaScript to set up Microsoft's and the Mozilla-based browser's different request objects.

You have to make sure the "engine" behind Ajax's server handshake is properly constructed, but you can never predict which browser your users show up with.

The programming tool that allows Ajax applications to make HTTP requests to a server is an object that you can use from within JavaScript code. In the world of Firefox, Netscape, Safari, and Opera, this object is named XMLHttpRequest. Recent vintages of Internet Explorer (IE), which incidentally introduced this object with IE 5.0, implement the object as an ActiveX object named Microsoft.XMLHTTP or Msxml2.XMLHTTP, depending upon which code library the user has downloaded as part of the IE package.

NOTE

My version of IE 6 will initialize the ActiveX object using either of these constructors.

We are going to refer to the ActiveX or XMLHttpRequest objects simply as the "request object" throughout this book, however, because they have very similar functionality.

As a first step to using Ajax, you must check if the browser supports either one of the Mozilla-based or ActiveX related request objects, and then properly initialize the object.

Use a Function for Checking Compatibility
Wrap the compatibility check inside a JavaScript function, then call this function before you make any HTTP requests using the object. For example, in Mozilla-based browsers such as Netscape 7.1, Firefox 1.0.2, or Safari 2.0, the request object is available as a property of the top-level window object. The reference to this object in JavaScript code is window.XMLHttpRequest. The compatibility check for these browser types looks like:

```
if(window.XMLHttpRequest){
request = new XMLHttpRequest();
request.onreadystatechange=handleResponse;
request.open("GET",theURL,true);
request.send(null);
}
```

The JavaScript variable request is to a top-level variable that will refer to the request object. If the browser supports XMLHttpRequest, then:

if(window.XMLHttpRequest) returns true because the XMLHttpRequest is not null or undefined;

The object will be instantiated with the new keyword;

Its onreadystatechange event listener (see the Introduction) will be defined as a function named handleResponse(); and

The code calls the request object's open() and send() methods.

NOTE

A common practice among programming types is to call functions that are associated with particular JavaScript objects as "methods." the XMLHttpRequest object's methods include open(), send(), and abort().

What About Internet Explorer users?
In this case, the window.XMLHttpRequest object will not exist in the browser object model. Therefore another branch of the if test in your code is necessary.

```
else if (window.ActiveXObject){
request=new ActiveXObject("Microsoft.XMLHTTP");
if (! request){
request=new  ActiveXObject("Msxml2.XMLHTTP");
}
if(request){
request.onreadystatechange=handleResponse;
request.open(reqType,url,true);
request.send(null);
}
}
```

This code fragment tests for the existence of the window top-level object ActiveXObject, thus signaling the use of Internet Explorer. The code then initializes the request using two of a number of possible parameters (e.g., Microsoft.XMLHTTP and Msxml2.XMLHTTP).

You can even get more finely grained when testing for different versions of the Microsoft request object, as in Msxml2.XMLHTTP.3.0. In the vast majority of cases, however, you will not be designing your application based on various versions of the IE request object, so the prior code will suffice.

Then the code makes one final check for whether the request object has been properly constructed (as in if(request){...}).

Given three chances, if the request variable is still null or undefined, then your browser is really out of luck when it comes to using the request object for Ajax!

Here's an example of an entire compatibility check.

```
/* Initialize a Request object that is already constructed */
function initReq(reqType,url,bool){
/* Specify the function that will handle the HTTP response */
request.onreadystatechange=handleResponse;
request.open(reqType,url,bool);
request.send(null);
}
```

```
/* Wrapper function for constructing a Request object.
Parameters:
reqType: The HTTP request type such as GET or POST.
url: The URL of the server program.
asynch: Whether to send the request asynchronously or not. */

function httpRequest(reqType,url,asynch){
//Mozilla-based browsers
if(window.XMLHttpRequest){
request = new XMLHttpRequest();
initReq(reqType,url,asynch);
} else if (window.ActiveXObject){
request=new  ActiveXObject("Msxml2.XMLHTTP");
if (! request){
request=new ActiveXObject("Microsoft.XMLHTTP");
}
if(request){
initReq(reqType,url,asynch);
/* Unlikely to branch here, as IE users will be able to use either one of the
constructors*/
} else {
alert("Your browser does not permit the use "+
"of all of this application's features!");}
} else {
alert("Your browser does not permit the use "+
"of all of this application's features!");}
}
```

You can include a final else test (as in the example) if all three checks fail, alerting the user that they will not be able to use the web application's features and perhaps recommending an upgrade.

```
} else {
alert("Your browser does not permit the use "+
"of all of this application's features!");
}
```

Use The Request Object To POST The Server Some Data

Step beyond the traditional mechanism of posting your user's form values.

This hack uses the POST HTTP request method to send data, communicating with the server without disrupting the user's interaction with the application (no page refreshing here!). Then the hack displays the server response to the user.

The web page is a simple one. It requests the user to enter their first name, last name, gender, country of origin, then click a button. Figure 1-1 shows what the web page looks like in a browser window.

Please Mister POST man

Here's the code for the HTML page:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
 "http://www.w3.org/TR/1999/REC-html401-19991224/strict.dtd">
<html>
<head>
 <script type=
 "text/javascript" src="/parkerriver/js/hack2.js"></script>
 <meta http-equiv="content-type" content="text/html; charset=utf-8">
 <title>Send a data tidbit</title>
```

```
</head>
<body>
<h3>A Few Facts About Yourself...</h3>
<form action="javascript:void%200" onsubmit=
"setQueryString();sendData();return false">
<p>First name: <input type="text" name="firstname" size="20"></p>
<p>Last name: <input type="text" name="lastname" size="20"> </p>
<p>Gender: <input type="text" name="gender" size="2"> </p>
<p>Country of origin: <input type="text" name="country" size="20"> </p>
<p><button type="submit">Send Data</button></p>
</form>
</body>
</html>
```

The first code element of interest is the `script` tag, which imports the JavaScript code (in a file named hack2.js) . The `form` tag's `onsubmit` attribute specifies two functions (`setQueryString()` and `sendData()`) which in turn format the data for a POST request and send it to the server. The hack2.js file defines the two functions. Here is the `setQueryString()` function.

```
function setQueryString(){
 //initialize the top-level variable; also reset the variable to cover when
 //the user clicks multiple times
 queryString="";
 var frm = document.forms[0];
 var numberElements = frm.elements.length;
 for(var i = 0; i < numberElements; i++) {
 if(i < numberElements-1) {
 queryString += frm.elements[i].name+"="+frm.elements[i].value+"&";
 } else {
 queryString += frm.elements[i].name+"="+frm.elements[i].value;
 }
 }
}
```

This function formats a POST-style `string` out of all of the form's `input` elements. All of the name/value pairs are separated by an "&" character except for the pair representing the last `input` element in the form. The entire string might look like `firstname=Bruce&lastname=Perry&gender=M&country=USA`.

Even if a Web-page designer adds another `input` element, this method will still be able to include the new element's values in the posted `string`. This is because the code iterates through the entire form `elements Array`, which is a property of the first form defined in the Web page.

NOTE

The variable `frm` holds a reference to this form, as in `var frm = document.forms[0];`

Now we have a `string` that we can use in a `POST` HTTP request. Let's look at the JavaScript code

that sends the request. Everything starts with the `sendData()` function. The code calls this function after calling `setQueryString()` in the HTML form tag's `onsubmit` attribute.

```
var request;
var queryString; //will hold the POSTed data
function sendData(){
 var   url="http://192.168.0.3:8080/parkerriver/s/sender";
 httpRequest("POST",url,true);
}

/* Initialize a Request object that is already constructed
 reqType: The HTTP request type such as "GET" or "POST."
 url: The URL of the server program.
 isAsynch: Whether to send the request asynchronously or not. */
 function initReq(reqType,url,isAsynch){
 /* Specify the function that will handle the HTTP response */
 request.onreadystatechange=handleResponse;
 request.open(reqType,url,isAsynch);
 /* set the Content-Type header for a POST request */
 request.setRequestHeader("Content-Type",
 "application/x-www-form-urlencoded; charset=UTF-8");
 request.send(queryString);
}

/* Wrapper function for constructing a Request object.
 Parameters:
 reqType: The HTTP request type such as GET or POST.
 url: The URL of the server program.
 asynch: Whether to send the request asynchronously or not. */

function httpRequest(reqType,url,asynch){
 //Mozilla-based browsers
 if(window.XMLHttpRequest){
 request = new XMLHttpRequest();
 initReq(reqType,url,asynch);
 } else if (window.ActiveXObject){
 request=new  ActiveXObject("Msxml2.XMLHTTP");
 if (! request){
 request=new ActiveXObject("Microsoft.XMLHTTP");
 }
 if(request){
 initReq(reqType,url,asynch);
 /* Unlikely to branch here, as IE uses will be able to use either
 one of the constructors*/
 } else {
 alert("Your browser does not permit the use of all "+
 "of this application's features!");}
 } else {
 alert("Your browser does not permit the use of all "+
 "of this application's features!");}
}
```

The purpose of the `httpRequest()` function is to check which request object the user's browser is associated with (see Hack# 1). Then the code calls `initReq()`, whose parameters are described in the comment just above the function definition.

The code `request.onreadystatechange=handleResponse;` specifies the event-handler function that will deal with the response. We describe this function a little later on. Then the code calls the request object's `open()` method, which prepares the object to send the request.

Setting Headers

The code can set any request headers after calling `open()`. In our case, we have to create a `Content-Type` header for a POST request.

> NOTE
>
> Mozilla Firefox 1.02 required the additional Content-Type header; Safari 1.3 did not. It is a good idea to add the proper header, as in most cases the server is expecting it from a POST request.

The code for adding the header and sending the POST request is:

```
request.setRequestHeader("Content-Type",
 "application/x-www-form-urlencoded;  charset=UTF-8");
request.send(queryString);
```

If we entered the raw `queryString` value as a parameter, the method call would look like:
`send("firstname=Bruce&lastname=Perry&gender=M&country=USA");`

Ogling the Result

Once your application POSTs data, then you want to display the result to the Web users. This is the responsibility of the `handleResponse()` function (remember the code in the `initReq()` function: `request.onreadystatechange=handleResponse;?). When the request object's` readyState@ property has a value of 4, signifying that the object's operations are complete, our code checks the HTTP response status for the value 200.

This value indicates that the HTTP request has succeeded. Then the`responseText` is displayed in an `alert` window. This is somewhat anticlimactic, but I thought I'd keep this hack's response handling simple, because so many other hacks do something cooler and more complex with it!

```
//event handler for XMLHttpRequest
function handleResponse(){
 if(request.readyState == 4){
 if(request.status == 200){
 alert(request.responseText);
 } else {
 alert("A problem occurred with communicating between "+
```

```
 "the XMLHttpRequest object and the server program.");
 }
 }//end outer if
}
```

Figure 1-2 shows what the alert window looks like after the response is received

Use A Separate Library For XML Http Request

Break out the code that initializes the request object and sends requests into its own JavaScript file.

In order to cleanly separate the concerns of big Ajax applications, create a separate file that manage the XMLHttpRequest object, then import that file into every web page that needs it. At the very least any changes that are necessary with how the code sets up the request object only have to be made in this file, as opposed to in every JavaScript file that uses Ajax-style requests.

This hack stores all of the request-object related code in a file http_request.js. Any web page that uses XMLHttpRequest can then import this file in the following way.

```
<script type="text/javascript" src="js/http_request.js"></script>
```

Here's the code for the file, a mere 71 lines including all the comments.

```
var request = null;
/* Wrapper function for constructing a Request object.
Parameters:
reqType: The HTTP request type such as GET or POST.
url: The URL of the server program.
asynch: Whether to send the request asynchronously or not.
respHandle: The name of the function that will handle the response.
Any fifth parameters represented as arguments[4] are the data a
POST request is designed to send. */
function httpRequest(reqType,url,asynch,respHandle){
//Mozilla-based browsers
if(window.XMLHttpRequest){
request = new XMLHttpRequest();
//if the reqType parameter is POST, then the
//5th argument to the function is the POSTed data
if(reqType.toLowerCase() != "post") {
initReq(reqType, url,asynch,respHandle);
} else {
//the POSTed data
var args = arguments[4];
if(args != null && args.length > 0){
initReq(reqType,url,asynch,respHandle,args);
}
}
} else if (window.ActiveXObject){
request=new  ActiveXObject("Msxml2.XMLHTTP");
if (! request){
request=new ActiveXObject("Microsoft.XMLHTTP");
```

```
}
if(request){
//if the reqType parameter is POST, then the
//4th argument to the function is the POSTed data
if(reqType.toLowerCase() != "post") {
initReq(reqType,url,asynch,respHandle);
} else {
var args = arguments[4];
if(args != null && args.length > 0){
initReq(reqType,url,asynch,respHandle,args);
}
}
} else {
alert("Your browser does not permit the use of all "+
"of this application's features!");}
} else {
alert("Your browser does not permit the use of all "+
"of this application's features!");}
}
/* Initialize a Request object that is already constructed */
function initReq(reqType,url,bool,respHandle){
try{
/* Specify the function that will handle the HTTP response */
request.onreadystatechange=respHandle;
request.open(reqType,url,bool);
//if the reqType parameter is POST, then the
//5th argument to the function is the POSTed data
if(reqType.toLowerCase() == "post") {
request.setRequestHeader("Content-Type",
"application/x-www-form-urlencoded; charset=UTF-8");
request.send(arguments[4]);
} else {
request.send(null);
}

} catch (errv) {
alert(
"The application cannot contact "+
"the server at the moment. "+
"Please try again in a few seconds.\n"+
"Error detail: "+errv.message);
}
}
```

The applications that use this code call the httpRequest() function with four or five (with POST requests) parameters. You will read lots of examples of calling this function in the other hacks; here is another.

```
var _url = "http://www.parkerriver.com/s/sender";
var _data="first=Bruce&last=Perry&middle=W";
httpRequest("POST",_url,true,handleResponse,_data);
```

The code comments describe the meaning of each of these parameters. The last parameter represents the data that accompanies a POST request.

NOTE

A POST HTTP request includes the posted data beneath the request header information, instead of appending parameter name/values on to the URL, as in a GET.

If the code is not using POST, then the client code only uses the first four parameters. The fourth parameter can be either the name of a function that is declared in the client code (i.e., this response handling function appears outside of the http_request.js file), or a function literal. The latter option involves defining a function inside of a function call, which is often awkward and difficult to read. However, it is sensible in situations where the HTTP response handling is short and simple, as in:

```
var _url = "http://www.parkerriver.com/s/sender";
//a debugging set-up
httpRequest("POST",_url,true,function(){alert(request.responseText);});
```

httpRequest() initiates the same browser detection and set up of XMLHttpRequest for Internet Explorer and non-Microsoft browsers as Hack #1 described. initReq() handles the second step of setting up the request object: specifying the onreadystatechange event handler (see Hack #1), and calling the open() and send() methods to make an HTTP request. The code traps any errors or exceptions thrown by these request method calls using a try/catch statement. For example, if the code calls open() with a URL specifying a different host than the one used to download the enclosing web page, the try/catch statement will catch the error and pop up an alert() window.

Finally, as long as the web page imports http_request.js, then the request variable is available to code external to the imported file; request is in effect a global variable.

request is thus reserved as a variable name, because local variables that use the var keyword will supercede with unintentional consequences the globally used request, as in:

```
function handleResponse(){

//supercedes the imported request variable

var request = null;

try{

if(request.readyState 4){ if(request.status 200){...
```

Receive Data As XML

Ajax and server programs provide a DOM Document object ready to go.

Many technologies are exchanging data as Extensible Markup Language (XML), mostly because XML is a standardized and extensible format that the software world has generally agreed upon.

This allows different parties to use existing or well-known technologies to generate, send, and receive XML, without having to adapt to the software tools that the party with whom they are exchanging XML data is using. An example is a Global Positioning System (GPS) device that can share the data it has recorded about a hike or bike ride with a location-aware web application. You just stick the USB cable attached to the GPS device into the USB computer port, launch software that sends the device data to the web, and that's it. This data format is usually an XML language that has been defined already for GPS software. The web application and the GPS device "already speak the same language."

Although this book is not the place for an extensive introduction to XML, you have probably seen these text files in one form or the other. XML is used as a "meta" language that describes and categorizes specific types of information. XML data starts with an optional XML declaration (e.g., <?xml version="1.0" encoding="UTF-8"?>), followed by a root element and zero or more child elements. An example is:

```
<?xml version="1.0" encoding="UTF-8"?>
<gps>
<gpsMaker>Garmin</gpsMaker>
<gpsDevice>
Forerunner 301
</gpsDevice>
</gps>
```

`gps` is the root element, and `gpsMaker` and `gpsDevice` are child elements. Ajax and the request object can receive data as XML, which is very useful for handling web-services responses that use XML Once the HTTP request is complete, the request object has a property named `responseXML`. This object is a Document Object Model (DOM) `Document` object that your Ajax application can use.

```
function handleResponse(){
 if(request.readyState == 4){
 if(request.status == 200){
 var doc = request.responseXML;
...
}
```

In the code sample, the `doc` variable is a DOM `Document` object. This hack receives XML from a server then initiates a little DOM programming with the `Document` object to pull out some information

from the XML.

NOTE

If you just want to see the raw XML text, use the `request.responseText` property instead.

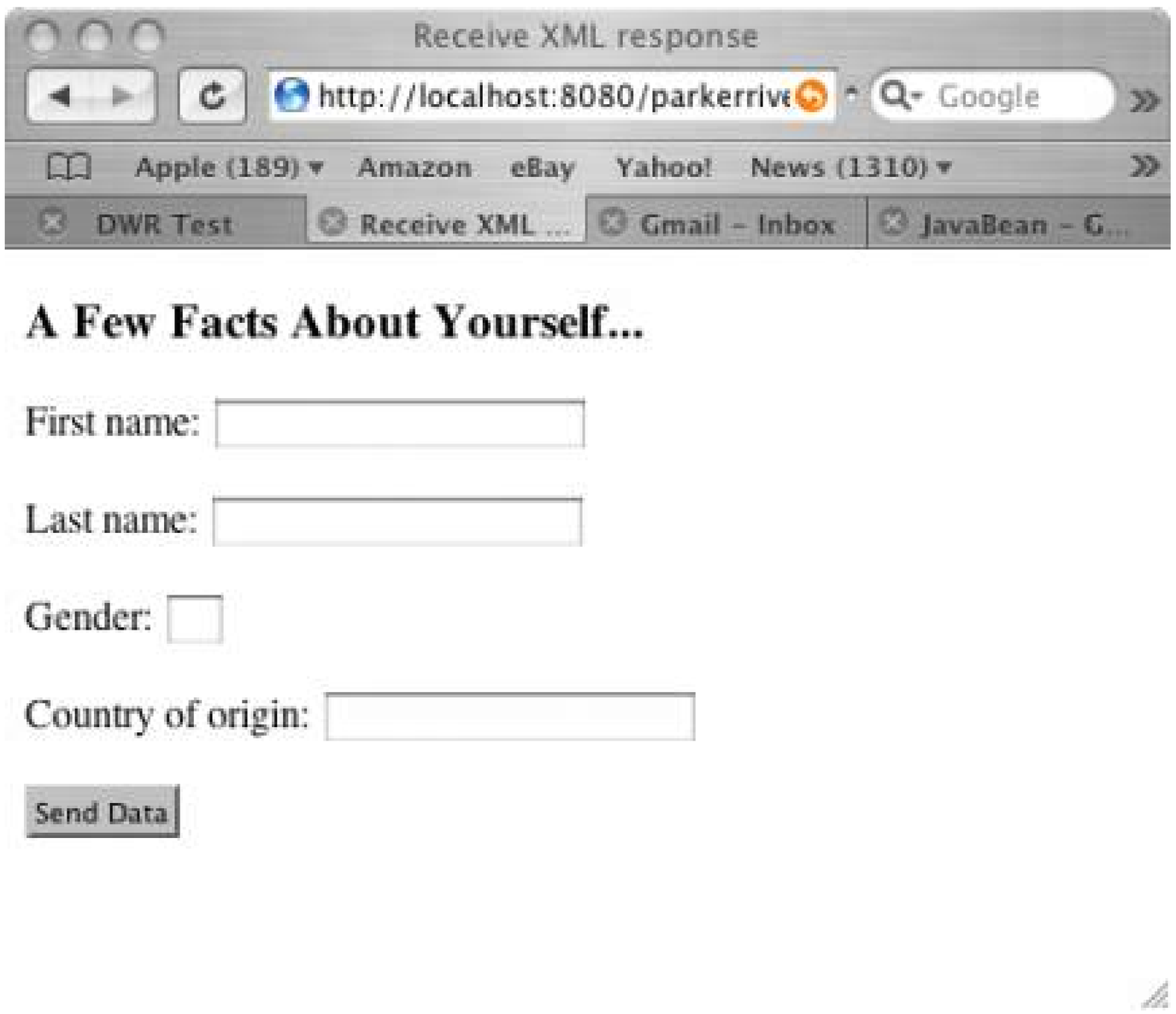
The HTML file is basically the same one we have been using throughout this chapter. We write a `div` element at the end of the HTML where the code will display information about the returned XML. Here's the code for the HTML file.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
 "http://www.w3.org/TR/1999/REC-html401-19991224/strict.dtd">
<html>
<head>
 <script type="text/javascript" src="js/hack3.js"></script>
 <meta http-equiv="content-type" content="text/html; charset=utf-8">
 <title>Receive XML response</title>
</head>
<body>
<h3>A Few Facts About Yourself...</h3>
<form action="javascript:void%200" onsubmit=
"setQueryString();sendData();return false">
 <p>First name: <input type="text" name="firstname" size="20"> </p>
 <p>Last name: <input type="text" name="lastname" size="20"> </p>
 <p>Gender: <input type="text" name="gender" size="2"> </p>
 <p>Country of origin: <input type="text" name="country" size="20"> </p>
<p><button type="submit">Send Data</button></p>

<div id="docDisplay"></div>

</form>
</body>
</html>
```

NOTE

You may be wondering about the weird-looking form action="javascript:void%200" part. Since we are calling JavaScript functions when the form is submitted, we do not want to give the `action` attribute anything but a JavaScript URL that has no return value, as in "javascript:void 0." We have to encode the space between void and 0, which is where the "%20" comes from. If the user turns off JavaScript in their browser, then clicking the submit button on the form will have no effect, because the `action` attribute does not point to a valid URL. In addition, certain HTML validators will display warnings if you use action="". Another way of writing this code is to include the function calls as part of the `window.onload` event handler in the JavaScript .js file, which is the approach used by most of the hacks.

Figure 1-3 shows what the page looks like before the user enters any information:



The JavaScript code in the hack3.js file POSTs its data to a server application, which sends back a response in XML format. Like other examples in this chapter, the server program echoes the parameter names and values back to the client, as in `<params><firstname>Bruce</firstname></params>`. This technique suits our purpose for showing a simple example of programming XML in an Ajax application.

```
var request;
var queryString; //will hold the POSTed data

function sendData(){
 var  url="http://localhost:8080/parkerriver/s/sender";
 httpRequest("POST",url,true);
}

//event handler for XMLHttpRequest
function handleResponse(){
 if(request.readyState == 4){
 if(request.status == 200){
 var doc = request.responseXML;
 var info = getDocInfo(doc);
```

```
 stylizeDiv(info,document.getElementById("docDisplay"));
 } else {
 alert("A problem occurred with communicating between "+
 "the XMLHttpRequest object and the server program.");
 }
 }//end outer if
}

/* Initialize a Request object that is already constructed */
function initReq(reqType,url,bool){
 /* Specify the function that will handle the HTTP response */
 request.onreadystatechange=handleResponse;
 request.open(reqType,url,bool);
 request.setRequestHeader("Content-Type",
 "application/x-www-form-urlencoded;  charset=UTF-8");
 /* Only works in Mozilla-based browsers */
 //request.overrideMimeType("text/XML");
 request.send(queryString);
}

/* Wrapper function for constructing a Request object.
 Parameters:
 reqType: The HTTP request type such as GET or POST.
 url: The URL of the server program.
 asynch: Whether to send the request asynchronously or not. */
function httpRequest(reqType,url,asynch){
 //Mozilla-based browsers
 if(window.XMLHttpRequest){
 request = new XMLHttpRequest();
 initReq(reqType,url,asynch);
 } else if (window.ActiveXObject){
 request=new  ActiveXObject("Msxml2.XMLHTTP");
 if (! request){
 request=new ActiveXObject("Microsoft.XMLHTTP");
 }
 if(request){
 initReq(reqType,url,asynch);
 } else {
 alert(
 "Your browser does not permit the use of all "+
 "of this application's features!");}
 } else {
 alert(
 "Your browser does not permit the use of all "+
 "of this application's features!");}
}

function setQueryString(){
 queryString="";
 var frm = document.forms[0];
 var numberElements = frm.elements.length;
 for(var i = 0; i < numberElements; i++) {
```

```
 if(i < numberElements-1) {
 queryString += frm.elements[i].name+"="+frm.elements[i].value+"&";
 } else {
 queryString += frm.elements[i].name+"="+frm.elements[i].value;
 }

 }
}
/* provide the div elements content dynamically. We can add
 style information to this function if we want to jazz up the div */
function stylizeDiv(bdyTxt,div){
 //reset DIV content
 div.innerHTML="";
 div.style.backgroundColor="yellow";
 div.innerHTML=bdyTxt;
}

/* Get information about an XML document via a DOM Document object */
function getDocInfo(doc){
 var root = doc.documentElement;
 var info = "<h3>Document root element name: <h3 />"+ root.nodeName;
 var nds;
 if(root.hasChildNodes()) {
 nds=root.childNodes;
 info+= "<h4>Root node's child node names/values:<h4/>";
 for (var i = 0; i < nds.length; i++){
 info+= nds[i].nodeName;
 if(nds[i].hasChildNodes()){
 info+= " : \""+nds[i].firstChild.nodeValue+"\"&#x00A;";
 } else {
 info+= " : Empty&#x00A;";
 }
 }
 }
 return info;
}
```

NOTE

Mozilla Firefox can use the `request.overrideMimeType()` function to force the interpretation of the response stream as a certain mime type, as in request.overrideMimeType("text/XML"). Internet Explorer's request object does not have this function. This function call does not work with Safari 1.3 either.

After the code POSTs its data and receives a response, it calls a method named`getdocInfo()`, which builds a `string` displaying some information about the XML document and its child or sub elements.

```
var doc = request.responseXML;
var info = getDocInfo(doc);
```

The `geTDocInfo()` function gets a reference to the root XML element (`var root = doc.documentElement;`), then it builds a `string` specifying the name of the root element and information about any of its child nodes or elements, such as the child node name and value. The code then feeds this information to the `stylizeDiv()` method. The `stylizeDiv()` method uses the `div` element at the end of the HTML page to dynamically display the gathered information.

```
function stylizeDiv(bdyTxt,div){
 //reset DIV content
 div.innerHTML="";
 div.style.backgroundColor="yellow";
 div.innerHTML=bdyTxt;
}
```

Figure 1-4 what the Web page looks like after the application receives the XML response.

Delving into XML return values

The core DOM API offered by the browser's JavaScript implementation allows developers a powerful tool for programming complex XML return values

E B V N

We are Vietnames

Get Plain Old Strings

Manage weather readings, stock quotes, web-page scrapings, or similar non-XML data as plain old strings.

The request object has the perfect property for the web applications that do not have to handle server return values as XML. `request.responseText`. This hack asks the user to choose a stock symbol, and the server returns the stock price for display. The code handles the return value as a `string`.

> NOTE
>
> A variation on this program in the next hack requires the stock prices to be handled as numbers.

First, here is the HTML for the web page. It imports JavaScript code in a file named hack9.js.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
 "http://www.w3.org/TR/1999/REC-html401-19991224/strict.dtd">
<html>
<head>
 <script type="text/javascript" src="js/hack9.js"></script>
 <meta http-equiv="content-type" content="text/html; charset=utf-8">
 <title>Choose a stock</title>
</head>
<body>
<h3>Stock prices</h3>
<form action="javascript:void%200" onsubmit=
"getStockPrice(this.stSymbol.value);return false">
 <p>Enter stock symbol: <input type="text" name=
"stSymbol" size="4"><span id="stPrice"></span></p>
<p><button type="submit">Get Stock Price</button></p>
</form>
</body>
</html>
```

Figure 1-5 shows the web page as displayed in Firefox. The user enters a symbol such as "GRMN" (case insensitive), clicks the button, and the JavaScript fetches an associated stock price and displays it within a `span` element to the right of the text field.

Instantaneously displaying a stock price

The function that sets the request process in motion is`getStockPrice()`. This function takes the value of the text field named `stSymbol` and returns the stock price. The function uses the request object to talk to a server component, which fetches the actual stock price. Here is the JavaScript code:

```
var request;
var symbol; //will hold the stock symbol

function getStockPrice(sym){
 symbol=sym;
 if(sym) {
 var url=
 "http://localhost:8080/parkerriver/s/stocks?symbol="+sym;
 httpRequest("GET",url,true);
 }
}

//event handler for XMLHttpRequest
function handleResponse(){
 if(request.readyState == 4){
 if(request.status == 200){
 /*Grab the result as a string*/
 var stockPrice = request.responseText;
 var info = "&#171;The price is: $"+stockPrice+"&#187;";
 document.getElementById("stPrice").style.fontSize="0.9em";
 document.getElementById("stPrice").style.backgroundColor="yellow";
 document.getElementById("stPrice").innerHTML=info;
```

```
 } else {
 alert("A problem occurred with communicating between "+
 the XMLHttpRequest object and the server program.");
 }
 }//end outer if
}

/* Initialize a Request object that is already constructed */
function initReq(reqType,url,bool){
 /* Specify the function that will handle the HTTP response */
 request.onreadystatechange=handleResponse;
 request.open(reqType,url,bool);
 request.send(null);
}

/* Wrapper function for constructing a Request object.
 Parameters:
 reqType: The HTTP request type such as GET or POST.
 url: The URL of the server program.
 asynch: Whether to send the request asynchronously or not. */
function httpRequest(reqType,url,asynch){
 //Mozilla-based browsers
 if(window.XMLHttpRequest){
 request = new XMLHttpRequest();
 initReq(reqType,url,asynch);
 } else if (window.ActiveXObject){
 request=new  ActiveXObject("Msxml2.XMLHTTP");
 if (! request){
 request=new ActiveXObject("Microsoft.XMLHTTP");
 }
 if(request){
 initReq(reqType,url,asynch);
 } else {
 alert("Your browser does not permit the use of all "+
 "of this application's features!");}
 } else {
 alert("Your browser does not permit the use of all "+
 "of this application's features!");}
}
```

The function `getStockPrice()` wraps a call to the function `httpRequest()`, which is responsible for setting up the request object. If you have already read through some of this chapter's other hacks, you will recognize the `handleResponse()` function as enclosing much of the interesting action.

`Hack#1 explains the <literal>httpRequest() </literal>function in more detail.`

If the request is complete (if `request.readyState` has a value of 4) and the HTTP response status is 200 (meaning that the request has succeeded), then the code grabs the server response as the `request.responseText` property value. The code then uses Document Object Model (DOM) programming to display the stock price with some CSS style-related attributes.

```
document.getElementById("stPrice").style.fontSize="0.9em";
document.getElementById("stPrice").style.backgroundColor="yellow";
document.getElementById("stPrice").innerHTML =info;
```

The `style` attributes make the font size a little bit smaller than the user's preferable browser font size, and specify yellow as the background color of the text display. The`innerHtml` property of the `span` element is set to the stock price within double-angle quotation characters.

NOTE

This application would typically take more than a few seconds to return its server value. Therefore, developers may consider including a progress indicator, as explained in Display a Progress Indicator For Web Users.

Receive Data As A Number

Do numerical calculations that depend on the request object's return value as a number.

This hack receives a stock quote as a number, then dynamically displays the total value of a stock holding based on the number of shares a user enters. If the server does not send a valid number then the application displays an error message to the user.

The great advantage of Ajax technology is in receiving discrete values rather than entire web pages from a server. The discrete information you receive might have to be used as a number, rather than as a `string` or some other object. JavaScript is usually pretty smart about converting values to number types without your intervention, but still you don't want the application to multiply an innocent investor's share quantity by `undefined` or some other weird data the server returns!

This hack checks that the user has entered a proper number for a "number of shares" value. The code also checks the server return value to make sure it is numerically valid. Then the hack dynamically displays the stock price and total value of the shares in the user's browser.

First, figure 1-6 shows what the browser form looks like:

Discover a total share value

The following code shows the HTML for the Web page.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
 "http://www.w3.org/TR/1999/REC-html401-19991224/strict.dtd">
<html>
<head>
 <script type="text/javascript" src="/parkerriver/js/hack4.js">
 </script>
 <meta http-equiv="content-type" content="text/html; charset=utf-8">
 <title>Tally your stocks</title>
</head>
<body>
<h3>Your total Stock Holdings</h3>
<form action="javascript:void%200" onsubmit=
"getStockPrice(this.stSymbol.value,this.numShares.value);return false">
<p>Enter stock symbol: <input type="text" name=
"stSymbol" size="4"> <span id="stPrice"></span></p>
 <p>Enter share amount: <input type="text" name="numShares" size="10"> </p>
<p><button type="submit">Get Total Value</button></p>
<div id="msgDisplay"></div>
</form>
</body>
</html>
```

When the user clicks the `Get Total Value` button, this action triggers the `form` element's `onsubmit` event. The event handler for this event is the `getStockPrice()` function. This function takes the stock symbol and the number of shares as its two parameters. The `return false` part of the event-handling code cancels the browser's typical submission of the form values to the URL specified by the `form` tag's `action` attribute.

Number Crunching

Now let's look at the JavaScript code, which the HTML file imports as part of the hack4.js file.

```
var request;
var symbol; //will hold the stock symbol
var numberOfShares;

function getStockPrice(sym,shs){
 if(sym && shs) {
 symbol=sym;
 numberOfShares=shs;
 var   url="http://localhost:8080/parkerriver/s/stocks?symbol="+sym;
 httpRequest("GET",url,true);
 }
}
//event handler for XMLHttpRequest
```

```
function handleResponse(){
 if(request.readyState == 4){
 alert(request.status);
 if(request.status == 200){
 /*Check if the return value is actually a number.
 If so, multiple by the number
 of shares and display the result*/
 var stockPrice = request.responseText;
 try{
 if(isNaN(stockPrice)) { throw new Error(
 "The returned price is an invalid number.");}
 if(isNaN(numberOfShares)) { throw new Error(
 "The share amount is an invalid number.");}
 var info = "Total stock value: "+ calcTotal(stockPrice);
 displayMsg(document.getElementById("msgDisplay"),info,"black");
 document.getElementById("stPrice").style.fontSize="0.9em";
 document.getElementById("stPrice").innerHTML ="price: "+stockPrice;
 } catch (err) {
 displayMsg(document.getElementById("msgDisplay"),
 "An error occurred with symbol "+symbol+ ": "+
 err.message,"red");
 }
 } else {
 alert(
 "A problem occurred with communicating between the XMLHttpRequest "+
 "object and the server program.");
 }
 }//end outer if
}

/* Initialize a Request object that is already constructed */
function initReq(reqType,url,bool){
 /* Specify the function that will handle the HTTP response */
 request.onreadystatechange=handleResponse;
 request.open(reqType,url,bool);
 request.send(null);
}

/* Wrapper function for constructing a Request object.
 Parameters:
 reqType: The HTTP request type such as GET or POST.
 url: The URL of the server program.
 asynch: Whether to send the request asynchronously or not. */
function httpRequest(reqType,url,asynch){
 //Mozilla-based browsers
 if(window.XMLHttpRequest){
 request = new XMLHttpRequest();
 initReq(reqType,url,asynch);
 } else if (window.ActiveXObject){
 request=new  ActiveXObject("Msxml2.XMLHTTP");
 if (! request){
 request=new ActiveXObject("Microsoft.XMLHTTP");
```

```
 }
 if(request){
 initReq(reqType,url,asynch);
 } else {
 alert(
 "Your browser does not permit the use of all "+
 "of this application's features!");}
 } else {
 alert(
 "Your browser does not permit the use of all "+
 "of this application's features!");}
}

function calcTotal(price){
 return stripExtraNumbers(numberOfShares * price);
}
/*Strip any characters beyond a scale of four characters
past the decimal point, as in 12.3454678 */
function stripExtraNumbers(num) {
 //check if the number's already okay
 //assume a whole number is valid
 var numStr = num.toString();//working with the number as a string
 var indx =numStr.indexOf(".");
 if(indx == -1) { return num; }
 var chArray = numStr.split(".");
 //the second array member includes all the chars after the decimal point
 if(chArray[1].length <= 4) { return num; }
 //use the Number.toPrecision method to restrict the
 //decimal-point numbers to the length of the characters
 //prior to the decimal point plus four
 return num.toPrecision(chArray[0].length + 4);
}

function displayMsg(div,bdyText,txtColor){
 //reset DIV content
 div.innerHTML="";
 div.style.backgroundColor="yellow";
 div.style.color=txtColor
 div.innerHTML=bdyText;
}
```

NOTE

The getStockprice() function wraps a call to httpRequest(), which sends the request for a particular stock price to the server. See Hack #1 for a description of constructing the request object with httpRequest(). Hack #3 describes using a pre-established library to handle the request object.

All of the number-crunching starts in `handleResponse()`. First, the code receives the response as a

`string` in `var stockPrice = request.responseText`. Then the code tests the validity of the `stockPrice` variable using a method that is part of JavaScript's core API: `isNaN()`. This is the best way to test whether a `string` value in JavaScript can represent a valid number. For example, `isNaN("goodbye")` returns `true`, because "goodbye" cannot be converted to a number. The code also tests the number-of-shares value with this function.

If either of the methods return `true` indicating invalid number values, then the code throws an exception, which is another way of declaring "we can't use these values; get them out of here!" In these cases, the web page displays an error message to the user.

> NOTE
>
> Exception handling with Ajax is covered in the next hack.

However, we're not yet finished with our number crunching. The`calcTotal()` function then multiplies the share total by the stock price in order to display the total value to the user.

To make sure that the numerical display of the value is friendly enough to the eye (in terms of the U.S. stock exchange), the `stripExtraNumbers()` function only keeps no more than four characters to the right of the decimal point.

> NOTE
>
> Even though $10.9876 may look a little weird, since stock prices are sometimes displayed with four or more characters to the right of the decimal point, we decided to allow this display with the total share value.

JavaScript stores the parameter passed into the `stripExtraNumbers()` function as a number. Ironically (we've been so concerned with dealing with number values only!), this function must "cast' the value back to the `string` type so that we can discover the number's format. This is the purpose of the `var numStr = num.toString()` code. After calling the `toString()` method of the `Number` object, the variable `numStr` now holds the number as a string (as in "12000" instead of 12000). Therefore, `indexOf()` and other `string`-related functions may be called on it.

Finally, the code uses the `Number.toPrecision()` method to only return a number with the total number of significant digits represented by the number of characters to the left of the decimal point plus four.

DOM-inating

The code uses Document Object Model (DOM) programming to dynamically display new text and values on the page, al without having to make new server calls and refresh the entire page!

```
displayMsg(document.getElementById("msgDisplay"),info,"black");
document.getElementById("stPrice").style.fontSize="0.9em";
document.getElementById("stPrice").innerHTML ="price: "+stockPrice;
```

This bit of code within the `handleResponse()` function calls the `displayMsg()` function to show the user the total share value. The code also dynamically embeds the stock price just to the right of the text field where the user entered the stock symbol. All the code does here is get a reference to the `div` element with `id stPrice`, make its font-size style property a little smaller than the web user's font setting, then set the `div's innerHTML` property. Easy!

The `displayMsg()` function is also simple. It has a parameter that represents the font color, which allows the code to set the font color "red" for error messages.

```
function displayMsg(div,bdyText,txtColor){
 //reset DIV content
 div.innerHTML="";
 div.style.backgroundColor="yellow";
 div.style.color=txtColor
 div.innerHTML=bdyText;
}
```

Figure 1-7 shows what the page looks like when the user requests a stock value.

Tallying your investment

Figure 1-8 shows an example error message, in case the user enters values that cannot be used as

numbers, or if the server returns invalid values.

Having a bad number day

Receive Data In JSON Format

Ajax can receive data in efficient and powerful JavaScript Object Notation.

How would you like to use Ajax and receive data from the server as plain old JavaScript objects? Well, y
hack takes information entered by a Web user and initiates a server roundtrip, which returns the data in

JSON is simple and straightforward, which is probably why a lot of developers like it. JSON formatted da
values. An example is a server program that pulls product information from a database or cache and ret
represented by:

1. An opening curly brace: "{";
2. One or more property names separated from its value by a colon character;
3. Property/value pairs separated by commas; and
4. A closing curly brace.

The values of each property in the object can be:

- Simple strings like "hello."
- Arrays, such as [1,2,3,4].
- Numbers
- The values `true` , `false` , or `null` .
- Other objects, as in composition, an object containing one or more objects.

NOTE

See http://www.json.org .

This is exactly the format of an `Object` literal in JavaScript. Based on the information Hack #2 asked the

```
{
firstname:"Bruce",
lastname:"Perry",
gender:"M",
country:"USA"
}
```

We're going to use a similar HTML page and ask the user for the same information; however, this hack v
the server. Two `div` elements at the bottom of the HTML page will respectively show the JSON return va

in a more friendly fashion.

Here's the code for the HTML page:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
 "http://www.w3.org/TR/1999/REC-html401-19991224/strict.dtd">
<html>
<head>
 <script type="text/javascript" src="js/hack5.js"></script>
 <meta http-equiv="content-type" content="text/html; charset=utf-8">
 <title>Receive JSON response</title>
</head>
<body>
<h3>A Few Facts About Yourself...</h3>
<form action="javascript:void%200" onsubmit=
"setQueryString();sendData();return false">
 <p>First name: <input type="text" name="firstname" size="20"> </p>
 <p>Last name: <input type="text" name="lastname" size="20"> </p>
 <p>Gender: <input type="text" name="gender" size="2"> </p>
 <p>Country of origin: <input type="text" name="country" size="20"> </p>
<p><button type="submit">Send Data</button></p>
<div id="json"></div>
<div id="props"></div>
</form>
</body>
</html>
```

Figure 1-9 shows what the Web page looks like:

JSON is calling.

The JavaScript code is imported by the `script` tag and specified by the file hack5.js. The JavaScript sen
we won't go into great detail beyond showing you the code. Here's the entire code piece for this hack, th
JavaScript object.

```
var request;
var queryString; //will hold the POSTed data

function sendData(){
 url="http://localhost:8080/parkerriver/s/json";
 httpRequest("POST",url,true);
}

//event handler for XMLHttpRequest
function handleJson(){
 if(request.readyState == 4){
 if(request.status == 200){
 var resp = request.responseText;
 var func = new Function("return "+resp);
 var objt = func();
 var div = document.getElementById("json");
 stylizeDiv(resp,div);
 div = document.getElementById("props");
 div.innerHTML="<h4>In object form...</h4>"+
 "<h5>Properties</h5>firstname= "+
 objt.firstname +"&#x00A;lastname="+
 objt.lastname+ "&#x00A;gender="+
 objt.gender+ "&#x00A;country="+
 objt.country;
```

```
 } else {
 alert("A problem occurred with communicating between "+
 "the XMLHttpRequest object and the server program.");
 }
 }//end outer if
}

/* Initialize a Request object that is already constructed */
function initReq(reqType,url,bool){
 /* Specify the function that will handle the HTTP response */
 request.onreadystatechange=handleJson;
 request.open(reqType,url,bool);
 request.setRequestHeader("Content-Type",
 "application/x-www-form-urlencoded;  charset=UTF-8");
 request.send(queryString);
}

/* Wrapper function for constructing a Request object.
 Parameters:
 reqType: The HTTP request type such as GET or POST.
 url: The URL of the server program.
 asynch: Whether to send the request asynchronously or not. */

function httpRequest(reqType,url,asynch){
 //Mozilla-based browsers
 if(window.XMLHttpRequest){
 request = new XMLHttpRequest();
 initReq(reqType,url,asynch);
 } else if (window.ActiveXObject){
 request=new  ActiveXObject("Msxml2.XMLHTTP");
 if (! request){
 request=new ActiveXObject("Microsoft.XMLHTTP");
 }
 if(request){
 initReq(reqType,url,asynch);
 } else {
 alert(
 "Your browser does not permit the use of all "+
 "of this application's features!");}
 } else {
 alert(
 "Your browser does not permit the use of all "+
 "of this application's features!");}
}

function setQueryString(){
 queryString="";
 var frm = document.forms[0];
 var numberElements = frm.elements.length;
 for(var i = 0; i < numberElements; i++) {
 if(i < numberElements-1) {
 queryString += frm.elements[i].name+"="+frm.elements[i].value+"&";
```

```
 } else {
 queryString += frm.elements[i].name+"="+frm.elements[i].value;
 }

 }
}

function stylizeDiv(bdyTxt,div){
 //reset DIV content
 div.innerHTML=" ";
 div.style.fontSize="1.2em";
 div.style.backgroundColor="yellow";
 div.appendChild(document.createTextNode(bdyTxt));
}
```

As in this chapter's previous hacks, the `initReq()` function initializes the request object and sends an H

```
request.onreadystatechange=handleJson;
```

The event-handling function for when the response is ready is called `handleJson()` . The response is a JS
As is, JavaScript interprets this returned text as a `string` object. Therefore, the code initiates an openin
`object` literal. By the way, in this hack, the server takes the request parameters and reformats the para
the reformatted data as its response.

> NOTE
>
> We have not included special error-handling code here, as these elements require further explanati

Here's the `handleJson()` code:

```
//event handler for XMLHttpRequest
function handleJson(){
 if(request.readyState == 4){
 if(request.status == 200){
 var resp = request.responseText;
 var func = new Function("return "+resp);
 var objt = func();
 var div = document.getElementById("json");
 stylizeDiv(resp,div);
 div = document.getElementById("props");
 div.innerHTML="<h4>In object form...</h4>"+
 "<h5>Properties</h5>firstname= "+
 objt.firstname +"&#x00A;lastname="+
 objt.lastname+ "&#x00A;gender="+
 objt.gender+ "&#x00A;country="+
 objt.country;
 } else {
 alert(
```

```
 "A problem occurred with communicating between "+
 "the XMLHttpRequest object and the server program.");
 }
 }//end outer if
}
```

The variable `resp` refers to the HTTP response text, which JavaScript interprets as a `string` . The tricky s

```
var func = new Function("return "+resp);
```

This code creates a new `Function` object on the fly, and stores the `Function` in a variable named `func` . declared in code, or created as function literals. However, in this case we needed to define a function bo the perfect tool.

NOTE

Thanks to this site for the guidance on this code usage: http://jibbering.com/2002/4/httprequest.h around the web goes like this:

NOTE

```
var resp = request.responseText;
p(note). var obj = eval( "(" + resp + ")" );
p(note). You do not have to use the parentheses characters when using <literal>eval(

p(note). var resp = request.responseText;
p(note). //resp contains something like "[1,2,3,4]"
p(note). var arrObject = eval(resp);
```

The latter code creates a function that returns an `object` literal, representing the server return value. W display server values on the web page with DOM programming. All without complex`object` serialization

```
var objt = func();
var div = document.getElementById("json");
stylizeDiv(resp,div);
div = document.getElementById("props");
div.innerHTML="<h4>In  object  form...</h4><h5>Properties</h5>firstname=  "+
objt.firstname  +"&#x00A;lastname="+
objt.lastname+  "&#x00A;gender="+
objt.gender+  "&#x00A;country="+
objt.country;
```

A variable named `objt` stores the `object` literal. The values are pulled from the object with syntax such

has received a response:

Visualizing JavaScript properties

Handle Request Object Errors

Design your Ajax application to detect any server errors and provide a friendly user message.

Much of the oomph behind Ajax technology is that it allows JavaScript to connect with a server program without the user intervening. However, JavaScript developers often have no control over the server component itself, which could be a web service or other software designed outside of their organization. Even if your application involves your organization's server component, you cannot always be sure that the server is behaving normally, or even that your users are online at the moment they trigger your request object. You have to make sure that your application recovers in the event that the backend program is unavailable,

This hack traps errors and displays a meaningful error message, in the event the Ajax application loses server contact.

This hack address the following exceptional events, and recommends ways for the application to recover from them.

- The web application or server component you are connecting with is temporarily unavailable.
- The server your application is connecting with is down, or its URL has changed unbeknownst to you.
- The server component you connect with has one or more bugs, and it crashes during your connection (yeech!)
- When you call the `open()` method with the Request object, your code uses a different host address than the address from which the user downloaded the your web page. The request object throws an exception in this case when you try to call its`open()` method.

You can use this hack's exception-handling code in any application. This hack uses the stock calculation code from Receive Data As A Number. This hack shows the code that initializes the request object then the exception-handling mechanism.

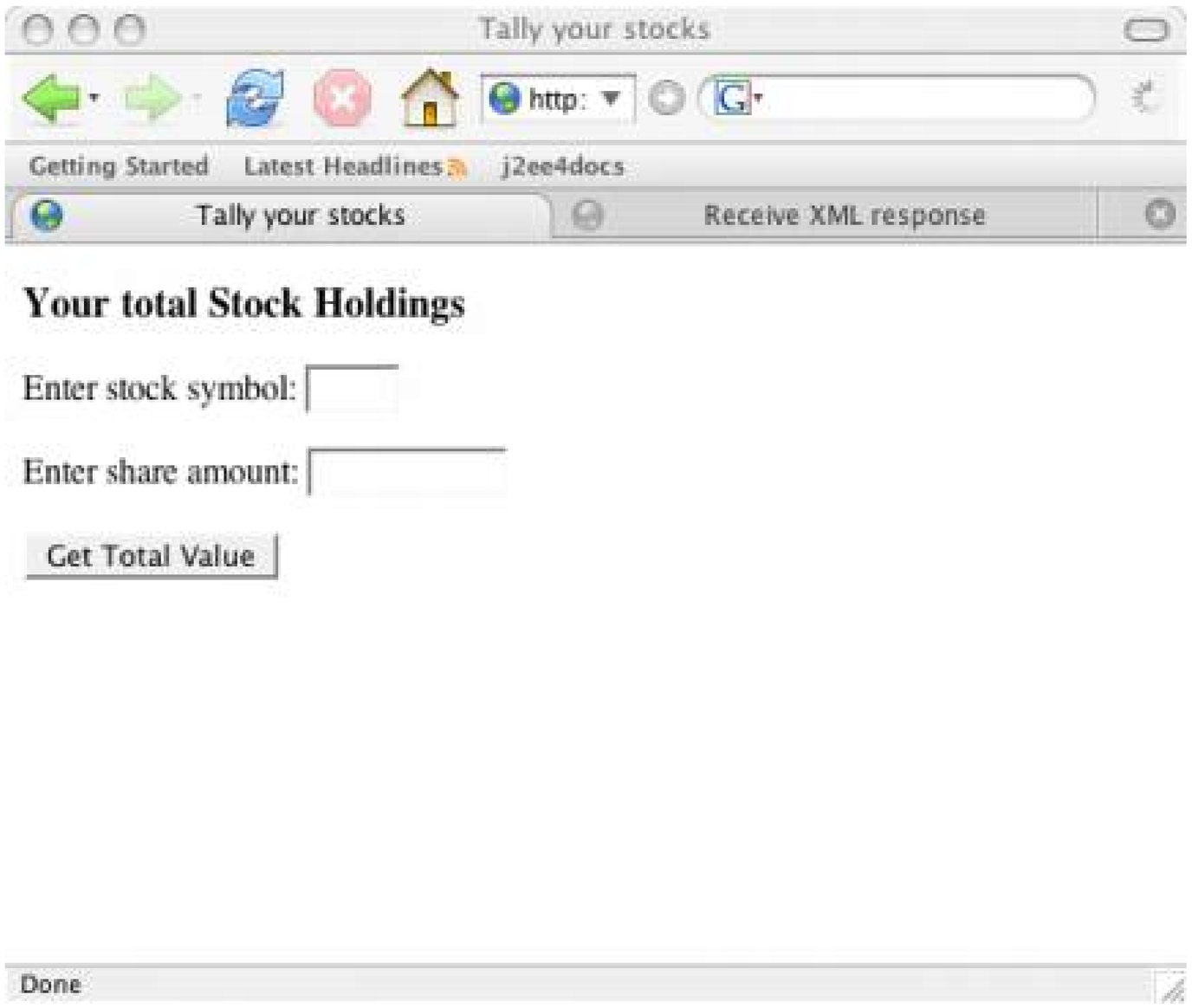
Here's the HTML file that imports the JavaScript code.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
 "http://www.w3.org/TR/1999/REC-html401-19991224/strict.dtd">
<html>
<head>
 <script type="text/javascript" src="js/hack6.js"></script>
 <meta http-equiv="content-type" content="text/html; charset=utf-8">
 <title>Tally your stocks</title>
</head>
<body>
<h3>Your total Stock Holdings</h3>
```

```
<form action="javascript:void%200" onsubmit=
"getStockPrice(this.stSymbol.value,this.numShares.value);return false">
 <p>Enter stock symbol: **<input type="text" name=**
**"stSymbol" size="4">** <span id="stPrice"></span></p>
 <p>Enter share amount: **<input type="text" name="numShares" size="10">** </p>
<p><button type="submit">Get Total Value</button></p>
<div id="msgDisplay"></div>
</form>
</body>
</html>
```

When the user loads this file into their browser they see Figure 1-11.

Request a stock's price



The code we are interested in will be able to trap exceptions involving unavailable applications, backend servers that are down, backend server bugs, and erroneous URLs. The`handleResponse()` function is the event handler for managing the server response, as in `request.onreadystatechange=handleResponse`. The code uses a nested `TRy/catch/finally` statement to deal with invalid numbers handled by the application (see hack#?).

```
function handleResponse(){
 var statusMsg="";
```

```
try{
if(request.readyState == 4){
if(request.status == 200){
/*Check if the return value is actually a number.
If so, multiple by the number
of shares and display the result*/
var stockPrice = request.responseText;

try{
if(isNaN(stockPrice)) { throw new Error(
"The returned price is an invalid number.");}
if(isNaN(numberOfShares)) { throw new Error(
"The share amount is an invalid number.");}
var info = "Total stock value: $"+ calcTotal(stockPrice);
displayMsg(document.getElementById("msgDisplay"),info,"black");
document.getElementById("stPrice").style.fontSize="0.9em";
document.getElementById("stPrice").innerHTML ="price: "+
stockPrice;
} catch (err) {
displayMsg(document.getElementById(
"msgDisplay"),"An error occurred with symbol "+symbol+ ": "
+err.message,"red");
}
} else {
//request.status is 503 if the application isn't available;
// 500 if the application has a bug
alert(
"A problem occurred with communicating between the "
"XMLHttpRequest object "+
"and the server program. Please try again very soon");
}
}//end outer if
} catch (err) {
alert("It does not appear that the server "+
"is available for this application. Please"+
" try again very soon. \nError: "+err.message);

}
}
```

Floored server

A `try` block will trap any exceptions thrown within its curly braces ("{}"). If the code throws an exception, this mechanism executes the code within the associated`catch` block. The inner `try` block, which is designed to manage exceptions thrown in the event of invalid numeric values, is explained by Receive Data As A Number. What happens if the server host is completely down, even though the URL your application uses is otherwise correct?

In this case, the code's attempt to access the`request.status` property will throw an exception, because the request object never received the expected response header from the server and the

status property is not associated with any data.

As a result, the code will display the `alert` window defined in the outer `catch` block. Figure 1-12 depicts what the `alert` window looks like after this type of error.

Oh oh, server down

The code displays a user message, as well as the more techie message associated with the exception You can leave that part of the message out if you desire; it is mainly useful for debugging purposes.

> NOTE
>
> The `err` variable in the code is a reference to the JavaScript `Error` object. The `message` property of this object (as in `err.message`) is the actual error message, a `string` generated by the JavaScript engine.

If you do not include this `TRy/catch/finally` mechanism, then the user sees just an alert window containing the indecipherable error message generated by JavaScript. Once the user dismisses this window (or leaves their desk in frustration), they have no way of knowing what state the application is in.

Backend Application Out to Lunch

Sometimes the application server or host is running okay, but the server component you want to connect with is out of service. In this case, the `request.status` property will be 503 ("Service Unavailable"). Since the `status` property holds a value other than 200, this code will execute the expression contained within the `else` statement block. In other words, the user sees another alert window informing them of the application's status.

```
} else {
 //request.status is 503 if the application isn't available;
 // 500 if the application has a bug
 alert(
```

```
 "A problem occurred with communicating between the XMLHttpRequest object "+
 "and the server program. Please try again very soon");
}
```

This alert also appears if the server component has a bug and crashes. This event typically (such as with the Tomcat servlet container) results in a 500 response status code ("Internal Server Error"). So `response.status` evaluates to 500 instead of 200 ("Okay"). In addition, any 404 response codes involving a static or dynamic component that the server cannot find at the URL you provided, will also be captured with this `TRy` statement.

> NOTE
>
> The `TRy/catch/finally` statement is only available with JavaScript engines of JS version 1.4 or later. The optional `finally` statement block follows the `catch` block. The code enclosed by `finally{ }` executes whether or not an exception is thrown.

Woops, Wrong URL

What if the URL that your Ajax application uses in the `request.open()` method is wrong or has changed? In this case the request.open() call will throw the exception. This is where you have to position your try/catch/finally statement. The code at the top of the next example constructs a request object (see hack# 1). The following function definition initReq()@catches the exception we just mentioned.

```
function httpRequest(reqType,url,asynch){
 //Mozilla-based browsers
 if(window.XMLHttpRequest){
 request = new XMLHttpRequest();
 initReq(reqType,url,asynch);
 } else if (window.ActiveXObject){
 request=new  ActiveXObject("Msxml2.XMLHTTP");
 if (! request){
 request=new ActiveXObject("Microsoft.XMLHTTP");
 }
 if(request){
 initReq(reqType,url,asynch);
 } else {
 alert("Your browser does not permit the use of all "+
 "of this application's features!");}
 } else {
 alert("Your browser does not permit the use of all "+
 "of this application's features!");}
}

/* Initialize a Request object that is already constructed */
function initReq(reqType,url,bool){
 try{
```

```
 /* Specify the function that will handle the HTTP response */
 request.onreadystatechange=handleResponse;
 request.open(reqType,url,bool);
 request.send(null);
 } catch (err) {

 alert(
 "The application cannot contact the server at the moment."+
 " Please try again in a few seconds.");
 }
}
```

Another variation of this error is when the URL you use with the`request.open()` method includes a different host than the host from which the user downloaded the web page. For example, the user downloads the web page from http://www.myorg.com/app, but the URL you use for `open()` is http://www.yourorg.com/. This type of error will also be caught by the code's`try/catch/finally` statement.

NOTE

You can also optionally abort or cancel the request in the catch block with request.abort(). See Set A Time Limit For The HTTP Request and its discussion of setting a timeout for the request, and aborting it in the event the request is not complete within a certain period.

Dig Into The HTTP Response

Display the value of various HTTP response headers in addition to or in lieu of a typical server return value.

An HTTP response header is descriptive information, according to the HTTP 1.1 protocol, that web servers send requestors along with the actual web page or data. If you have already coded with the `XMLHttpRequest` object (see the Introduction), then you know that the `request.status` property equates to an HTTP response status code sent from the server. This is an important number value to check before your page does anything cool with the HTTP response.

> NOTE
>
> These status values can include 200 (request went through okay), 404 (the request file or URL path was not found), or 500 (internal server error).

However, you might want to see some of the other response headers associated with the request, such as the type of web server software associated with the response (the Server response header), or the content type of the response (the `Content-Type`). This hack requests the user to enter a URL, then when they're finished and click TAB or outside of the text field, the browser displays the other available HTTP response headers. As usual with Ajax, this happens without a page refresh.

> NOTE
>
> This request object method only returns a sub-set of response headers, not all of them, including Content-Type, Date, Server, and Content-Length.

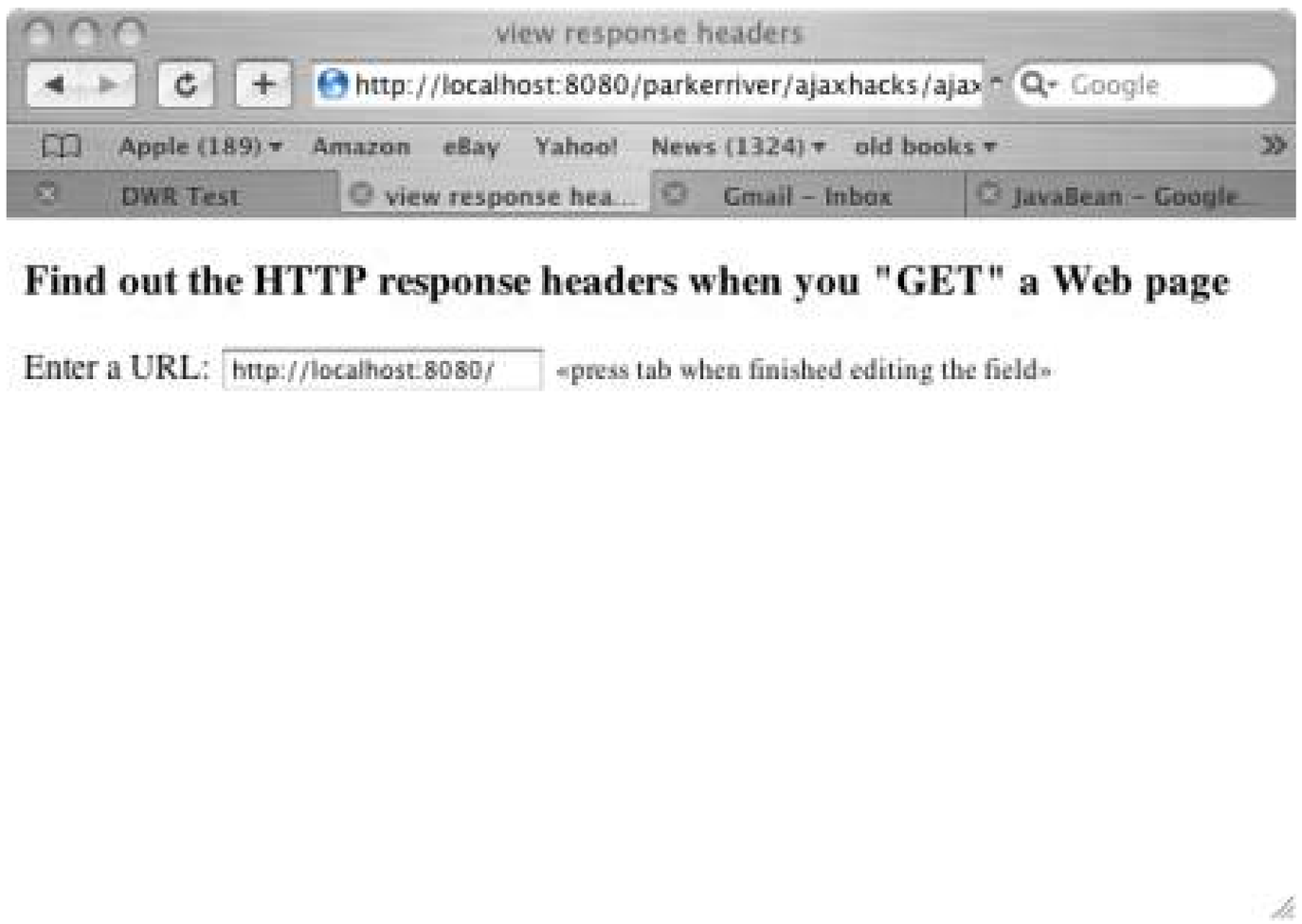
Here is the HTML page code:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
 "http://www.w3.org/TR/1999/REC-html401-19991224/strict.dtd">
<html>
<head>
 <script type="text/javascript" src="js/hack7.js"></script>
 <meta http-equiv="content-type" content="text/html; charset=utf-8">
 <title>view response headers</title>
 <link rel="stylesheet" type="text/css" href="/parkerriver/css/hacks.css">
</head>
<body onload="document.forms[0].url.value=urlFragment">
<h3>Find out the HTTP response headers when you "GET" a Web page</h3>

<form  action="javascript:void%200">
```

```
<p>Enter a URL:
 **<input type="text" name="url" size="20" onblur="getAllHeaders(this.value)">**
<span class="message">::press tab when finished editing the field::</span></p>
<div id="msgDisplay"></div>
</form>
</body>
</html>
```

Figure 1-13 shows the page in the Safari browser.

Scoping the response



When the user types a URL in the text field and presses the TAB character or clicks outside the text field, this action triggers the text field's`onblur` event handler. The event handler is defined as a function named `getAllHeaders()`, which passes the URL the user has entered to the request object. The request object then sends a request to the URL and returns the available response headers to the Web page.

NOTE

The application already includes the main URL part as the text field value (as in "http://localhost:8080/") because the request object cannot send a request to a different host than the host that uploaded the web page to the user. In other words, the partially completed URL provides a hint to the user that the application can only send a request to that specified host.

Here is all the code included in the hack7.js file that the page imports. After showing this code, we w explain the parts that deal with displaying the server's response headers. Hack#1 explains how to initialize and open an HTTP connection with the request object, otherwise known as`XMLHttpRequest`. Hack#? explains trapping any errors with JavaScript's`try/catch/finally` statement.

```
var request;
var  urlFragment="http://10.0.1.3:8080/";

function getAllHeaders(url){
 httpRequest("GET",url,true);
}

//function for XMLHttpRequest onreadystatechange event handler
function handleResponse(){
 try{
 if(request.readyState == 4){
 if(request.status == 200){
 /*All headers received as a single string*/
 var headers = request.getAllResponseHeaders();
 var div = document.getElementById("msgDisplay");
 div.className="header";
 div.innerHTML="<pre>"+headers+"</pre>";
 } else {
 //request.status is 503 if the application isn't available;
 //500 if the application has a bug
 alert(request.status);
 alert("A problem occurred with communicating between "+
 "the XMLHttpRequest object and the server program.");
 }
 }//end outer if
 } catch (err) {
 alert("It does not appear that the server is "+
 "available for this application. Please"+
 " try again very soon. \nError: "+err.message);

 }
}

/* Initialize a Request object that is already constructed */
function initReq(reqType,url,bool){
 try{
 /* Specify the function that will handle the HTTP response */
 request.onreadystatechange=handleResponse;
 request.open(reqType,url,bool);
 request.send(null);
 } catch (errv) {
 alert(
 "The application cannot contact the server at the moment. "+
 "Please try again in a few seconds." );
 }
}
```

```
/* Wrapper function for constructing a Request object.
 Parameters:
 reqType: The HTTP request type such as GET or POST.
 url: The URL of the server program.
 asynch: Whether to send the request asynchronously or not. */
function httpRequest(reqType,url,asynch){
 //Mozilla-based browsers
 if(window.XMLHttpRequest){
 request = new XMLHttpRequest();
 initReq(reqType,url,asynch);
 } else if (window.ActiveXObject){
 request=new  ActiveXObject("Msxml2.XMLHTTP");
 if (! request){
 request=new ActiveXObject("Microsoft.XMLHTTP");
 }
 if(request){
 initReq(reqType,url,asynch);
 } else {
 alert(
 "Your browser does not permit the use of all "+
 "of this application's features!");}
 } else {
 alert(
 "Your browser does not permit the use of all "+
 "of this application's features!");}
}
```

The interesting stuff takes place in the `handleResponse()` function. This function calls the request object's `getAllResponseHeaders()` method, which returns (rather awkwardly) all the available response headers preformatted into a string.

> NOTE
>
> To get one header, you can also use request.getResponseHeader(). An example would be request.getResponseHeader("Content-Type");

A developer would probably prefer this value to be returned in JSON format as an associative array, rather than a monolithic string where extra code is required to pull out individual header information. The code then gets a hold of the `div` element where it will display the header values.

```
if(request.status == 200){
/*All headers received as a single string*/
var headers = request.getAllResponseHeaders();
var div = document.getElementById("msgDisplay");
div.className="header";
div.innerHTML="<pre>"+headers+"</pre>";
}
```

In order to provide a Cascading Style Sheets (CSS) style for the message display, the code then sets the `className` property of the `div` to a class that is already defined in a style sheet. Here's the style sheet, which is linked to the web page:

```
div.header{ border: thin solid black; padding: 10%;
 font-size: 0.9em; background-color: yellow}
span.message { font-size: 0.8em; }
```

In this manner, the code dynamically connects a `div` to a certain CSS class, which is defined by a separate style sheet. This strategy helps separate DOM programming from presentation decisions. Finally, the `div's innerHTML` property is set to the returned header values. We use the `pre` tag to conserve the existing formatting.

NOTE

You could alternatively manipulate the returned `string` and format the headers in a different way, using a custom function.

Figure 1-14 shows what the browser displays after the user submits a URL.

Separate the headers from the chaff

Generate A Styled Message With ACSS File

Let the users choose pre-designed styles for the messages they see.

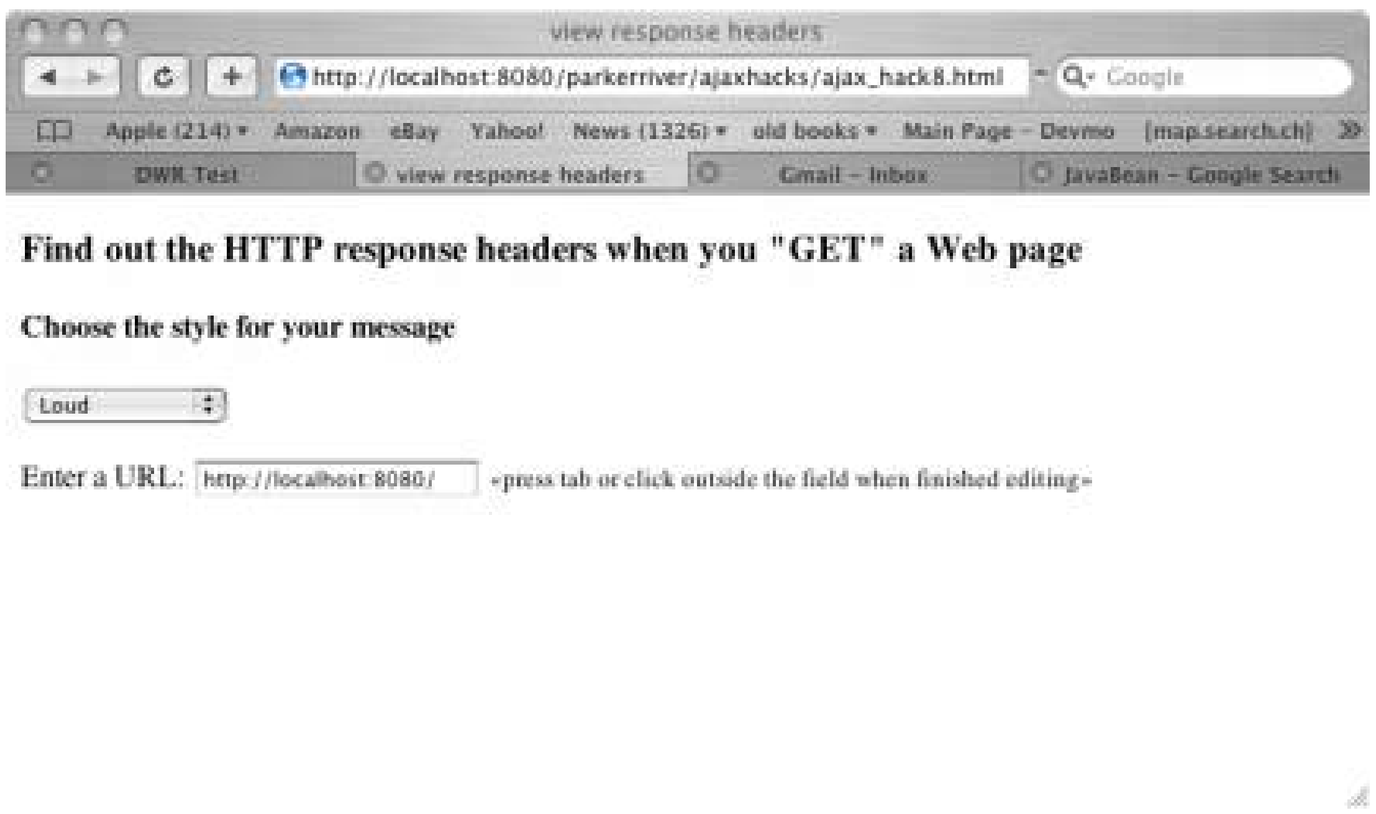
This hack lets the `request` object grab a text message. The user's choices and CSS styles provide the actual message appearance. Here's the HTML code for the page. All it includes are a `select` tag listing the styles the user can choose, and a text field containing a partial URL they can complete and submit to a server (this is an embellishment of the HTML page used for the previous hack). The information relates to response headers returned by the server (see Dig Into the HTTP Response). However, we are interested in this hack's dynamic message generation and style assignment.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
 "http://www.w3.org/TR/1999/REC-html401-19991224/strict.dtd">
<html>
<head>
 **<script  type="text/javascript"  src="js/hack8.js"></script**>
 <script type="text/javascript">
 function setSpan(){
 document.getElementById("instr").onmouseover=function(){
 this.style.backgroundColor='yellow';};
 document.getElementById("instr").onmouseout=function(){
 this.style.backgroundColor='white';};
 }
 </script>
 <meta  http-equiv="content-type"  content="text/html;  charset=utf-8">
 <title>view response headers</title>
 <link rel="stylesheet" type="text/css" href="/parkerriver/css/hacks.css">
</head>
<body   onload="document.forms[0].url.value=urlFragment;setSpan()">
<h3>Find out the HTTP response headers when you "GET" a Web page</h3>
<h4>Choose the style for your message</h4>
<form  action="javascript:void%200">
<p>
<select name="_style">
<option label="Loud" value="loud" selected>Loud</option>
<option label="Fancy" value="fancy">Fancy</option>
<option label="Cosmopolitan" value="cosmo">Cosmopolitan</option>
<option label="Plain" value="plain">Plain</option>
</select>
</p>
<p>**Enter a URL: <input type="text" name="url" size="20" onblur=**
**"getAllHeaders(this.value,this.form._style.value)">** <span id=
"instr" class="message">&#171;press tab or click outside the field
when  finished  editing&#187;</span>
</p>
<div id="msgDisplay"></div>
```

```
</form>
</body>
</html>
```

> NOTE
>
> The purpose of the `setSpan()` function defined within the web page's `script` tags is to give some instructions a yellow background when the user passes their cursor over the instructions ("press tab or click outside the field when finished editing").

Before we describe some of these code elements, you may be interested in how the web page appears in a browser. Figure 1-15 shows this window.



The CSS styles used by this web page derive from a style sheet file named hacks.css. When the user chooses a style (say "Cosmopolitan") with the `select` button, then finishes entering values in the text field, their chosen style is dynamically assigned to the container that will hold the message (a `div` element with `id msgDisplay`). Here is the hacks.css style sheet.

```
div.header{ border: thin solid black; padding: 10%;
 font-size: 0.9em; background-color: yellow; max-width: 80%}

span.message { font-size: 0.8em; }
div { max-width: 80% }

.plain { border: thin solid black; padding: 10%;
 font: Arial, serif font-size: 0.9em; background-color: yellow; }
```

```
.fancy { border: thin solid black; padding: 5%;
 font-family: Herculanum, Verdana, serif;
 font-size: 1.2em; text-shadow: 0.2em 0.2em grey; font-style: oblique;
 color: rgb(21,49,110); background-color: rgb(234,197,49)}
.loud { border: thin solid black; padding: 5%; font-family: Impact, serif;
 font-size: 1.4em; text-shadow: 0 0 2.0em black; color: black;
background-color: rgb(181,77,79)}
.cosmo { border: thin solid black; padding: 1%;
 font-family: Papyrus, serif;
 font-size: 0.9em; text-shadow: 0 0 0.5em black; color: aqua;
 background-color: teal}
```

The style sheet defines several classes (plain, fancy, loud, cosmo). A class in a CSS style sheet begins with a period character (as in `.fancy`) and defines various style properties, such as the font family and background color. Using this technique, your CSS experts can define the actual styles in one place. Clearly, an experienced designer would have some, ah, differences with my style-attribute choices here, but please bear with me!

The Ajax-related JavaScript code can assign the pre-defined styles to page elements based on user choices. Therefore, the presentation tier of your web application is separated from the application logic or domain tier.

The `onblur` event handler for the text field submits the URL value and the style name to a function named `getAllHeaders()`.

```
onblur="getAllHeaders(this.value,this.form._style.value)"
```

The reference this.form._style.value is JavaScript that represents the value of the option chosen from the `select` list (the style name like "fancy"). The reference this.value is the text entered by the user in the text field. Here is the JavaScript code in hacks8.js for the page, with the code highlighted that dynamically assigns the style to the displayed message.

```
var request;
var  urlFragment="http://10.0.1.3:8080/";
var st;

function getAllHeaders(url,styl){
 if(url){
 st=styl;
 httpRequest("GET",url,true);
 }
}

//event handler for XMLHttpRequest
**function handleResponse()**{
 try{
 if(request.readyState == 4){
 if(request.status == 200){
 /*All headers received as a single stringt*/
 var headers = request.getAllResponseHeaders();
```

```
var div = document.getElementById("msgDisplay");
div.className= st == "" ? "header" : st;
div.innerHTML="<pre>"+headers+"</pre>";
} else {
//request.status is 503 if the application isn't available;
//500 if the application has a bug
alert(request.status);
alert("A problem occurred with communicating between "+
"the XMLHttpRequest object and the server program.");
}
}//end outer if
} catch (err) {
alert("It does not appear that the server is available for "+
"this application. Please"+
" try again very soon. \nError: "+err.message);

 }
}

/* Initialize a Request object that is already constructed */
function initReq(reqType,url,bool){
 try{
 /* Specify the function that will handle the HTTP response */
 request.onreadystatechange=handleResponse;
 request.open(reqType,url,bool);
 request.send(null);
 } catch (errv) {

 alert(
 "The application cannot contact the server at the moment. "+
 "Please try again in a few seconds.");
 }
}

/* Wrapper function for constructing a Request object.
 Parameters:
 reqType: The HTTP request type such as GET or POST.
 url: The URL of the server program.
 asynch: Whether to send the request asynchronously or not. */
function httpRequest(reqType,url,asynch){
 //Mozilla-based browsers
 if(window.XMLHttpRequest){
 request = new XMLHttpRequest();
 initReq(reqType,url,asynch);
 } else if (window.ActiveXObject){
 request=new  ActiveXObject("Msxml2.XMLHTTP");
 if (! request){
 request=new ActiveXObject("Microsoft.XMLHTTP");
 }
 if(request){
 initReq(reqType,url,asynch);
 } else {
```

```
 alert(
 "Your browser does not permit the use of all "+
 "of this application's features!");}
 } else {
 alert(
 "Your browser does not permit the use of all "+
 "of this application's features!");}
}
```

Easy as Pie

The `getAllHeaders()` function sets a top-level `st` variable to the name of a CSS style class (plain, fancy, loud, or cosmo). The code then sets the `className` property of the `div` that holds the message in a shockingly simple way, which changes the style assigned to the message.

```
if(request.status == 200){
 /*All headers received as a single stringt*/
 var headers = request.getAllResponseHeaders();
 var div = document.getElementById("msgDisplay");
 div.className= st == "" ? "header" : st;
 div.innerHTML="<pre>"+headers+"</pre>";
}
```

If for some reason the choice of class name derived from the web client is the empty `string` (it cannot be here because the `select` tag only contains complete `string` values), then the `div` element is assigned a default style class name of `header`.

> NOTE
>
> This JavaScript could potentially be imported into another client web page, so you have to include some checks for invalid input values.

The hacks.css style sheet also defines the `header` class.

Here are two examples of the same message assigned different styles by the user. First, figure 1-16 reproduces the selection of a "cosmo" style.

A cosmo styled message

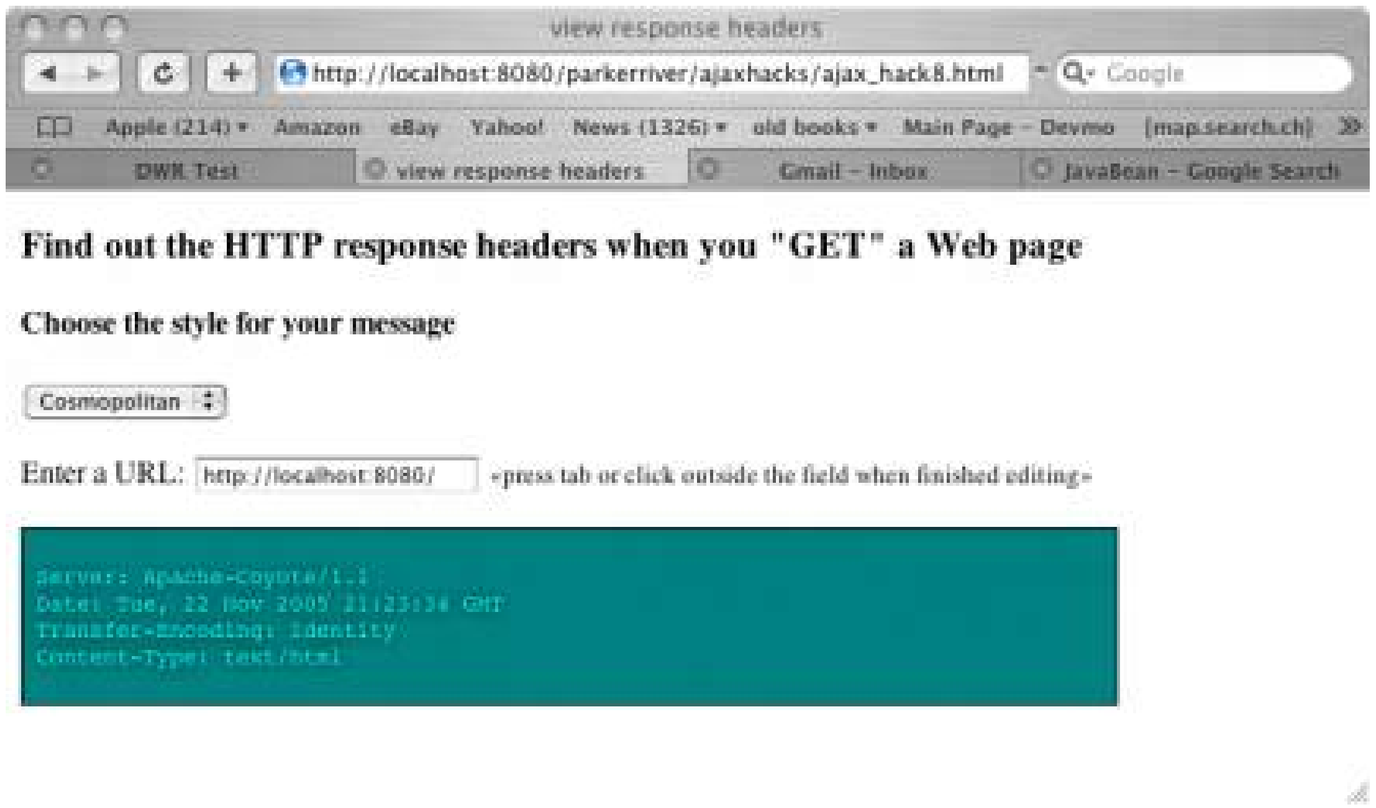


Figure 1-17 depicts an alternate style.

Alas, a plain style

Generate A Styled User Message On The Fly

Dynamically define and assign CSS styles to web page content.

JavaScript and Document Object Model (DOM) programming allow you to define Cascading Style Sheet (CSS) style attributes and apply them to page elements from scratch. An example of where you may want to implement these methods is a Wiki page that permits users to develop their own page designs and styles.

> NOTE
>
> In most cases, separating the style definitions from the JavaScript code is the way to go. Separating application concerns or tiers in this manner allows each of these elements to evolve independently, and makes web development less complex and more efficient.

This hack, like the one before it, dynamically displays server information based on the user's choice of style categories. Unlike the previous hack, this one formulates the styles in code, then applies the chosen style to an HTML element. Here is the code, with the style information highlighted.

```
var request;
var  urlFragment="http://localhost:8080/";
var st;

function getAllHeaders(url,styl){
 if(url){
 st=styl;
 httpRequest("GET",url,true);
 }
}

 /* Set one or more CSS style attributes on an DOM Element
 CSS2Properties Object.
 Parameters:
 stType stands for a style name, as in 'plain,''fancy,''loud,' or 'cosmo'.
 stylObj is the HTML Element's style property, as in div.style. */

function setStyle(stType,stylObj){
 switch(stType){
 case 'plain' :
 stylObj.maxWidth="80%";
 stylObj.border="thin solid black";
 stylObj.padding="5%"
 stylObj.textShadow="none";
 stylObj.fontFamily="Arial, serif";
```

```
stylObj.fontSize="0.9em";
stylObj.backgroundColor="yellow"; break;
case 'loud' :
stylObj.maxWidth="80%";
stylObj.border="thin solid black";
stylObj.padding="5%"
stylObj.fontFamily="Impact, serif";
stylObj.fontSize="1.4em";
stylObj.textShadow="0 0 2.0em black";
stylObj.backgroundColor="rgb(181,77,79)"; break;
case 'fancy' :
stylObj.maxWidth="80%";
stylObj.border="thin solid black";
stylObj.padding="5%"
stylObj.fontFamily="Herculanum, Verdana, serif";
stylObj.fontSize="1.2em";
stylObj.fontStyle="oblique";
stylObj.textShadow="0.2em 0.2em grey";
stylObj.color="rgb(21,49,110)";
stylObj.backgroundColor="rgb(234,197,49)"; break;
case 'cosmo' :
stylObj.maxWidth="80%";
stylObj.border="thin solid black";
stylObj.padding="1%"
stylObj.fontFamily="Papyrus, serif";
stylObj.fontSize="0.9em";
stylObj.textShadow="0 0 0.5em black";
stylObj.color="aqua";
stylObj.backgroundColor="teal"; break;
default :
alert('default');

 }
}

//event handler for XMLHttpRequest
function handleResponse(){
 try{
 if(request.readyState == 4){
 if(request.status == 200){
/*All headers received as a single stringt*/
var headers = request.getAllResponseHeaders();
var div = document.getElementById("msgDisplay");
if(st){
setStyle(st,div.style);
} else {
setStyle("plain",div.style);
}
div.innerHTML="<pre>"+headers+"</pre>";
} else {
//request.status is 503 if the application isn't available;
//500 if the application has a bug
```

```
 alert(request.status);
 alert("A problem occurred with communicating between "+
 "the XMLHttpRequest object and the server program.");
 }
 }//end outer if
 } catch (err) {
 alert("It does not appear that the server is available for "
 "this application. Please"+
 " try again very soon. \nError: "+err.message);

 }
}

/* Initialize a Request object that is already constructed */
function initReq(reqType,url,bool){
 try{
 /* Specify the function that will handle the HTTP response */
 request.onreadystatechange=handleResponse;
 request.open(reqType,url,bool);
 request.send(null);
 } catch (errv) {

 alert(
 "The application cannot contact the server at the moment. "+
 "Please try again in a few seconds." );
 }
}

/* Wrapper function for constructing a Request object.
 Parameters:
 reqType: The HTTP request type such as GET or POST.
 url: The URL of the server program.
 asynch: Whether to send the request asynchronously or not. */
function httpRequest(reqType,url,asynch){
 //Mozilla-based browsers
 if(window.XMLHttpRequest){
 request = new XMLHttpRequest();
 initReq(reqType,url,asynch);
 } else if (window.ActiveXObject){
 request=new  ActiveXObject("Msxml2.XMLHTTP");
 if (! request){
 request=new ActiveXObject("Microsoft.XMLHTTP");
 }
 if(request){
 initReq(reqType,url,asynch);
 } else {
 alert(
 "Your browser does not permit the use of all "+
 "of this application's features!");}
 } else {
 alert(
 "Your browser does not permit the use of all "+
```

```
 "of this application's features!");}
}
```

Nudging Aside the Style sheet

Each HTML element on a web page has a `style` property, if its host browser support CSS style sheets. For example, a `div` element has a property `div.style` that allows a JavaScript writer to set inline style attributes for that `div` (as in `div.style.fontFamily="Arial"`). This is how the `setStyle()` function works in the prior code. The two function parameters are a style name like "fancy" (which we made up) and the `style` property of a specific `div` element. The function then sets the appearance of the HTML `div` element on the web page.

The information that appears on the page (a bunch of response headers) is derived from the server using the request object. As in the previous hack, the user enters a URL, then clicks outside the text field or presses the `TAB` key, thus firing an `onblur` event handler that sets the request object and CSS styling in motion. Here is the HTML for the page. It is not much different than Generate a Styled Message With a CSS File, but omits the link to a style sheet. All the styling for this hack is defined by the imported JavaScript file.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
 "http://www.w3.org/TR/1999/REC-html401-19991224/strict.dtd">
<html>
<head>
 <script type="text/javascript" src="/parkerriver/js/hack10.js"></script>
 <script type="text/javascript">
 function setSpan(){
 document.getElementById("instr").onmouseover=function(){
 this.style.backgroundColor='yellow';};
 document.getElementById("instr").onmouseout=function(){
 this.style.backgroundColor='white';};
 }
 </script>
 <meta http-equiv="content-type" content="text/html; charset=utf-8">
 <title>view response headers</title>
</head>
<body  onLoad="document.forms[0].url.value=urlFragment;setSpan()">
<h3>Find out the HTTP response headers when you "GET" a Web page</h3>
<h4>Choose the style for your message</h4>
<form  action="javascript:void%200">
<p>
<select name="_style">
<option label="Loud" value="loud" selected>Loud</option>
<option label="Fancy" value="fancy">Fancy</option>
<option label="Cosmopolitan" value="cosmo">Cosmopolitan</option>
<option label="Plain" value="plain">Plain</option>
</select>
</p>
<p>Enter a URL: <input type=
"text" name="url" size="20" onblur=
"getAllHeaders(this.value,this.form._style.value)">
```

```
<span id=
"instr" class="message">
&#171;press tab or click outside the field when finished editing&#187;
</span></p>
<div id="msgDisplay"></div>
</form>
</body>
</html>
```

The `getAllHeaders()` function, an event handler for `onblur`, passes into the application the name of the style the user has chosen from a select list (such as "cosmo") as well as the URL of the server component. The only purpose of the server component is to provide a value for display. We're mainly interested in dynamically generating styles for any type of server information your applications could acquire via Ajax and the request object.

NOTE

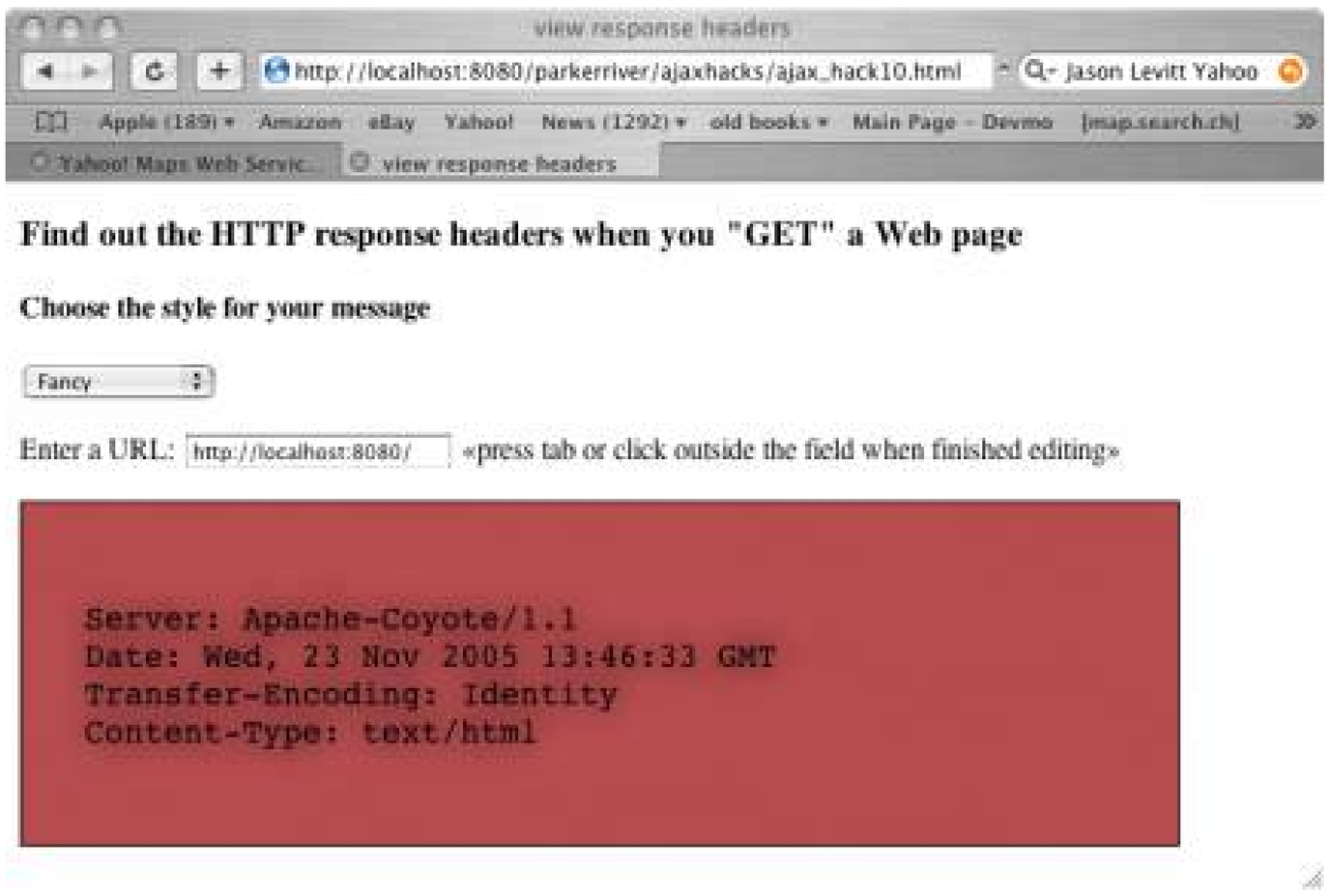
The purpose of the `setSpan()` function defined within the web page's `script` tags is to give some instructions a yellow background when the user passes their cursor over the instructions ("press tab or click outside the field when finished editing").

Figure 1-18 what the page looks like in the web browser prior to the sending of the HTTP request.

Choose a style for dynamic generation

Then when the user optionally selects a style name, fills out the URL address in the text field, and presses TAB, figure 1-19 depicts what the browser looks like.

View styled server data



None of these web-page changes involves waiting for the server to deliver a new page. The request object fetches the data from the server in the background, and the client-side JavaScript styles the displayed information. Voila, Ajax!

[1] The object can make an asynchronous request to a server, meaning that once the request has been initiated, the rest of the JavaScript code does not have to wait for a response to execute. `XMLHttpRequest` can also make synchronous requests.

[2] The Mozilla Firefox `XMLHttpRequest` object has `onload`, `onprogress`, and `onerror` properties that are Event Listener types. Firefox has also defined `addEventListener()`, `dispatchEvent()`, `overrideMimeType()`, and `removeEventListener()` methods. See http://www.xulplanet.com/references/objref/XMLHttpRequest.html for more details on these Firefox request object members.

Chapter 2. Validation

Validation

Previous

E B V N
We are Vietnames

Validate A Texfield Or Textarea For Blank Fields

A web developer does not want their Ajax application to hit the network with a request if the user leaves a necessary text field blank. This includes`input` tags of the type `text`, and the large boxes called `textarea` tags in HTML. This is one of the most common forms of validation. With a web application, you can't get something for nothing!

This hack shows the code for checking if a text field or`textarea` is blank. The inline way of doing it is by assigning a check for the field's value in the text field's event handler.

```
<input type="text" name="firstname" id="tfield" onblur=
"if (this.value) {doSomething();}" />
```

Or, in a textarea...

```
<textarea name="tarea" rows="20" id="question" cols="20" onblur=
"if (this.value) {doSomething();}">
```

The JavaScript phrase `if (this.value) { }` returns `false` if the user leaves a field blank, so the function call `doSomething()` will never occur. JavaScript evaluates a blank web-form text field as the empty `string` or `&&`, which evaluates to `false` when it's used in the context of a programming test. The `this` keyword is a nice generic way of referring to the form field that contains the event handler attribute such as `onblur`. `this.value` returns the text field's value, which in our case is the empty `string`. Easy huh?

> NOTE
>
> `onblur` captures the event involving the transfer of the keyboard focus away from a form field. For example, the user triggers an`onblur` event handler when they type in a text field then click on another form field or type the`TAB` character. If you used the `onchange` event handler, the browser will only call the`onchange`-related function if the field's value changes. In other words, the `change` event would not capture the instances where the user leaves the text field blank.

Separating Logic from View

Probably a better way of going about your event handling tasks is separating the logic of your code from the HTML or template text that comprises the application's visual aspects. The JavaScript belongs in an external file that the HTML page imports with a`script` tag. Inside the external file, the code binds a field's various event handlers to a function, or the code that represents your

application's behavior.

Say we have a web page myapp.html, which includes the following HTML in its header:

```
<html  xmlns="http://www.w3.org/1999/xhtml"  xml:lang="en"  lang="en">
<head>
      <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <script type="text/javascript" src="js/hacks_method.js" ></script>
    <title>Cool Ajax application</title>
</head>
```

The file hacks_method.js is located in a directory js, which is in the same directory as the HTML file. The HTML file contains the same `textarea` and text field as mentioned in this hack, except these fields no longer have an `onblur` attribute. The JavaScript file includes this code.

```
window.onload=function(){
 var txtA = document.getElementById("tarea");
    if(txtA != null){
        txtA.onblur=function(){
            if (this.value) { doSomething();}};  }
 var tfd = document.getElementById("tfield");
 /* An alternative:
 if(tfd != null && !txtA!
= null){tfd.onblur = txtA.onblur; }
 */
    if(tfd != null){
        tfd.onblur=function(){
            if (this.value) { doSomething();}};  }
}
```

`window.onload` involves the binding of the `load` event to your blank-field checks. `load` occurs when the browser has completed loading the web page, so when that happens, all the stuff after `window.onload=` happens.

The `getElementById()` method returns a reference to an HTML element, such as the `textarea` reference stored in `txtA`. Then the code binds the `textarea's onblur` event handler to a function, which checks for blank field values before it calls `doSomething()`. The code initiates the same behavior for the text field referred by the variable `tfd`.

> NOTE
>
> If the web designers leave out the text fields with the `id tarea` or `tfield` then nothing happens, because the `getElementById()` method would return `null` and the code includes a check for that occurrence.

Another way of binding an event handler to a function is to declare the function somewhere and then use the function name.

```
window.onload=function(){
```

```
 var txtA = document.getElementById("tarea");
 txtA.onblur=doSomething;//no parens...
}
function doSomething(){ //... }
```

When the code binds an event handler to a previously defined function, leave the parentheses off the function name.

Programmers often consider the definition of the blank-field checks and other coding stuff in an external file a better way of organizing any but the most trivial applications. In addition, the XHTML document or web-page definition discourages the mixing up of JavaScript code logic with the structure of the page, as evidenced by the use of event-handler tag attributes such as`onblur`.

Validate Email Syntax

Check email syntax on the client-side before the server component takes over.

Many Web sites ask their users to register a user name as an email address. This hack makes sure the s valid, before the server component finds out whether the email address has already been used as a use Names takes care of the second step of this task.

The Longest Wait

When registering with a Web site, users typically type in an email address, make up a password, click Sul long wait staring at the browser as the page is slowly reconstructed (if they're lucky). To add insult to in addresses are supposed to be unique, people often register at a site more than once (guilty as charged! visited, they try to register with the same email address. Therefore, the application often has to both ch whether the name is already being used.

AJAX can validate the email on the client-side and initiate a trip to the server behind the scenes to find o in use, without disrupting the current view of the page. Validate Unique User Names ensures the unique these hacks share the same code base, a mix of JavaScript and other Ajax techniques.

Checking Out the Email Syntax

Web sites use email addresses because they are guaranteed to be unique, as long as they are valid. In a the email to communicate with you later. You do not have to initiate a server-roundtrip just to validate a task can be initiated in the client, which cancels the submission of the user name to the server if the em

What criteria can we use for validation? A fairly dry technical document, "RFC 2822" is a commonly acce oganizations can use as a basis for validating emails. Let's look at an example email address to briefly s hackreader@oreilly.com. The `hackreader` is called the `local part` of the address, which typically identif the `commercial at` sign (@), which precedes the `Internet domain` , those often well-known addresses handle in-transit emails. Google.com and yahoo.com come to mind.

All of this is common knowledge. You may not know that RFC 2822 specifies that the `local part` cannot quoted, which is rare, as in "bruce perry"@gmail.com). The `local part` also cannot contain various spec ( ) < > , " @ : ; \ [ ]. Maybe if someone tries to create an email address that looks like <(([[))>@ yours outright rather than give them points for originality!

The `local part` can and often does contain period characters, as in bruce.perry@google.com. But the p followed by alphanumeric characters (you cannot use an email such as bruce.@google.com). Your organ user name and domain be more than two characters (you don't see too many domains like zz.org). The one period, as in bruce@lists.myorg.net. But the domain cannot begin or end with a period (bruce@.lists guidelines permit but discourage a `domain literal` that you almost never encounter: as in bruce@[192 we can check for in our in our validation code, plus some more special characters that our organization addresses provided by our users.

Looking at the Code

First, take a look at the page that imports the JavaScript code:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/2000/REC-xhtml1-20000126/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<script type="text/javascript" src="js/http_request.js"></script>
<script type="text/javascript" src="js/email.js"></script>
<meta http-equiv="content-type" content="text/html; charset=utf-8">
<title>Enter email</title>
</head>
<body>
<form action="javascript:void%200">
<div id="message"></div>
Enter email: <literal><b>&lt;input type=&quot;text&quot; name=&quot;email&quot; size=
<button type="submit" name="submit" value="Send">Send</button>
</form>
</body>
</html>
```

Idtxt A shows a simple web page with a textfield for entering an email and aSend button.

Write down your email, please

The user types their email into the text field then clicks theSend button. This action does not send the e

First, the code has to validate the email's syntax. The HTML code imports two JavaScript files with thesc for our thorough email-syntax check. http_request.js sends the email as a user name to a server compc Ajax to Hack #25.

If the user types in an invalid email address, figure 3-2 shows what the browser window looks like. The message summarizing what appeared to be wrong with the entered email address. If the email address sends it to a server component to determine if the address has already been used as a user name. Here

```
var user,domain, regex, _match;

window.onload=function(){
    document.forms[0].onsubmit=function() {
        checkAddress(this.email.value);
        return false;
    };
};
/* Define an Email constructor*/
function Email(e){
    this.emailAddr=e;
    this.message="";
    this.valid=false;
}
/* containsSpecials is a class method of the Email object */
function containsSpecials(str){
    /* RFC 2822 generally does not allow these characters to appear in the
 username email segment ( ( ) < > , " @ : ; \ [ ]  ), plus ASCII 0-31 and
 the space character (unless the username is quoted,
 which you rarely see in typical usage, as in "joe"@hotmail.com )
 We've optionally added some other special characters to forbid from user names,
 such as !?#$%&='^|{}* */
    aardvark*var re = /([()<>\,\"\@\:\;\\[\]\|\!\?\#\$\%XXXredpre#7XXX\*
```

First, the code sets up the handling for the user's click on theSend button. `window.onload` specifies an e the browser completes the loading of the web page.

NOTE

Event handlers are designed to assign functions or blocks of code that specify the application's beha this happens in the browser." For example, when the user submits the form, call this function first (handler).

The reason the code uses `window.onload` is that before the code can control `form` -related behavior, the from JavaScript, that is, fully loaded into the browser.

```
window.onload=function(){
    document.forms[0].onsubmit=function() {
        checkAddress(this.email.value);
        return false;
    };
```

```
};
```

In turn, the code sets up the form element's `onsubmit` event handler, a function that calls `checkAddress` handler intercepts the form submission, because we want to validate what the user entered into the tex anything else. `checkAddress()` takes as a parameter the address that the user typed, if they typed any

Check Email at the Door

Now we will look at the `checkAddress()` function.

```
function checkAddress(val){
    var eml = new Email(val);
    var url;
    eml.validate();
    if (! eml.valid) {eMsg(eml.message,"red")};
    if(eml.valid)
    {
        url="http://www.parkerriver.com/s/checker?email="+
            encodeURIComponent(val);
        httpRequest("GET",url,true,handleResponse);
    }
}
```

This function creates a new `Email object` , validates the user's email, and if the email is valid, submits first element of this description you might be curious about is `Email object` . What the heck is that? An template that we can use over and over again every time we want to check the syntax of an email. In fa that handled emails, we would break this code off into it's own file (like emailObject.js) so that it would of lines of additional complex code in future applications. Here is the `Email object` definition.

```
/* Define an Email constructor*/
function Email(e){
    this.emailAddr=e;
    this.message="";
    this.valid=false;
}
```

Simple, huh? An `Email object` is constructed using a JavaScript function definition that takes the email parameter, stored here as e.

> NOTE
>
> This is a special kind of function that is called a constructor in object-oriented parlance, because it i object.

An `Email object` has three properties: an email address (`emailAddr`); a message, and a `boolean` or tr When you use the `new` keyword in JavaScript to create a new `Email object` , the `emailAddr` property is address (remember? In `e`). The message is initialized to the empty `string` because new `Email objects`

messages associated with them. The validity of the email, somewhat pessimistically, is innitialized as `fal`

```
var email = new Email("brucew@yahoo.com");
```

The `this` keyword refers to the instance of `Email` that the browser creates in memory when the code ge
For example, a bicycle company might create a mold for new bicycle helmets. Conceptually, the mold is
the company makes new helmets, these helmets are instances of the mold or tempate that was develop

On to Validation

An `Email object` validates its email, which in our application takes place when the user clicks the `Send` b
function had code like `eml.validate()` and `if(eml.valid)` indicating that our application validates indi
`valid` properties. This happens because the code defines a `validate()` function then signals that the `Em`
with that function.

NOTE

Using code such as `Email.prototype.validate=validate;` is a special way in JavaScript to specif
function `validate()` and every new `Email object` has its own `validate()` method. Using object-o
mandated, but it makes the code a little cleaner, more concise, readable, and reusable.

Now we will describe the validation code, which contains a few regular expressions for checking email sy
prior code sample for email.js, is fairly complex, but the embedded comments are designed to help you
the code accomplishes. In order, here are the rules for our validation logic, partly based on RFC 2822 ar
criteria for proper email syntax:

1. If the email is the empty `string` , or if the `emailAddr` property value is `null` , or the email address
   or any periods at all, it is rejected. No surprises there.
2. The code then uses a regular expression to grab the `local part` of the email. This is usually the us
   characters preceding the `.` This encompasses a character that isn't a period or space character, foll
   that include zero or one periods, and ending at the `@` point with a character that is not a period.
3. The code then checks whether the local part contains any special characters by using the `contains`
   `local part` does contain any of these characters then the user receives an error message.
4. The code then grabs all of the characters after the `@ and checks whether the character` string`@`
   literal (however rare that would be), or a typical domain syntax. The rule for the latter syntax is ex
   <redpre#215></nodocbook> character followed by at least three characters that are not a period
   followed by zero or one instances of two or more characters ending with a period, followed by thre
   periods."
5. This portion of the email address, the `domain` , is also checked for the forbidden special characters
   method.

Now we will take a look at the code for the `containsSpecials()` method. This method is designed to ch
parameter contains any various special characters (like the `@ sign for the reason that it shouldn'`
`in its psition between the` local part `and the` domain@) and punctuation marks.

```
function containsSpecials(str){
    /* RFC 2822 generally does not allow these characters to appear in the
 username email segment ( ( ) < > , " @ : ; \ [ ]  ), plus ASCII 0-31 and
 the space character (unless the username is quoted,
```

```
which you rarely see in typical usage, as in "joe"@hotmail.com )
We've optionally added some other special characters to forbid from user names,
such as !?#$%&='^|{}* */

    var re = /([()<>\,\"\@\:\;\\[\]\|\!\?\#\$\%XXXredpre#12XXX\*
```

The function's comments describe most of the gist of this code. You simply pass a`string` to the function characters using a regular expression. JavaScript's built-in`RegExp` object's `exec` method returns an `arra` otherwise. The `if(match){ return [true,RegExp.$1];}` code returns an `array` whose elements are a matching character in the `string` . We'll show how that character is used in the user message in a mom does not find any of the characters, then it returns an`array` containing the boolean `false` value.

The final piece of code here, `Email.containsSpecials=containsSpecials` , makes the `containsSpeci` `method` of the `Email` object. You can call the function in the manner of`var sp = Email.containsSpeci` will then refer to the returned `array` .

User Gets the Third Degree

If the user includes illegal characters in their email, or otherwise types in an invalid address or leaves th greeted with a message like figure 3-2.

Shame on you

For example, the following code inside of`validate()` creates one of these messages if the email addres in the domain or part after the `@` .

```
//check for special characters in the domain
sp = Email.containsSpecials(domain);
//The first array member in the return value will be true or false
//so if sp[0] evals to true then the regular expression matched a
//special character, which is stored inthe second array member,sp[1]
if(sp[0]){
```

```
    this.message="The domain version containing the top-level domain suffix "+
        "(e.g., .net) cannot contain special characters that RFC 2882"+
       " or our rules forbid such as \""+sp[1]+"\" .";
     this.valid=false;
     return;
}
```

Notice that the code also sets the `Email object's valid` property to `false` . Then `checkAddress()` che the email address heads off to the server in the next hack.

```
//inside checkAddress()...
eml.validate();
if (! eml.valid) {eMsg(eml.message,"red")};
if(eml.valid)
{
    url="http://www.parkerriver.com/s/checker?email="+
        encodeURIComponent(val);
    httpRequest("GET",url,true,handleResponse);
}
```

The `eMsg()` function generates the message. True to Ajax, `eMsg()` uses a little DOM, a little dynamic Cas programming, and JavaScript.

```
function eMsg(msg,sColor){
    var div = document.getElementById("message");
    div.style.color=sColor;
     div.style.fontSize="0.9em";
    //remove old messages
    if(div.hasChildNodes()){
        div.removeChild(div.firstChild);
    }
    div.appendChild(document.createTextNode(msg));

}
```

The parameters to this function are the text message and the color of the text. The application uses red user notifications about their user name. This is discussed in Validate Unique User Names. The code dyn inside a `div` that the HTML reserves for that purpose.

```
var div = document.getElementById("message");
```

On Deck

While the user is typing and clicking around this page, trying out new email versions, the page itself doe shows different content. It seems as if a server component never participates (although a server role do email address is valid), as the application's responsiveness speeds along.

However, we have not gone into very much detail about what's happening on the server end. The serve unique user names for its web application. Once this hack gives the green light on email syntax, then th the server, which checks to see if the application already has a stored version. The next hack dives into

?

Previous

Validate Unique User Names

Ensure that an email address used as a user name is unique, but do not submit anything else on the page.

The purpose of all this email-address validation is so you can safely send the email off to the server-side program, where it will be checked against an existing database to see if it has already been used. This hack does exactly that.

Figure 3-2 shows what the web page looks like when the user types an entry that breaks our validity check. The user is greeted with a browser page such as figure 3-3 if the program deems the email syntax okay.

Unique name passes muster

The message will convey to the user that their chosen name has already been taken, or that they have typed a unique email address for this application. But all email addresses are unique, you might declare. That's true, but web users often register more than once at web applications, because whoever remembers the tedious details about registering at the countless web sites we typically use? You might enter the same email address when you are registering a new user name, and the application responds that the email is already taken.

A non-Ajax web application will submit all the form values at once when a user registers, and often painstakingly reconstruct the page, if only to notify the user that they have to try again. This hack only submits the email address and does not touch or refresh other page elements.

Here's How it Works

Here is the HTML code, which the pevious hack also uses.

```
XXXredpre#17XXX
```

The JavaScript in email.js sends the email address to the server, which checks an existing database of user names and responds with a "1" if the address is already used. The simple XML response output looks like XXXredpre#177XXX. The code uses XXXredpre#178XXX to send the validated emai address to the server component. See Hack #24 for the details on email validation. As the next step, this hack focuses on the data exchange with the server component, to find out if someone else is already using the user name. Here is the code from the XXXredpre#179XXX function that sends the email.

```
XXXredpre#18XXX
```

The XXXredpre#180XXX function wraps the creation and initialization of the request object. http_request.js contains this code. XXXredpre#181XXX takes as parameters:

- The type of request as in GET or POST
- The URL or server web address
- A boolean indicating whether the request is asynchronous or not.
- The name of a function or a function literal that handles the server response.

Server Handshake

The server then returns some XML indicating whether it has found the user name or not. Here is the code for XXXredpre#182XXX, which appears in email.js.

```
XXXredpre#19XXX
```

XXXredpre#183XXX gets the XML by accessing the XXXredpre#184XXX property of XXXredpre#185XXX. The code calls the DOM Document method XXXredpre#186XXX, which returns a XXXredpre#187XXX, just like an XXXredpre#188XXX, of nodes that have the specified tag name. XXXredpre#189XXX is the tag name, as in XXXredpre#190XXX. Since the return value is an XXXredpre#191XXX structure, the code gets the first and only array member using XXXredpre#192XXX:

```
XXXredpre#20XXX
```

The code then accesses the text contained by the XXXredpre#193XXX tag and generates a user message. Hack #23 shows the XXXredpre#194XXX code.

For Those Server Hackers...

Here is the code for the server-side component, which is a Java XXXredpre#195XXX that mimics a database. It uses a XXXredpre#196XXX type, a kind of XXXredpre#197XXX object, to contain the stored user names; however, a full-fledged production application would use middleware to connect with a database and check on user names.

```
XXXredpre#21XXX
```

In this case, the server component does not have to check the validity of the email, because the client-side JavaScript has already taken care of that job. The server role is just to check the database of user names, then add the new name if it doesn't already exist.

Validate Credit-card Numbers With AJAX

Validate credit card numbers without submitting and refreshing the entire web page.

Entering a credit card number on a web page has become common place. This hack verifies the entered credit card number, then only submits it to the server component if the number is valid. Nothing else changes on the page except for a user message, which notifies the user of any error conditions or that their credit card has passed muster and has been sent to the server to be processed. The server connection would likely be initiated over Secure Sockets Layer (SSL), such as with the HTTPS protocol, and is involved with an e=commerce component that further verifies the purchase information with a merchant bank. This hack, however, just verifies the number, generates a message, and makes an HTTP request using Ajax techniques.

Figure 3-4 shows what the web page looks like.

Enter credit-card number for verification

This is the web page code. It imports two JavaScript files.

XXXredpre#22XXX

The user chooses a credit card type (e.g., "Mastercard"), enters the card number, expiration date, and Card Security Code, and clicks the XXXredpre#198XXX button. However, instead of having the

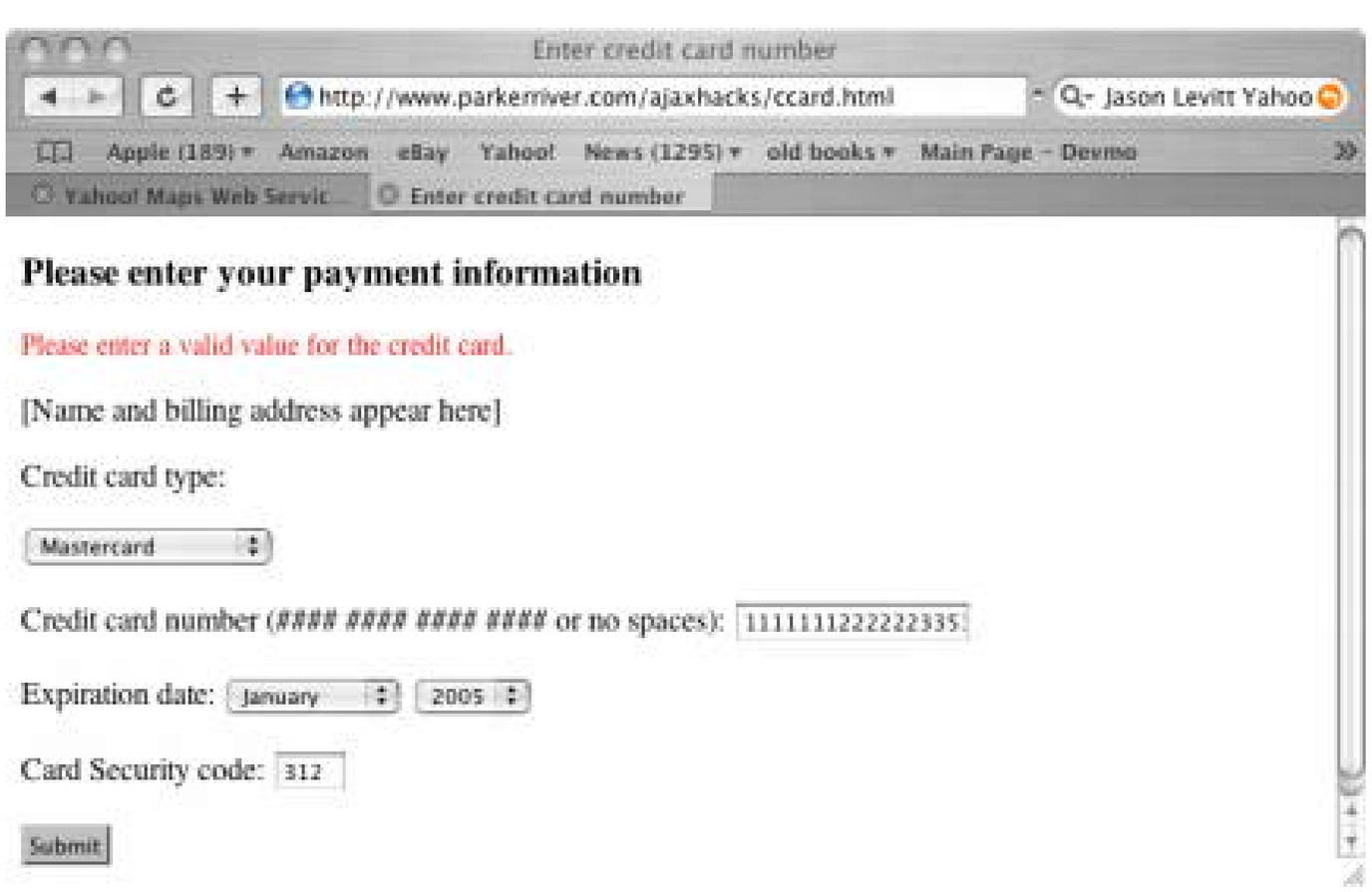
page dissolve and the values depart immediately for the server, the application verifies a few conditions first. The JavaScript makes sure that the fields are not blank or contain a minimum number of characters (such as three for the Card Security Code), then it verifies the card number using the Luhn formula or algorithm.

NOTE

This is a well-known algorithm used to verify ID numbers like credit card numbers. See http://en.wikipedia.org/wiki/Luhn_formula.

If one of these checks fails, the hack displays a message in red. Figure 3-5 shows one of these messages.

Time to re-enter the credit card



If the credit card number is verified and everything else has been correctly entered, then the hack uses XXXredpre#199XXX to send this information to a server.

We are not strictly making a secure connection in this hack, but a real application would not send any purchase information unencrypted over a network.

A message in blue notifies the user, as in figure 3-6.

Purchase information passes muster

Verifying the Card Number

cc.js contains the code for responding to the user's button click, as well as for verifying the information and generating a user message. http_request.js creates and calls the methods of XXXredpre#200XXX. See Hack #3. Here is the code for cc.js.

XXXredpre#23XXX

There is a lot of functionality to absorb here, so first we will discuss the button click. When the browser completes loading the web page, this event is captured by the code XXXredpre#201XXX. This event handler is a sensible place to set up other event handlers, because the code is assured that the browser has finished loading other HTML tags that might be used by these handlers. Then the code sets up an event handler for when the user submits the form.

XXXredpre#24XXX

The form's XXXredpre#202XXX event handler points to a function that calls XXXredpre#203XXX, then retuns XXXredpre#204XXX, which effectively cancels the browser's form submission. We are using the request object to send the form values, only after verifying that the submissions are valid. Let's look at the XXXredpre#205XXX function.

XXXredpre#25XXX

This function includes a number of common-sense checks before it validates the credit-card number using another function, XXXredpre#206XXX. If the latter function returns XXXredpre#207XXX, then the code builds a URL for the server component, then uses XXXredpre#208XXX to send the card information.

The XXXredpre#209XXX function is responsible for setting up XXXredpre#210XXX and connecting with the server. The function takes four parameters: The type of request (as in XXXredpre#211XXX or XXXredpre#212XXX), the URL, whether or not the request shuld be asynchronous or not, and the name of the function that will handle the response.

> NOTE
>
> This function name should be passed in without the following parentheses. It can also be a function literal, as in XXXredpre#213XXX.

This code appears in the file http_request.js. See Hack #3.

Shooting the Luhn

The XXXredpre#214XXX function verifies that the credit-card number is at least 13 characters and does not contain any letters. If the credit card number passes these checks, then the code removes any spaces or dashes from the string and calls a function that uses the Luhn formula.

```
XXXredpre#26XXX
```

Here is the code for the XXXredpre#215XXX function.

```
XXXredpre#27XXX
```

Information on the Luhn formula or algorithm is easily found on the web, so we will not take up a lot of space describing it here.

> NOTE
>
> See Wikipedia's explanation of the Luhn algorithm: http://en.wikipedia.org/wiki/Luhn_formula.

This function takes a XXXredpre#216XXX of numbers, applies the formula to the numbers, and returns the sum to XXXredpre#217XXX. If the total can be evenly divided by 10, then the credit card number is valid. Here is the piece of code from XXXredpre#218XXX that makes this determination.

```
XXXredpre#28XXX
```

The server component returns a bit of XML indicating success or failure, mimicking the processing of purchase order, as in XXXredpre#219XXX. The XXXredpre#220XXX function generates a user message from this return value.

```
XXXredpre#29XXX
```

The XXXredpre#221XXX function is responsible for generating a styled user message, in red in the event of an error in handling the purchase information, in blue otherwise. Figure 3-6 shows a user message after the credit card has been verified and handled by the server. However, the entire process takes place back stage, the web page never refreshes, and only small parts of the user interface change as the user interacts with the application.

Validate Credit-card Security Codes

Make sure the security code is entered correctly in your Ajax credit card application.

A Card Security Code is the three- or four-digit number that is printed on the credit card along with the card number. The CSC is designed to augment the authentication of the credit-card user. Many online stores that take credit cards also request that the user enter a CSC associated with the card. This act in itself, however, puts in jeopardy the secure identity of the CSC, so that this authetication technique is far from airtight.

NOTE

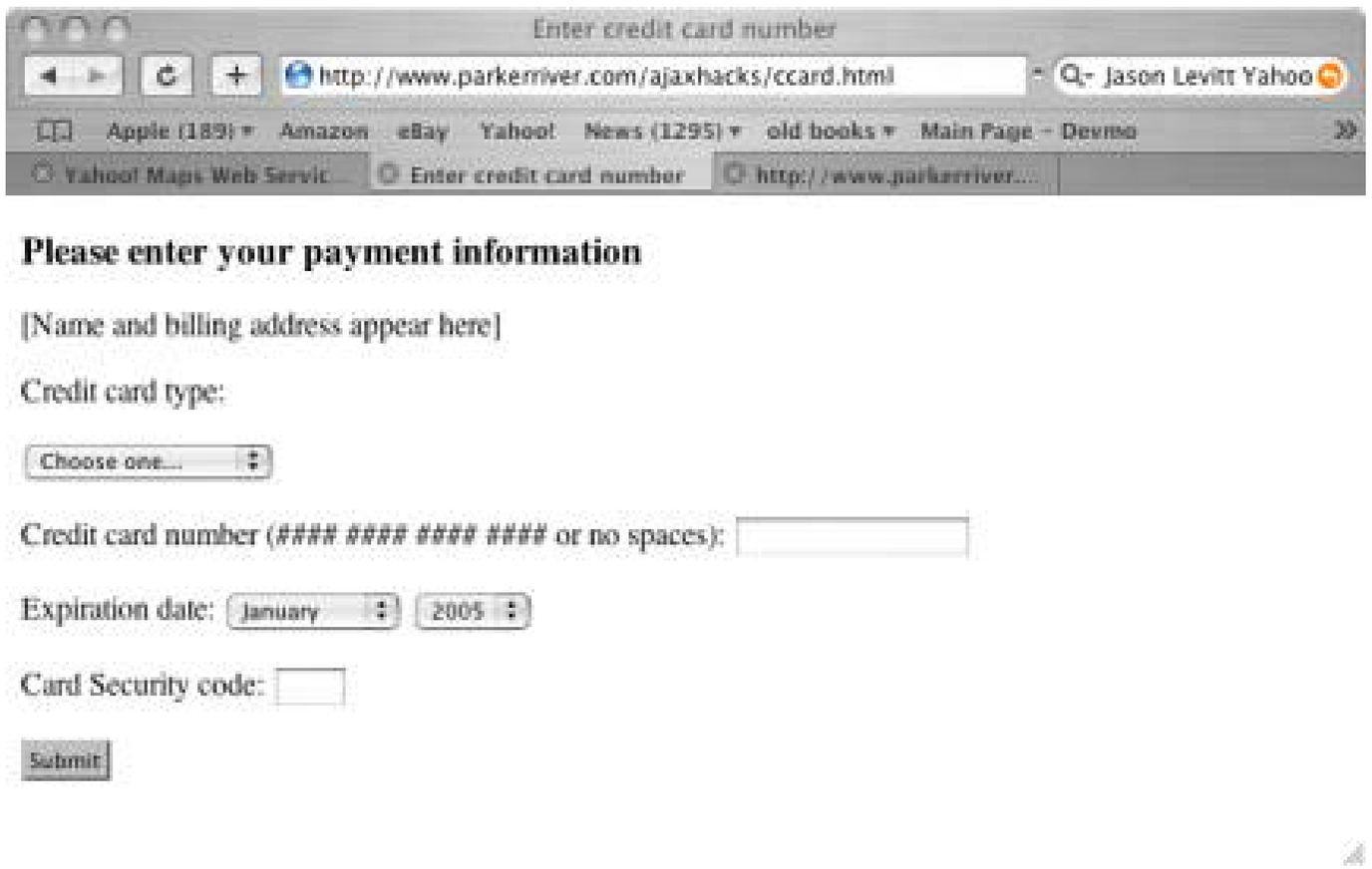
See http://en.wikipedia.org/wiki/Card_Security_Code.

The only entity that can validate a CSC is the merchant bank that has the responsibility for processing the credit card. There isn't a special formula like Luhn to validate it (it's only three or four numbers anyways!). However, this hack verifies that the user has entered the CSC correctly, as in using the following criteria:

- The field contains only numbers.
- If the credit card type is Mastercard, Visa, or Discover, the field has exactly three numbers.
- If the credit card is American Express, the field has exactly four numbers.

Figure 3-7 shows a web page that requests a CSC and other information (You may recognize it from the previous hack).

Validate Card Security Codes



This hack sets up the CSC validation so that when the user types in the text field and then clicks outside of the field or the XXXredpre#222XXX character, JavaScript code ensures that the previous criteria is met before continuing with the rest of the application. First, here is the web page code, which imports a JavaScript file cc.js.

```
XXXredpre#30XXX
```

Here is the code in cc.js that handles the Card Security Code text field.

```
XXXredpre#31XXX
```

The variable XXXredpre#223XXX refers to the text field where the user is supposed to enter the CSC The code sets the field's XXXredpre#224XXX event handler to a function that check's the security-code value. The function then generates a user message and disables the XXXredpre#225XXX buttor if the value is invalid. We want to disable XXXredpre#226XXX, because the application should prever the running of the form's XXXredpre#227XXX event handler until the security-code text field contains a valid value.

XXXredpre#228XXX validates the CSC field using regular expressions.

```
XXXredpre#32XXX
```

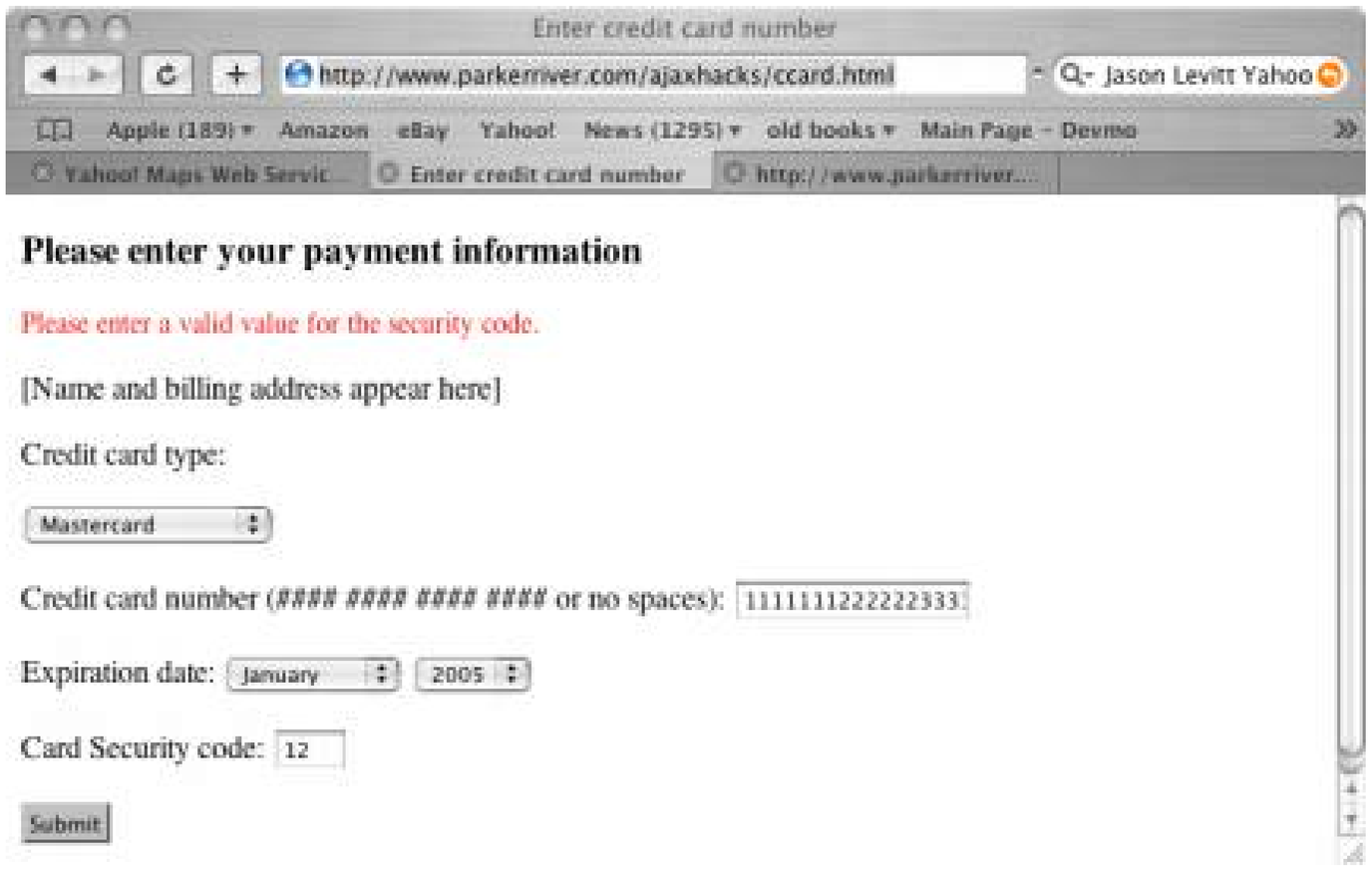
If the card is American Express, then the regular expression looks for a XXXredpre#229XXX containing four digits. The XXXredpre#230XXX object's XXXredpre#231XXX method returns XXXredpre#232XXX if its XXXredpre#233XXX parameter returns a match.

```
XXXredpre#33XXX
```

Similarly, the code checks the value associated with the three other credit-card companies for a

XXXredpre#234XXX containing three digits. A XXXredpre#235XXX return value from this method indicates an invalid value, and the user will see a red message and disabled XXXredpre#236XXX button, as in figure 3-8.

The security code text field checks itself



NOTE

You should trim the value in the security-code text field, because if the user inadvertantly types a space and three numbers (and is using say Mastercard) the regular expression will not find a match because the searched string will be " 123" instead of "123." The user will be irritated, because they seem to have typed the correct number. You can use the XXXredpre#237XXX method XXXredpre#238XXX, which replaces any space characters in the XXXredpre#239XXX with the empty XXXredpre#240XXX.

When the application has finished checking the card security code, then the user can click the submit button. Then an XXXredpre#241XXX event handler verifies the credit-card number, as in the previous hack, before sending a valid number to a server component to process a purchase order.

Validate A Postal Code

This hack checks what the user has entered in a text field, to make sure that the value represents the proper format for a U.S. zip code. This hack discusses the basics of validating a zip code; however, if you want to take it farther beyond validating the format of a zip, then you can use the code in Fetch a Postal Code as a secondary step to determine if the zip code is actually the correct one for the specified city and state.

Figure 3-9 shows what this hack's web page looks like. It is a sub-set of the typical form that asks fo the user's address information.

Enter the right zip code

The user types in a zip code (or fails to enter anything in the text field), then presses XXXredpre#242XXX or clicks outside of the field. The application's code then automatically validates what they typed.

NOTE

This hack only tests the first five digits of a zip code.

The code makes sure that the user entered five digits, and only five digits, into the field. The web

page imports the code in a JavaScript file named hacks3_7.js. Here is the content of the file.

XXXredpre#34XXX

The XXXredpre#243XXX event handler sets up the behavior for the application. XXXredpre#244XXX occurs when the browser completes loading the web page. At this point, the code creates an XXXredpre#245XXX event handler for the zip-code text field. This event handler is triggered when the user clicks XXXredpre#246XXX or outside of the zip-code field. Only if the user has typed a value into the city or zip-code text fields, as in XXXredpre#247XXX, then the code validates the format of the value in XXXredpre#248XXX.

This function uses a regular expression that represents a character phrase made up of five numbers. The code then tests the entered zip-code value against this regular expression to determine if the zip's format is correct.

> NOTE
>
> A regular expression represents a template for testing strings of characters. This regular expression, for example, looks for a line of text made up of just five numbers (the ^ means "beginning of the line" and $ is a special symbol for "end of the line."

If the format is not correct, then the code generates a user message. The web page devotes a XXXredpre#249XXX element with an id of "message" to contain these notifications.

XXXredpre#35XXX

Hacking the Hack

If you want to ensure that the five numbers represent a real zip code, then you can use the code in Fetch a Postal Code to request a postal code for a certain city and state. Fetch a Postal Code requests the zip from a web service; your code can then compare this value with the value entered by the user.

Some cities have multiple zip codes, and Fetch a Postal Code only returns the first zip code found for a city/state combination. Therefore, this method is not a foolproof way of validating every zip-code value. You could alter the server component that connects with the web service, to return all zip codes found for a specified city, but this method would still require more user interaction to narrow down the choices to one zip code.

Chapter 3. Web Forms

Web Form

Previous

E B V N
We are Vietnames

Submit Textfield Or Textarea Values To The Server Without A Refresh

Create a smooth transition between entering information into a textarea or text field and instantly trans

Ajax applications can automatically send to a server program the information that the user has entered The application code waits for the text widget's `onblur` event to occur, then uses the request object to s field or textarea. In many applications, this technique is preferable to requiring the user to click a `Submit` the form's values in a big clump to the server. It is also much faster and snappier in terms of the applica

NOTE

The `onblur` event is triggered when a web form control like a text field loses the keyboard focus, w instance by the user pressing the `TAB` key or clicking outside of the field. You can also use the `onkey` `onkeyup` event handlers to initiate this type of behavior in a text widget.

This hack's sequence of events for sending text to the server is:

- the user tabs into the field or clicks in a textarea,
- types some text,
- then presses TAB or clicks on another part of the page.

NOTE

A side effect of intervention-less form sending is that the user is not accustomed to this kind of beh The user could be put off or confused by web-form controls like text fields that dynamically submit interface should make it clear that "something is going to happen" when the user is finished with th message or progress indicator when the request object is sending the data (see Display a Progress

This hack includes a text field and a textarea that send HTTP requests with their values when the user is shows the web page loaded into a browser window.

No buttons need apply

The user types some information into the text field or textarea (the larger data-entry box), moves to the application automatically sends what they typed to a server component. Here is the HTML code for this p

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/2000/REC-xhtml1-20000126/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
     <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <script type="text/javascript" src="js/hacks_2_1.js"></script>
    <link rel="stylesheet" type="text/css" href="/css/hacks.css" />
    <title>Submit your information</title>
</head>
<body>
<h3>Get stats from textareas and textfields using Ajax</h3>
<form action="javascript:void%200" >
<div id="textf">
Enter a few words for submitting to our server:
<input type="text" name="tfield" id="tfield" size="35" />
</div>
<div id="texta">
Enter a phrase for submitting to our server:</span> **<textarea name="tarea" rows="20
</textarea>
</div>
</form>
</body>
</html>
```

Instead of a user clicking a button to send the form information, each text control sets the action in mot

When the user presses `TAB` or clicks outside of the text field, the code specified by the widget's`onblur` e

upcoming code sample shows how this event handler is set-up after the browser has finished loading th

The `script` tag in the HTML imports a JavaScript file, hacks_2_1.js. This file contains all of the code nec
Here is the code the file contains. This sample includes all of the code for sending a request and handlin
`handleResponse()` function). The next hack explains the related technique of inserting the server's resp
that shouldn't prevent you from peaking at `handleResponse()` if you want!

```
var formObj = null;
var formObjTyp = "";

//input field's event handlers
window.onload=function(){
    var txtA = document.getElementById("tarea");
    if(txtA != null){
        txtA.onblur=function(){if (this.value) { getInfo(this);}};    }

    var tfd = document.getElementById("tfield");
    if(tfd != null){
        tfd.onblur=function(){if (this.value) { getInfo(this);}};  }
}

function getInfo(obj){
    if (obj == null ) { return; }
    formObj=obj;
    formObjTyp =obj.tagName;
    if(formObjTyp == "input" || formObjTyp == "INPUT"){
        formObjTyp = formObjTyp + " "+formObj.type;
    }
    formObjTyp = formObjTyp.toLowerCase();
    var url = "http://www.parkerriver.com/s/webforms?objtype="+
              encodeURIComponent(formObjTyp)+"&val="+ encodeURIComponent(obj.value);
    httpRequest("GET",url,true);
}

//event handler for XMLHttpRequest
function handleResponse(){
    try{
        if(request.readyState == 4){
            if(request.status == 200){
                var resp = request.responseText;
                var func = new Function("return "+resp);
                var objt = func();
                if(formObjTyp == "textarea"){
                    if(formObj != null){
                        formObj.value = objt.Form_field_type +
                        " character count: "+objt.Text_length+
                        "\nWord count: "+
                        objt.Word_count+"\nServer info: "+
                        objt.Server_info;
                    }
                }  else if(formObjTyp == "input text"){
                    if(formObj != null){
```

```
                    formObj.value = objt.Form_field_type +
                    " # characters: "+objt.Text_length+
                    " Word count: "+objt.Word_count; }
                }
            } else {
                //request.status is 503
                //if the application isn't available;
                //500 if the application has a bug
                alert(
               "A problem occurred with communicating between the "+
               "XMLHttpRequest object and the server program.");
            }
        }//end outer if
    } catch (err)   {
        alert("It does not appear that the server is available "+
              "for this application. Please"+
              " try again very soon. \nError: "+err.message);

        }
    }

    /* Initialize a Request object that is already constructed */
    function initReq(reqType,url,bool){
        try{
            /* Specify the function that will handle the
            HTTP response */
            request.onreadystatechange=handleResponse;
            request.open(reqType,url,bool);
            request.send(null);
        } catch (errv) {
            alert(
                    "The application cannot contact the server "+
                    "at the moment. "+
                    "Please try again in a few seconds." );
        }
    }
    /* Wrapper function for constructing a Request object.
     Parameters:
      reqType: The HTTP request type such as GET or POST.
      url: The URL of the server program.
      asynch: Whether to send the request asynchronously or not. */
    function httpRequest(reqType,url,asynch){
        //Mozilla-based browsers
        if(window.XMLHttpRequest){
            request = new XMLHttpRequest();
            initReq(reqType,url,asynch);
        } else if (window.ActiveXObject){
             request=new ActiveXObject("Msxml2.XMLHTTP");
            if (! request){
                request=new ActiveXObject("Microsoft.XMLHTTP");
            }
            if(request){
```

```
            initReq(reqType,url,asynch);
        }  else {
            alert("Your browser does not permit the use "+
            "of all of this application's features!");}
    } else {
        alert("Your browser does not permit the use "+
        "of all of this application's features!");}
}
```

The code declares two top-level JavaScript variables: `formObjTyp` and `formObj` . The latter variable will h object (other functions in the code will need access to it later), and the former`var` holds a `string` repres name, such as "INPUT" or "TEXTAREA." This `string` is one of the parameters that the server component URL described beneath "Get the First Serve In").

NOTE

These variables are simply part of this hack's behavior and are not required in general for sending t request object.

As we mentioned previously, the code sets up the text widget's`onblur` event handlers when the browse You can accomplish this task in JavaScript by assigning a function to the window's`onload` event handler. code is an alternative to calling the JavaScript functions from within an HTML element's`onblur` attribute. that we are using for our page discourages the mixing of JavaScript function calls with the XHTML struct binds event handlers to functions within the imported JavaScript file instead.

```
window.onload=function(){
    var txtA = document.getElementById("tarea");
    if(txtA != null){
        txtA.onblur=function(){if (this.value) { getInfo(this);}};    }
    var tfd = document.getElementById("tfield");
    if(tfd != null){
        tfd.onblur=function(){if (this.value) { getInfo(this);}};  }
}
```

NOTE

Event handler event schmandler! These are simply attributes of an object to which your code can a of code that defines some behavior. So if you want to control how a radio button behaves when it's handler named `onclick` . As in:

NOTE

```
//Get a reference to a radio button element on a web //page
p(note).  var  rad  =  document.getElementById("radio1");
```

```
    p(note). //display a pop-up dialog window when it's clicked
    p(note). rad.onclick=function(){ alert("I was clicked!");};
```

These text fields are now hot; once the user types something and exits the controls, the user doesn't ha information they typed in is already off and running to the server.

Get the First Serve In

The main job of the text-field event handlers is to call the`getInfo()` function. This function grabs whate text widget and sends this value to the server.

```
function getInfo(obj){
    if (obj == null ) { return; }
    formObj=obj;
    formObjTyp =obj.tagName;
    if(formObjTyp == "input" || formObjTyp == "INPUT"){
        formObjTyp = formObjTyp + " "+formObj.type;
    }
    formObjTyp = formObjTyp.toLowerCase();
    var url = "http://www.parkerriver.com/s/webforms?objtype="+
              encodeURIComponent(formObjTyp)+"&val="+
              encodeURIComponent(obj.value);
    httpRequest("GET",url,true);
}
```

The `getInfo()` function takes as a parameter an object that represents the text field or textarea. We ar `input` or `textarea` objects so that the JavaScript code can use them to handle the server return value.

NOTE

Hint hint, the next hack shows how to display the server's return value inside these text widgets. S holds more information than a text field, the server sends back more data if the original object was a text field.

That last part, `httpRequest(&GET&,url,true)` , is the function call that actually sends the user's inform

However, a few things have to occur before the code calls that function, such as putting together a prop on the Internet). The server component is expecting a`string` describing what kind of form object the da application, the `string` will be formulated from the `tagName` property of the `Element` object (returning "I

NOTE

The code needs this value to tell the server whether its return value will be inserted into a text fielc is described in the next hack!

The code further refines the `input` object's description by what `input` sub-type it represents (text input? accomplished by appending the value of the `input` object's `type` property ("text" in our case) to the strir final string "input text."

In other words, this `type` property returns "text" only if the object represents an `<input type="text" .` `string` is forced to lower case and submitted to the server with the user's content.

```
formObjTyp =obj.tagName;
if(formObjTyp == "input" || formObjTyp == "INPUT"){
   formObjTyp = formObjTyp + " "+formObj.type;
}
formObjTyp = formObjTyp.toLowerCase();
var url = "http://www.parkerriver.com/s/webforms?objtype="+
              encodeURIComponent(formObjTyp)+"&val="+ encodeURIComponent(val);
httpRequest("GET",url,true);
```

The global JavaScript function `encodeURIComponent()` is a method for ensuring that certain characters s when they are included in URLs. Otherwise, your program may send a partial or truncated URL to the se The entire URL might look like this in a real case:

```
http://www.parkerriver.com/s/webforms?objtype=input%20text&
val=Hello%20There!
```

What's Next?

The `httpRequest()` function wraps the code that initializes and uses the request object, which works be user doesn't have to manually send the data. Chapter 1 and Hack #3 describes this function in detail.

So what happens to the submitted data next? That depends on your application. The next hack explores using JavaScript and Ajax to take an HTTP response and insert data into existing text fields or textareas

> NOTE
>
> The users can put tons of information in a large textarea, so in these cases use the POST method r request object as Hack #2 describes. For example, you can write the `httpRequest()` function as `httpRequest("POST",url,true)` and the request object's `send()` method has the POST querystrin `request.send(val=Hello%20there%20and%20a%20lot%20of%20other%20stuff);`

?

Display Text Field Or Textarea Values Using Server Data

Have server information magically appear in text boxes without the web page refreshing.

You can have a server component interact with information that the user enters in a text box, withou the jarring effect of the page reconstituting every time the user enters new information. A typical example is a spell checker or auto-complete field (see Hack #68). Using the request object as an intermediary, a server component responds in real time to what the user types.

This hack displays an automatic server response, so that the response appears as if by magic in the text field or textarea, without anything else changing in the web page. The hack is an extension of our previous hack, which used the request object to submit textarea or text field values to a server component behind the scenes.

This hack takes the information the user has submitted and displays a character count and word count in the same field. You could accomplish the same thing with client-side JavaScript, of course, but just to prove that a server component is doing the work, the hack displays some information about the server in the text area.

Figure 2-2 shows the web page after the user has entered some data into the text field.

Enter data and elicit a response

Figure 2-3 shows the browser window after the user has entered data in both fields then clicked TAB.

This code is the HTML page we are using. It imports a JavaScript file named hacks_2_1.js, which contains the code that does most of the work.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/2000/REC-xhtml1-20000126/DTD/xhtml1-strict.dtd">
<html  xmlns="http://www.w3.org/1999/xhtml"  xml:lang="en"  lang="en">
<head>
     <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <script type="text/javascript" src="js/hacks_2_1.js"></script>
    <link rel="stylesheet" type="text/css" href="/css/hacks.css" />
    <title>Submit your information</title>
</head>
<body>
<h3>Get stats from textareas and textfields using Ajax</h3>
<form action="javascript:void%200" >
<table  border="0"><tr>
<td>Enter a few words for submitting to our server:
**<input  type="text"  name="tfield"  size="35"/>**</td></tr>
<tr><td valign="top">Enter a phrase for submitting to our server:
**<textarea  name="tarea"  rows="20"  cols="20">**
</textarea></td>  </tr>
</table></form>
</body>
</html>
```

The last hack explained how the code submits the user's information without refreshing the web page. In other words, after the user has typed in their information and presses `TAB` or otherwise clicks outside of the field, just the piece of data they have added to the text field or textarea is sent in an HTTP request to the server.

An `onblur` event handler calls a function `getInfo()`, passing in the text field or textarea object as a parameter.

> NOTE
>
> An `onblur` event handler points to a function that the browser calls when the user's keyboard focus leaves a form control, as in pressing the `TAB` character or clicking outside of the text field.

The entire code for this behavior appears in the previous hack, so we will not reproduce it again here We will show the code in the `getInfo()` and `handleResponse()` functions, which do the work of sending the server component the information it needs, then handling the server's response.

```
function getInfo(obj){
    if (obj == null ) { return; }
    formObj=obj;
    formObjTyp =obj.tagName;
    if(formObjTyp == "input" || formObjTyp == "INPUT"){
        formObjTyp = formObjTyp + " "+formObj.type;
    }
    formObjTyp = formObjTyp.toLowerCase();
    var url = "http://www.parkerriver.com/s/webforms?objtype="+
        encodeURIComponent(formObjTyp)+"&val="+
        encodeURIComponent(obj.value);
    httpRequest("GET",url,true);
}
```

This function passes along to the server component the user's typed-in information as part of the `val` parameter. In addition, the `obj` parameter holds a reference to the text control where the user typed, such as a text field or textarea. The reference is specifically a Document Object Model (DOM) object, such as an `HTMLInputElement` or `HTMLTextAreaElement`.

> NOTE
>
> You do not have to worry about the DOM object tree at this point (although it is interesting!). The HTML code for this hack refers to the particular text control by using the `this` keyword in the `onblur` attribute. Then the `getInfo()` function knows exactly what kind of text control the user is interacting with, a text field or textarea, by accessing the object's `tagName` property. JavaScript: The Definitive Guide by David Flanagan has excellent coverage on programming the DOM objects.

Instant Server Messaging

The server program takes the user's typed in information and sends back the associated number of characters and words. To make this response information palatable to our receiving code, the server returns its information in a known format named JavaScript Object Notation (JSON) (See Hack #6 and www.json.org). JSON is similar to XML in that it structures data for the purpose of making the data easier for software to digest and work with.

> NOTE
>
> Your own program could simply return data using an XML language or a simple`string`. Using JSON as a return value is a programmer's personal preference. It is particularly useful if the server client is composed of JavaScript code.

This code shows a typical JSON server return value, if the user typed 55 words into a textarea.

```
{
Form_field_type: "textarea",
Text_length:  "385",
Word_count:  "55",
Server_info:  "Apache  Tomcat/5.0.19"
}
```

This code represents a JavaScript object with four different properties:`Form_field_type`, `Text_length`, `Word_count`, and `Server_info`. See the explanation about how these properties are used after the next code sample.

Now the hack takes this information and plugs it back into the textarea. This is the job of the `handleResponse()` function.

```
//event handler for XMLHttpRequest
function handleResponse(){
    try{
        if(request.readyState == 4){
            if(request.status == 200){
                var resp = request.responseText;
                if(resp != null){
                    var func = new Function("return "+resp);
                    var objt = func();
                    if(formObjTyp == "textarea"){
                        if(formObj != null){
                         formObj.value = objt.Form_field_type +
                         " character count: "+objt.Text_length+
                         "\nWord count: "+
                         objt.Word_count+"\nServer info: "+
                         objt.Server_info;
                        }
                    }  else if(formObjTyp == "input text"){
                         if(formObj != null){
```

```
                        formObj.value = objt.Form_field_type +
                        " # characters: "+objt.Text_length+
                        " Word count: "+objt.Word_count; }
                    }
                }
            } else {
                //request.status is 503
                //if the application isn't available;
                //500 if the application has a bug
                alert(
                 "A problem occurred with communicating "+
                 "between the XMLHttpRequest object and "+
                 "the server program.");
            }
        }//end outer if
    } catch (err)   {
        alert(err.name);
        alert("It does not appear that the server "+
              "is available for this application. Please"+
              " try again very soon. \nError: "+err.message);
    }
}
```

This code grabs the response as text. Since the text is already formatted in JSON syntax (as an object literal in JavaScript), the code uses a special technique that Hack #6 describes. A `Function` constructor returns the text as a JavaScript object. In this case, the variable `objt` now refers to the server component's response in an object-centric way, so you can access the server information with syntax such as `objt.Server_info`.

The latter code piece accesses the `Server_info` property of the object that the variable `objt` refers to.

```
var resp =  request.responseText;
var func = new Function("return "+resp);
//call the function and return the object which
//the objt variable now points to
var objt = func();
```

The rest of the code goes about inserting this information back into the textarea using this syntax.

```
if(formObjTyp == "textarea"){
  if(formObj!= null){
    formObj.value = objt.Form_field_type +
    " character count: "+objt.Text_length+
    "\nWord count: "+
    objt.Word_count+"\nServer info: "+
    objt.Server_info;
 }
}
```

Figure 2-3 shows what the `textarea` looks like after the information is placed inside it.

We are able to get access to the `textarea` because a top-level JavaScript variable refers to it: `formObj`. One of the keys to this code is setting the `value` of a textarea or text field with a "dot property-name" syntax common to JavaScript, as in `formObj.value`.

NOTE

The server program sends more information back to a `textarea` than a text field, including line breaks (`\n` in JavaScript), because the `textarea` is a big box that can hold more sentences. You cannot include line breaks in a text field, for instance, because it only holds one line (even if that line can be numerous characters).

The code formats the value of the `textarea` by connecting strings to the properties of the object the server returned, as in `& character count: &+objt.Text_length`. Although in a conventional web interface, the user expects a `textarea` or field to be reserved for their own data entry, this hack demonstrates how to provide direct feedback to what they are typing into a particular field.

Submit Selection- List Values To The Server Without A Browser Refresh

Whisk the user's multiple list choices off to the server without delay.

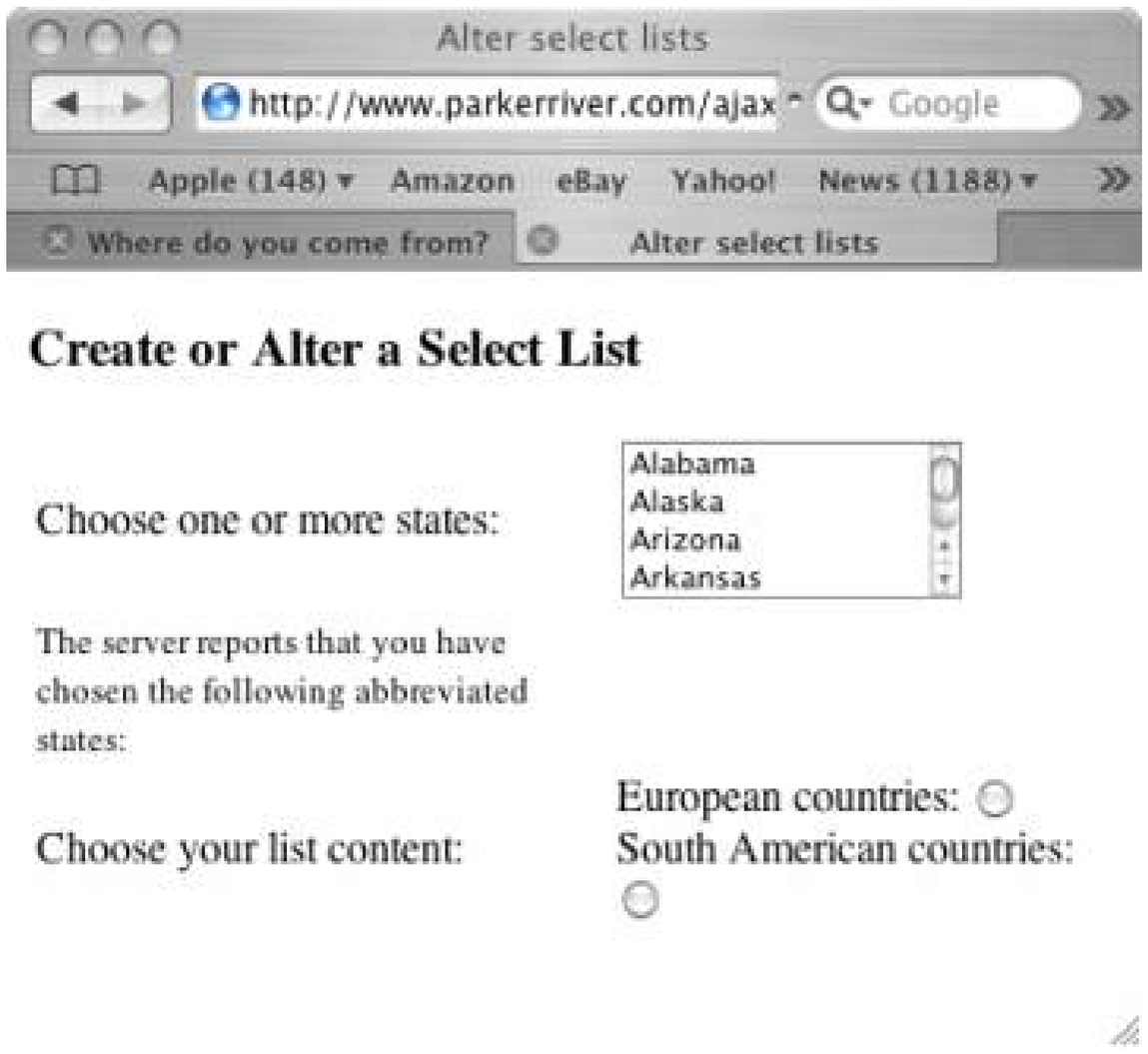
A number of web developers will see the advantage of taking the multiple choices of a user in a list button or `select list`, and sending them to a server program using the request object, rather than requiring the user to click a button and send the entire form. This gives the application greater responsiveness, and increases efficiency by sending discrete values rather than clumps of information.

This hack sends the user's choices of U.S. states to a server program when they move the keyboard focus away from the popup button. The `select` element looks like this in the HTML code that underlies the web page.

```
<select name="_state" multiple="multiple" size="4">
```

This is a popup button that allows the user to choose more than one item. When the keyboard focus moves from the `select` list via a `TAB` or click elsewhere on the page the code defined by the element's `onblur` event handler executes. An upcoming section shows this code. The `size=4` part indicates that four state names can be displayed at a time in the `select` list. Figure 2-4 shows the page loaded into the Safari browser.

Multiple choices for immediate delivery



A JavaScript file named hacks_2_4.js declares all the code this hack needs. Here is the entire HTML for the web page.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/2000/REC-xhtml1-20000126/DTD/xhtml1-strict.dtd">
<html  xmlns="http://www.w3.org/1999/xhtml"  xml:lang="en"  lang="en">
<head>
     <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <script type="text/javascript" src="js/hacks_2_4.js"></script>
    <link rel="stylesheet" type="text/css" href="/css/hacks.css" />
    <title>Alter select lists</title>
</head>
<body>
<h3>Create or Alter a Select List</h3>
<form  action="javascript:void%200"  >
    <table border="0">
        <tr><td>Choose one or more states: </td><td>
**<select  name="_state"  multiple="multiple"  size="4">**
            <option value="al">Alabama</option>
            <option value="ak">Alaska</option>
            <option value="az">Arizona</option>
            <option value="ar">Arkansas</option>
            <option value="ca">California</option>
            <option value="co">Colorado</option>
            <option value="ct">Connecticut</option>
            <option value="de">Delaware</option>
            <option value="dc">District of Columbia</option>
            <option value="fl">Florida</option>
```

```
            <option value="ga">Georgia</option>
            <option value="hi">Hawaii</option>
            <!&#8212;snipped...-->
        </select></td>  </tr>
        <tr><td**><span id="select_info" class=**
**      "message">The server reports that you have chosen the **
**following abbreviated states: </span>**
            <tr><td>Choose your list content:</td>
<td>European countries:
<input type="radio" name="countryType" id="euro" value="euro" />
South American countries:
<input type="radio" name="countryType" id=
"southam" value="southam" />
</td></tr>
<tr><td><div id="newsel"></div></td></tr>
    </table></form>
</body>
</html>
```

A `span` element contains a message that the user will see after they make some choices in the `select` list. This message is styled by a CSS rule in the file hacks.css. But first, we will look at the code that submits the user's choices to a server. The code is a little complicated at first glance, but bear with us as what it accomplishes is really quite simple.

The code takes all of the options associated with the `select` element, and determines which of them the user has selected. These options represent the user's choice(s) of U.S. states. The code takes each one of the selected options and stores them in a `string`, separated by commas (if there is more than one choice) so that the string looks like `ma,nh,vt`. @@

This task would be made easier if the browser stored the selected values in one convenient place, such as a `value` property of the `select` object, but this isn't the case! You have to grab the bunch of options, determine which ones were selected, and store those somewhere, such as in a JavaScript `array`.

NOTE

A `select` element contains `option` elements, as in `<select name="_states"><option value="vt">Vermont</option>...</select>`. In the DOM, the `select` element is represented by a `select` object that has an `options` property, an `array` of `option` objects. You get the value of each `option`, which the user sees as words in a list like "Vermont," by using the `value` property of an `option` object. Phew, fun to code, but endless objects and properties!

```
function getSelectInfo(selectObj){
    if (selectObj == null) { return; }
    formObj=selectObj;
    formObjTyp =formObj.tagName.toLowerCase();
    var optsArray = formObj.options;
    var selectedArray = new Array();
```

```
var val = "";
//store selected options in an Array
for(var i=0,j=0; i < optsArray.length; i++){
    if(optsArray[i].selected) {
        selectedArray[j]=optsArray[i].value;
        j++;
    }

}
//create a comma-separated list of each
//selected option value
for(var k = 0; k < selectedArray.length; k++){
    if(k !=selectedArray.length-1 ) { val +=selectedArray[k]+",";}
    else {val +=selectedArray[k]; }
}
var url = "http://www.parkerriver.com/s/webforms?objtype="+
          encodeURIComponent(formObjTyp)+"&val="+ encodeURIComponent(val);
    httpRequest("GET",url,true);
}
```

The server component is expecting an `objtype` parameter, which in this case equals "select." We are also sending the `string` of comma-separated choices, which the `val` parameter points to. Since we are using JavaScript's global function `encodeURIComponent()`, the commas will be encoded into `%2C`, since certain punctuation marks are not allowed in the character strings that are sent to server components.

> NOTE
>
> `encodeURIComponent()` is a global function that is designed to encode portions of a Uniform Resource Indicator (URI), which is a fancy name for the addresses you enter into browser location fields to download a web page. This function encodes punctuation characters that have special purposes in URIs, such as /, :, @, and ;, as well as space characters, into their hexadecimal equivalents. For example, a ";" character is encoded into %3B. `encodeURIComponent()` does not encode ASCII numbers or letters. Use `encodeURIComponent()` to encode query strings or path information that your JavaScript code is handling.

We could have used spaces, colons or some other delimiter to separate each choice. Here is an example of a URL sent by this JavaScript:

```
http://www.parkerriver.com/s/webforms?objtype=select&   val=ma%2Cvt%2Cny
```

This URL contains ma, vt, and ny as choices; after the `val` parameter is decoded it will read `ma,vt,ny`.

Okay, Now What Happens?

I thought you'd never ask. The server grabs the selected values and redirects them back to the application with some extra information. This is where the displayed message comes to the fore. It will display the user's current choices and the brand of server that the application is connected with. Figure 2-5 shows the browser page after the user has made some choices and moved the keyboard focus from the `select` list.

Instant feedback on list choices

The message changes dynamically without anything else rebuilt or refreshed on the web page. It gives the user instant feedback while connected to a server, without any browser roundtrips. How does this happen? Here is the JavaScript for the`handleResponse()` function, which deals with the server return value. We have highlighted only the code that converts the return value into the user message.

```
//event handler for XMLHttpRequest
function handleResponse(){
    try{
        if(request.readyState == 4){
            if(request.status == 200){
                if(formObjTyp.length > 0 && formObjTyp == "input"){
                //working with existing radio button
                    var resp =  request.responseText;
                    //return value is a JSON array
                    var func = new Function("return "+resp);
                    var objt = func();
                    var sel = document.createElement("select");
```

```
            sel.setAttribute("name","countries");
            createOptions(sel,objt);
            var newsel = document.getElementById("newsel");
            reset(newsel);
            newsel.appendChild(sel);
        } else  if(formObjTyp.length > 0 && formObjTyp == "select"){
            var resp =  request.responseText;
            //return value is a JSON object literal
            var func = new Function("return "+resp);
            var objt = func();
            var fld = document.getElementById("select_info");
            if(fld != null){
                fld.innerHTML = "The server <strong>"+objt.Server_info+
                                "</strong> reports that you have chosen"+
                                "&#x00A; the following "+
                                "abbreviated states: <strong>"+
                                objt.Selected_options+"</strong>";
            }
        }
    } else {
        //request.status is 503
        //if the application isn't available;
        // 500 if the application has a bug
        alert(
          "A problem occurred with communicating "+
          "between the XMLHttpRequest object and the "+
          "server program.");
    }
  }//end outer if
} catch (err)   {
  alert("It does not appear that the server "+
        "is available for this application. Please"+
        " try again very soon. \nError: "+err.message);

}
```

Hello JSON, Again

The server provides its answer in a special, useful syntax, JavaScript Object Notation (JSON). See Hack #6. This is a `string` that can be easily converted by the client-side browser code into a JavaScript object. An example of a server return value, which some readers may recognize as an object literal, is:

```
{
Server_info:  "Apache  Tomcat/5.0.19",
Selected_options: "vt ny nh ma"
}
```

This code represents an object that has two properties, `Server_info` and `Selected_options`. To

derive the property values from the object, you use the "dot" syntax, as in`obj.Selected_options`. If the `obj` variable was set to the prior code's object literal, then the latter code line would return "vt ny nh ma." Hack #6 describes the JavaScript code to use for sending and handling JSON syntax.

A Dabble of Serverside

For those interested in one method of sending JSON formatted values back to Ajax, here is a method written in Java that is used for this hack. This method takes as parameters the property names and values in a `String`, and the character, such as a comma, that separates the property names from the values.

```
public static String getJsonFormat(
                                String propValues, String delim) {
         if(propValues == null || propValues.length()==0) { return "";}

        StringBuffer structure = new StringBuffer("");
        structure.append("{\n");
        if (delim == null || delim.length() == 0) { delim = ",";}
         /* We're expecting comma-separated values such as prop1,val1,
        prop2,val2  etc. */
        StringTokenizer toke = new StringTokenizer(propValues,delim);
        int j = 0;
        int c =  toke.countTokens();
         for(int i = 1; i <=c; i++) {
            j = i%2;
            if(j != 0) { structure.append(toke.nextToken()).
            append(": ");   }//it's a property name
            else { structure.append("\"").append(toke.nextToken()).
            append("\""); //it's a property value
                if(i != c){structure.append(",");}
                structure.append("\n");
            }
        }
        structure.append("}");
        return structure.toString();
}
```

If the Java servlet calls the method this way, `getJsonFormat("Server_info,Apache Tomcat,Selected_options,ca ma nh ny",",")`, then the method returns:

```
{
Server_info: "Apache Tomcat",
Selected_options: "ca ma nh ny"
}
```

DOM API

The hack's next step is to store this return value in a variable, so that the JavaScript can display its value to the user.

```
var func = new Function("return "+resp);
var objt = func();
var fld = document.getElementById("select_info");
```

Hack #6 explains this use of the`Function` constructor to take advantage of the JSON format. Suffice it to say, the variable `objt` now contains the properties/values we are interested in.

The variable `fld` represents the `div` element we reserved on the HTML page for containing this user message from the server. `getElementById()` is a DOM API method for getting a reference to a tag in HTML code, so the code can change its behavior or alter its appearance.

```
if(fld != null){
  fld.innerHTML = "The server <strong>"+objt.Server_info+
  "</strong> reports that you have chosen"+
  "&#x00A; the following "+
  "abbreviated states: <strong>"+
  objt.Selected_options+"</strong>";
}
```

Displaying the object's information is easy using syntax such as`objt.Selected_options`. Figure 2-4 shows the states that the user has chosen and the name of the server software. This message will change automatically as you make different selections in the list. the information is derived from a server rather than just being generated by client-side JavaScript!

Dynamically Generate A New Selection List With Server Data

Create a list of choices on a web page that automatically branches into a new selection list without refreshing the entire page.

Some choices in a user interface naturally lead to a subsequent set of choices. An example is a support page for computer hardware, where one `select` list has a choice for hardware platform, such as Apple or HP, which generates when the user makes a choice a second list of related operating systems, and on and on. Ajax can shine in these situations where the user interface can automatically be customized for the browser user, as well as where the content for the `select` list must come from a server.

You could set this page up using only dynamic HTML, where JavaScript creates the new `select` list. However, the choices for the new list would have to be hard-coded into the JavaScript. Ultimately, this content for new lists will change, creating an awkward situation where developers have to constantly add persistent lists to an existing JavaScript file. Without being able to store these values in a sensible location such as in a database or other persistent store, this application model becomes unwieldy.

This hack displays two radio buttons on a web page where users can select either European countries or South American countries. Either choice results in a new selection list but with different content. Figure 2-6 shows the web page for the hack.

Select a category of countries

Here is the HTML code underlying the web page. We removed most of the long`select` list above the radio buttons, because that code is related to the previous hack.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/2000/REC-xhtml1-20000126/DTD/xhtml1-strict.dtd">
<html  xmlns="http://www.w3.org/1999/xhtml"  xml:lang="en"  lang="en">
<head>
     <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <script type="text/javascript" src="js/hacks_2_4.js"></script>
    <link rel="stylesheet" type="text/css" href="/css/hacks.css" />
    <title>Alter select lists</title>
</head>
<body>
<h3>Create or Alter a Select List</h3>
<form  action="javascript:void%200"  >
     <table border="0">
         <tr><td>Choose one or more states: </td>
<td>  <select  name="_state" multiple="multiple" size="4">
             <option value="al">Alabama</option>
             <!&#8212;more options... -->
         </select></td>  </tr>
         <tr><td><span id="select_info" class="message">
The server reports that you have chosen the following abbreviated states:
</span>
<tr><td>Choose your list content:</td><td>European countries:
<input type=
       "radio" name="countryType" id="euro" value=
       "euro" /> South American countries:
```

```
        <input type="radio" name=
        "countryType" id="southam" value="southam" /></td></tr>
            <tr><td><div id="newsel"></div></td></tr>
    </table></form>
</body>
</html>
```

The purpose of this code is to create a new `select` list whenever the browser user clicks on a radio button. With radio buttons on a web page, only one can be selected at a time. If you select a certain button, the other one(s) will automatically be de-selected.

Central to this hack is each radio button's `onclick` event handler. This is an attribute of an HTML element that points to a JavaScript function. The function's code will execute each time the user clicks on a radio button. In other words, if the button is de-selected and the user ticks it, then the code will call the function named `generateList()`.

The code appears in the file that the web page imports: hacks_2_4.js. Here is the code that controls the response to the user's radio-button clicks.

```
//input field's event handlers
window.onload=function(){
    var eur = document.getElementById("euro");
    if(eur != null){
        eur.onclick=function(){generateList(this);  };}
    var southa = document.getElementById("southam");
    if(southa != null){
        southa.onclick=function(){generateList(this);  };}
}
```

Each `onclick` event handler points to a "function literal" which simply calls `generateList()`. You will notice the `this` keyword as a parameter. That will hold a reference to each radio button that was clicked, so that the function's code can grab the button's value and send the value to a server component.

Presto, New Lists

The `generateList()` function is defined in a file named hacks_2_4.js. The HTML code for the web page imports this file using a `script` element. Here are the highlights of this file, with the emphasis on the functions used to generate a new list.

```
var formObj = null;
var formObjTyp = "";

function generateList(obj){
    if (obj == null ) { return; }
    if(obj.checked) {
        formObj=obj;
        formObjTyp =formObj.tagName.toLowerCase();
        var url = "http://www.parkerriver.com/s/selectl?countryType="+
```

```
                    encodeURIComponent(obj.value);
        httpRequest("GET",url,true);
    }
}

//event handler for XMLHttpRequest
function handleResponse(){
    try{
        if(request.readyState == 4){
            if(request.status == 200){
                if(formObjTyp.length > 0 && formObjTyp == "input")   {
                   if (resp != null){
                        //return value is a JSON array
                        var objt = eval(resp);
                        //create a new select element
                        var sel = document.createElement("select");
                        sel.setAttribute("name","countries");
                        //give the select element some options based
                        //based on a list of countries from the server
                        createOptions(sel,objt);
                        //the div element within which the
                        //the select appears
                        var newsel = document.getElementById("newsel");
                        reset(newsel);
                        newsel.appendChild(sel);
                    }
                } else  if(formObjTyp.length > 0 && formObjTyp == "select"){
                    //code edited out here for the sake of brevity...
                    }
                }
            } else {
                //request.status is 503 if the application isn't available;
                //500 if the application has a bug
                alert("A problem occurred with communicating between "+
                "the XMLHttpRequest object and the server program.");
            }
        }//end outer if
    } catch (err)   {
        alert("It does not appear that the server is available"+
             " for this application. Please"+
              " try again very soon. \nError: "+err.message);

    }
}

function createOptions(sel,_options) {
    //_options is an array of strings that represent the values of
    //a select list, as in each option of the list.
    //sel is the select object
    if(_options == null || _options.length==0) { return;}
    var opt = null;
    for(var i = 0; i < _options.length; i++) {
```

```
        opt = document.createElement("option");
        opt.appendChild(document.createTextNode(_options[i]));
        sel.appendChild(opt);
    }
}
//remove any existing children from an Element object
function reset(elObject){
    if(elObject != null && elObject.hasChildNodes()){
        for(var i = 0; i < elObject.childNodes.length; i++){
            elObject.removeChild(elObject.firstChild);
        }
    }
}
/* Initialize a Request object; code omitted, see Hack #11*/
```

When the user clicks their mouse on a radio button, the control will either indicate a selected state, o if it was already selected this action will de-select the button. The`onclick` event handler does not differentiate between checked or unchecked radio buttons; it is designed simply to react when the button has been clicked. Just to make sure the radio button is selected (even though the button is designed to be selected based on a click event), our code first checks whether the object was in a checked state before it begins creating a new`select` list.

```
function generateList(obj){
    if (obj == null ) { return; }
    if(obj.checked) {
        formObj=obj;
        formObjTyp =formObj.tagName.toLowerCase();
        var url = "http://www.parkerriver.com/s/selectl?countryType="+
                  encodeURIComponent(obj.value);
        httpRequest("GET",url,true);
    }
}
```

Querying the Server

The code queries a server with the value of the checked radio button. Recall that the new`select` list will contain choices, the words the user sees such as "United Kingdom," which are stored on the server side. To determine which of these sets of values to acquire from the server, the European or South American countries, we include a parameter in the request URL named`countryType`. The value for this parameter derives from the radio button's value attribute, as in:

```
<input type="radio" name="countryType" id="southam" **value="southam"** />
```

The code sends this information to the server using the request object and the`httpRequest()` function. Hack #2 and #11 (among others) describes this function, which wraps the initialization of the request object and the calling of its methods. The URL the request object uses to connect with the server might look like <literal><a href="http://www.parkerriver.com/s/selectl?countryType=euro">http://www.parkerriver.com/s/selectl?countryType=euro.

The code then receives the response and builds the new`select` list. It pulls the values out of the response using our familiar `handleResponse()` function, which shows up in most of the other hacks. Here are the key lines of JavaScript for this hack.

```
if(request.readyState == 4){
    if(request.status == 200){
      if (resp != null){
      //return value is a JSON array
      var objt = eval(resp);
      //create a new select element
      var sel = document.createElement("select");
      sel.setAttribute("name","countries");
      //give the select element some options based
      //based on a list of countries from the server
      createOptions(sel,objt);
      //the div element within which the
      //the select appears
      var newsel = document.getElementById("newsel");
      reset(newsel);
      newsel.appendChild(sel);
```

The server's return value can be used as a JavaScript`array`, which looks like `["Spain", "Germany", "Austria"]`. The code takes the `string` return value and convert it into an `array` with `eval()`. See Hack #6. The JavaScript then uses the DOM API to create a new`select` element. It's a good idea to give the newly generated HTML element a name and a value, in case your application calls for later submitting these values to a server component.@

```
var sel = document.createElement("select");
sel.setAttribute("name","countries");
```

`Using the` array `of values returned by the server, the` createOptions() `function populates the` select `element with a new` option `element pointing at each` array `member.` The end result is a new `select` element built from scratch that looks like `<select name=&countries&><option>United Kingdom</option>...</select>`. Here is the code for the `createOptions()` function.@

```
function createOptions(sel,_options) {
    // options is an array of strings that represent the values of
    //a select list, as in each option of the list. sel is the select object
     if(_options == null || _options.length==0) { return;}
    var opt = null;
    for(var i = 0; i < _options.length; i++) {
        opt = document.createElement("option");
        opt.appendChild(document.createTextNode(_options[i]));
        sel.appendChild(opt);
    }
}
```

The `_options` variable contains all the country names. The code uses the DOM API to create each

new `option` element, call the element's `appendChild()` method to add the country name to the `option`, and finally call the `select` element's `appendChild()` method to add the `option` to the `select` list.

Final Step

We have to figure which block-level element in the HTML will hold the new `select` element, rather than willy-nilly just throwing the `select` somewhere within the `body` tag. For this purpose we created a `div` element with the `id newsel`. The `div` element appears beneath the radio buttons on the page, but since it initially does not contain any visible HTML elements the user will not be aware of it.

```
<div id="newsel"></div>
```

The code uses another popular DOM method named `getElementById()` to get a reference to this `div`, then append to it the `select` element as a node.

```
var newsel = document.getElementById("newsel");
reset(newsel);
newsel.appendChild(sel);
```

To prevent users from continuously clicking the radio buttons and generating a million new lists, another method named `reset()` first checks the `div` for any existing child nodes, which would represent a previously created `select` element. The function deletes any existing nodes before the code adds a new `select` list inside the `div`.

```
function reset(elObject){
    if(elObject != null && elObject.hasChildNodes()){
        for(var i = 0; i < elObject.childNodes.length; i++){
            elObject.removeChild(elObject.firstChild);
        }
    }
}
```

Figure 2-7 shows the web browser page after the user has clicked one of the radio buttons.

Choose your country

Create or Alter a Select List

Choose one or more states:

Alabama
Alaska
Arizona
Arkansas

The server reports that you have chosen the following abbreviated states:

Choose your list content:

European countries:
South American countries:

Brazil

E B V N

We are Vietnames

Populate An Existing Selection List

Give the browser user an option to modify an existing list before they make and submit their choices

Imagine that you have a list of U.S. states as in the `select` element of Submit Selection-List Values To The Server Without A Browser Refresh. As part of registering a customer, you ask them what state they live in (for sales-tax purposes, say). Now you want register customers from other continents, because your product can now be distributed overseas. You do not want to include in the `select` element every country on earth, for geo-political reasons (countries frequently change, such as the former Yugoslavia), and the `select` list will be too big to fit nicely on the page. Your application will query the server for specific countries associated with the name of a continent, which is passed to the server program.

The user thus has the option to make a selection that adds a sub-set of `select` options associated with specific countries to the page.

For example, provide a `select` list of continents. When the user makes a selection, the countries enclosed by that continent are derived from the server and automatically added to an existing `select` list without the page being refreshed. Figure 2-8 shows the web page for this hack, which is based on the previous hack containing the `select` list of U.S. states.

Add options to a list.

The user selects a continent in the top-level `select` list. This action triggers the `onclick` event for the `select` element. Here is the HTML code for the page.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/2000/REC-xhtml1-20000126/DTD/xhtml1-strict.dtd">
<html  xmlns="http://www.w3.org/1999/xhtml"  xml:lang="en"  lang="en">
<head>
     <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <script type="text/javascript" src="js/hacks2_6.js"></script>
    <link rel="stylesheet" type="text/css" href="/css/hacks.css" />
    <title>Alter select lists</title>
</head>
<body>
<h3>Add Entries to a Select List</h3>
<form  action="javascript:void%200">
     <table border="0">
         <tr><td>Add your country: <select id="cts" name="_continents">
<option value="southam">South America</option>
<option value="euro">Europe</option>
</select></td></tr><tr><td>Choose one or more states: </td>
<td> <select id="sts" name="_state" multiple="multiple" size="4">
            <option value="al">Alabama</option>
            <option value="ak">Alaska</option>
            <option value="az">Arizona</option>
            <option value="ar">Arkansas</option>
            <option value="ca">California</option>
            <option value="co">Colorado</option>
            <option value="ct">Connecticut</option>
            <option value="de">Delaware</option>
            <option value="dc">District of Columbia</option>
            <option value="fl">Florida</option>
            <option value="ga">Georgia</option>
            <option value="hi">Hawaii</option>
            <!&#8212;snipped here...-->
        </select></td></tr>

    </table></form>
</body>
</html>
```

All of the JavaScript appears in the file hacks2_6.js. Here is the contents of this file, omitting the creation and initialization of the request object, which the first hack in this chapter and several other hacks show.

```
var origOptions = null;
/*set up the onclick event handler for the "countries"
select list */
window.onload=function(){
    var sel = document.getElementById("cts");
```

```
    var sel2 = document.getElementById("sts");
    if(sel != null){
        sel.onclick=function(){
            addCountries(this)};
    }
    origOptions = new Array();
    //save the original select list of states so that
    //it can be reconstructed with just the original states
    //and the newly added countries
    for(var i = 0; i < sel2.options.length; i++){
        origOptions[i]=sel2.options[i];
    }
}

function addCountries(obj){
    if (obj == null ) { return; }
    var url = "";
    var optsArray = obj.options;
    var val = "";
    for(var i=0; i < optsArray.length; i++){
        if(optsArray[i].selected) {
            val=optsArray[i].value; break;
        }

    }
    url = "http://www.parkerriver.com/s/selectl?countryType="+
          encodeURIComponent(val);
    httpRequest("GET",url,true);

}

//event handler for XMLHttpRequest
function handleResponse(){
  try{
    if(request.readyState == 4){
      if(request.status == 200){
         var resp =  request.responseText;
         if(resp != null){
           //return value is a JSON array
           var objt = eval(resp);
           addToSelect(objt);
         }
       } else {
        //request.status is 503  if the application isn't available;
       //500 if the application has a bug
         alert(
           "A problem occurred with communicating between"+
           " the XMLHttpRequest object and the server program.");
       }
  }//end outer if
} catch (err)   {
      alert("It does not appear that the server "+
```

```
                "is available for this application. Please"+
                " try again very soon. \nError: "+err.message);

 }
}
/* Take an array of string values (obj) and add an option
for each of the values to a select list*/
function addToSelect(obj){
    //contains the US states
    var _select = document.getElementById("sts");
    var el;
    //first remove all options; because the select could include
    //newly added countries from previous clicks
    while(_select.hasChildNodes()){
        for(var i = 0; i < _select.childNodes.length; i++){
            _select.removeChild(_select.firstChild);
        }
    }
    //now add just the original options; 52 states
    for(var h=0; h < origOptions.length;h++) {
       _select.appendChild(origOptions[h]);
    }
    //obj is an array of new options values
    for(var i=0; i < obj.length;i++) {
        el = document.createElement("option");
        el.appendChild(document.createTextNode(obj[i]));
        _select.insertBefore(el,_select.firstChild);
    }
}

/* Create and initialize a Request object; not shown...*/
```

When the browser first loads the web page, the code defines an`onclick` event handler for the `select` list containing the US states. This event will be triggered whenever the user clicks on the `select` widget, whether or not they change the value in the list. The event handler calls a function named `addCountries()`, passing in as a parameter a reference to the`select` object that was clicked.

```
window.onload=function(){
    var sel = document.getElementById("cts");
    var sel2 = document.getElementById("sts");
    if(sel != null){
        sel.onclick=function(){
            addCountries(this)};
    }
    origOptions = new Array();
    //save the original select list of states so that
    //it can be reconstructed with just the original states
    //and the newly added countries
    for(var i = 0; i < sel2.options.length; i++){
        origOptions[i]=sel2.options[i];
```

```
    }
}
```

The code also saves the original contents of the US-states list in an `Array` object. Otherwise, as the user adds countries to the list by clicking the upper `select` list, the same countries would be added to the second `select` list over and over again. Each time the user clicks the top-level `select` list, then, the bottom `select` list will be rebuilt with the new countries added in front of the original list of states. This `origOptions Array` variable caches the original list.

Next up is the `addCountries()` function. We do not need to show this function again because what it accomplishes is fairly simple. The function cycles through the `options` in the continent's `select` list, and if an `option` is checked (the selected `option`), it's value is submitted to a Java servlet. The servlet program returns an `array` of countries associated with the continent, and the code adds those countries to the other `select` list.

NOTE

Our apologies to all those other global citizens who are not represented by these continent choices. For the sake of brevity, we stopped at Europe and South America. A "real-world" (pun intended!) application would represent all of the world's continents except for perhaps Antarctica.

Figure 2-9 shows the web page after the user has chosen South America.

Add countries to the select list without a roundtrip

New Select List, or Mirage?

The code receives the return value in JavaScript Object Notation (JSON) format, as in several other hacks. See Hack #6. The return value takes the form of `["Brazil","Ecuador",etc...]`. The return value is a `string` that is evaluated as a JavaScript `array` using the `eval()` function. In the next step, as if by magic, the new countries appear at the top of the second `select` list. Here is the responsible `addToSelect()` function.

```
function addToSelect(obj){
    //contains the US states
    var _select = document.getElementById("sts");
    var el;
    //first remove all options; because the select could include
    //newly added countries from previous clicks
    while(_select.hasChildNodes()){
        for(var i = 0; i < _select.childNodes.length; i++){
            _select.removeChild(_select.firstChild);
        }
    }
    //now add just the original options; 52 states
    for(var h=0; h < origOptions.length;h++) {
        _select.appendChild(origOptions[h]);
    }
    //obj is an array of new options values
    for(var i=0; i < obj.length;i++) {
        el = document.createElement("option");
        el.appendChild(document.createTextNode(obj[i]));
        _select.insertBefore(el,_select.firstChild);
    }
}
```

This function involves basic DOM API programming, representing a `select` list as a parent node of several `option`-related child nodes. First, the code clears the `select` list and repopulates it with the original states. This is a rule for our application; the user can add new countries on top of the origina list, but not let the countries pile up in the list repetitively. The code then creates a new `option` element for each member of the `array` derived from the server, which is a country name (like "Brazil"). Finally, the code uses the `Node.insertBefore()` method to insert each new `option` before the first `option` in the `select` list.

NOTE

The `_select.firstChild` node keeps changing in the `for` loop. For example, if Alabama is at the top of the list, then `_select.firstChild` returns the `option` node containing the "Alabama" value. Then the loop inserts "Brazil" before "Alabama," and the `option` representing Brazil becomes the `firstChild` node.

Hacking the Hack

Naturally, the next step in this hack is to allow the user to dynamically submit the new country name from the second `select` element. The third hack in this chapter shows you how to add this behavior to a `select` list

---

Previous

Submit Checkbox Values To The Server Without A Browser Refresh

Generate immediate interaction with a server program when the browser user clicks a checkbox.

Checkboxes are those little squares or buttons that allow the user to make a choice among multiple options. The conventional set-up is for the user to check one or more checkboxes as part of a form that they have to submit later. What if you want your application to only submit the checkbox values and have that submission take place when the user clicks the checkbox, not at some indeterminate time later?

This hack represents a poll in which users vote for their favorite team and individual sports. When the browser user selects any of the checkboxes, this action triggers an event that submits this value to a server program then displays the poll results. Figure 2-10 shows what the page looks like in the browser.

Choose your favorite sports

The server program has a database that captures the poll results; the program updates then returns the poll results. This hack uses the `XMLHttpRequest` object to send the sport choices and handle the server's response. The hack uses DOM programming and Cascading Style Sheets to display the poll results. Here is the HTML code for the page.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
     "http://www.w3.org/TR/2000/REC-xhtml1-20000126/DTD/xhtml1-strict.dtd">
<html  xmlns="http://www.w3.org/1999/xhtml"  xml:lang="en"  lang="en">
<head>
      <meta http-equiv="content-type" content="text/html; charset=utf-8" />
     <script type="text/javascript" src="js/hacks2_5.js"></script>
    <link rel="stylesheet" type="text/css" href="/css/hacks2_5.css" />
     <title>submit checkbox values</title>
</head>
<body>
<h3>Choose  your  favorite  sports</h3>
<h4>Team  sport</h4>
<form  id="team"  action="javascript:void%200"  method="get">
<div id="team_d" class="team">
<input type="checkbox" name="team_sports" id=
"baseball" value="baseball" /> Baseball &#x00A;
<input type="checkbox" name="team_sports" id=
"soccer" value="soccer"  /> Soccer  &#x00A;
<input type="checkbox" name="team_sports" id=
"football" value="football"  /> Football  &#x00A;
<input type="checkbox" name="team_sports" id=
"basketball" value="basketball"  /> Basketball  &#x00A;
<input type="checkbox" name="team_sports" id=
"lacrosse" value="lacrosse" />Lacrosse  &#x00A;
<input type="checkbox" name="team_sports" id=
"hockey" value="hockey" /> Hockey  &#x00A;
<input type="checkbox" name="team_sports" id=
"tennis" value="tennis" /> Tennis  &#x00A;
</div>
</form>
<div id="team_poll" class="poll">
    <span id="t_title" class="p_title"></span>
    <span id="t_results" class="p_results"></span></div>
<h4>Individual  sport</h4>
<form  id="ind" action="javascript:void%200" method="get">
<div id="ind_d" class="ind">
<input type="checkbox" name="individual_sports" id=
"cycling" value="cycling" /> Cycling  &#x00A;
<input type="checkbox" name="individual_sports" id=
"running" value="running"  /> Running  &#x00A;
<input type="checkbox" name="individual_sports" id=
"swimming" value="swimming"  /> Swimming &#x00A;
<input type="checkbox" name="individual_sports" id=
"nordic_skiing" value="nordic_skiing"  />Nordic Skiing  &#x00A;
```

```
<input type="checkbox" name="individual_sports" id=
"inline_skating" value="inline_skating"  />Inline Skating &#x00A;
<input type="checkbox" name="individual_sports" id=
"triathlon" value="triathlon"  />Triathlon &#x00A;
<input type="checkbox" name="individual_sports" id=
"track" value="track"  />Track &#x00A;
</div>
</form>
<div id="individual_poll" class="poll">
    <span id="i_title" class="p_title"></span>
    <span id="i_results" class="p_results"></span></div>
</body>
</html>
```

This page first imports the JavaScript code that performs all of the application's work, in a file named hacks2_5.js. An upcoming code sample shows those functions. This HTML also imports a style sheet (hacks2_5.css) for controlling the page's appearance, as well as making the poll results invisible until the user is ready to see them.

The HTML page includes two `div` elements each containing a set of checkbox elements that specify the various team and individual sports. Here is the JavaScript code underlying this hack.

> NOTE
>
> We have omitted the code for creating and initializing the request object, such as the `httpRequest()` function, since so many of the other hacks have already included this code. See Hack #1 or #3 if you need another look!

```
var sportTyp = "";

window.onload=function(){
    var allInputs = document.getElementsByTagName("input");
    if(allInputs != null){
        for(var i = 0; i < allInputs.length;i++) {
            if(allInputs[i].type == "checkbox"){
                allInputs[i].onchange=function(){
                    sendSportsInfo(this)};
            }
        }
    }
}

function sendSportsInfo(obj){
    if (obj == null ) { return; }
    var url = "";
    var nme = "";
    if(obj.checked) {
        nme = obj.name;
        var sub = nme.substring(0,nme.indexOf("_"));
```

```
        sportTyp=sub;
        url = "http://www.parkerriver.com/s/fav_sports?sportType="+nme+
              "&choices="+obj.value;
        httpRequest("GET",url,true);
    }
}

//event handler for XMLHttpRequest
function handleResponse(){
    try{
        if(request.readyState == 4){
            if(request.status == 200){
                var resp =  request.responseText;
                if(resp != null){
                    //return value is a JSON object
                    var func = new Function("return "+resp);
                    displayPollResults(func());
                }
            } else {
                //request.status is 503
                /if the application isn't available;
                //500 if the application has a bug
                alert(
                "A problem occurred with communicating between"+
                " the XMLHttpRequest object and the server program.");
            }
        }//end outer if
    } catch (err)   {
        alert("It does not appear that the server "+
              "is available for this application. Please"+
              " try again very soon. \nError: "+err.message);

    }
}

function displayPollResults(obj){
    var div = document.getElementById(sportTyp+"_poll");
    var spans = div.getElementsByTagName("span");
    for(var i = 0; i < spans.length; i++){
        if(spans[i].id.indexOf("title") != -1){
            spans[i].innerHTML = "<strong>Here are the latest poll "+
            "results for "+sportTyp+
            " sports</strong>"
        }   else {
            //use the object and its properties
            var str ="&#x00A;";
            for(var prop in obj) { str += prop + " : "+obj[prop]+"&#x00A;";}
            spans[i].innerHTML = str;
        }
    }
    div.style.visibility="visible";
}
```

The first task of this code is to assign a function to execute when the checkbox's state changes (from `unchecked` to `checked`). This is the responsibility of the`window.onload` event handler, which the browser calls after the page has been completely loaded.

```
window.onload=function(){
    var allInputs = document.getElementsByTagName("input");
    if(allInputs != null){
        for(var i = 0; i < allInputs.length;i++) {
            if(allInputs[i].type == "checkbox"){
                allInputs[i].onchange=function(){
                    sendSportsInfo(this)};
            }
        }
    }
}
```

The code first stores an `Array` of all the page's `input` elements in an `allInputs` variable. If the `inputs` are of a checkbox type, as in `<input type="checkbox" .../>`, then their `onchange` property refers to a function that calls `sendSportsInfo()`. The code sets all of the checkbox's `onchange` event handlers at once; it will not affect any other `input` elements a page designer adds to the page.

> NOTE
>
> You can call a function in an `onchange` attribute of the `input` element itself. However, web developers using the XHTML DTD discourage the calling of JavaScript functions from element attributes.

Using this as a parameter to `sendSportsInfo()` is a handy mechanism for passing a reference to the exact `input` element whose state has changed.

Let's look at the `sendSportsInfo()` function more closely.

Vote Early and Often

This function constructs a URL or web address in order to send the user's sports choices to a server program.

```
function sendSportsInfo(obj){
    if (obj == null ) { return; }
    var url = "";
    var nme = "";
    if(obj.checked) {
        formObj=obj;
        nme = obj.name;
```

```
        var sub = nme.substring(0,nme.indexOf("_"));
        sportTyp=sub;
        url = "http://www.parkerriver.com/s/fav_sports?sportType="+nme+
              "&choices="+obj.value;
        httpRequest("GET",url,true);
    }
}
```

Since we used the `this` keyword as a parameter to `sendSportsInfo()`, the `obj` variable refers to an HTML `input` element. We are only going to hit the server if the `input` checkbox is selected, so the code checks for that state. The name of each `input` element in the form is set in the HTML to "team_sports" or "individual_sports," so the code captures the name and the name substring preceding the "_" character (we need that for the HTTP response code).

> NOTE
>
> The code `obj.name` accesses the `name` property of an `HTMLInputElement`, which is part of the DOM API. This property refers to the name in the HTML element code, as in `<input name="myname".../>`.

The URL requires the sport type and the value of the checkbox. A typical URL example looks like http://www.parkerriver.com/s/fav_sports?sportType= individual_sports&choices=soccer. The `httpRequest()` method uses the request object to query the server with these values.

Poll Vault

The server will return an HTTP response representing the latest poll results, after it stores the user's vote. The code has designated the `handleResponse()` function for dealing with the response, and calling another function for displaying the results.

```
if(request.readyState == 4){
   if(request.status == 200){
      var resp =  request.responseText;
      if(resp != null){
         //return value is a JSON object
         var func = new Function("return "+resp);
           displayPollResults(func());

       }
}
```

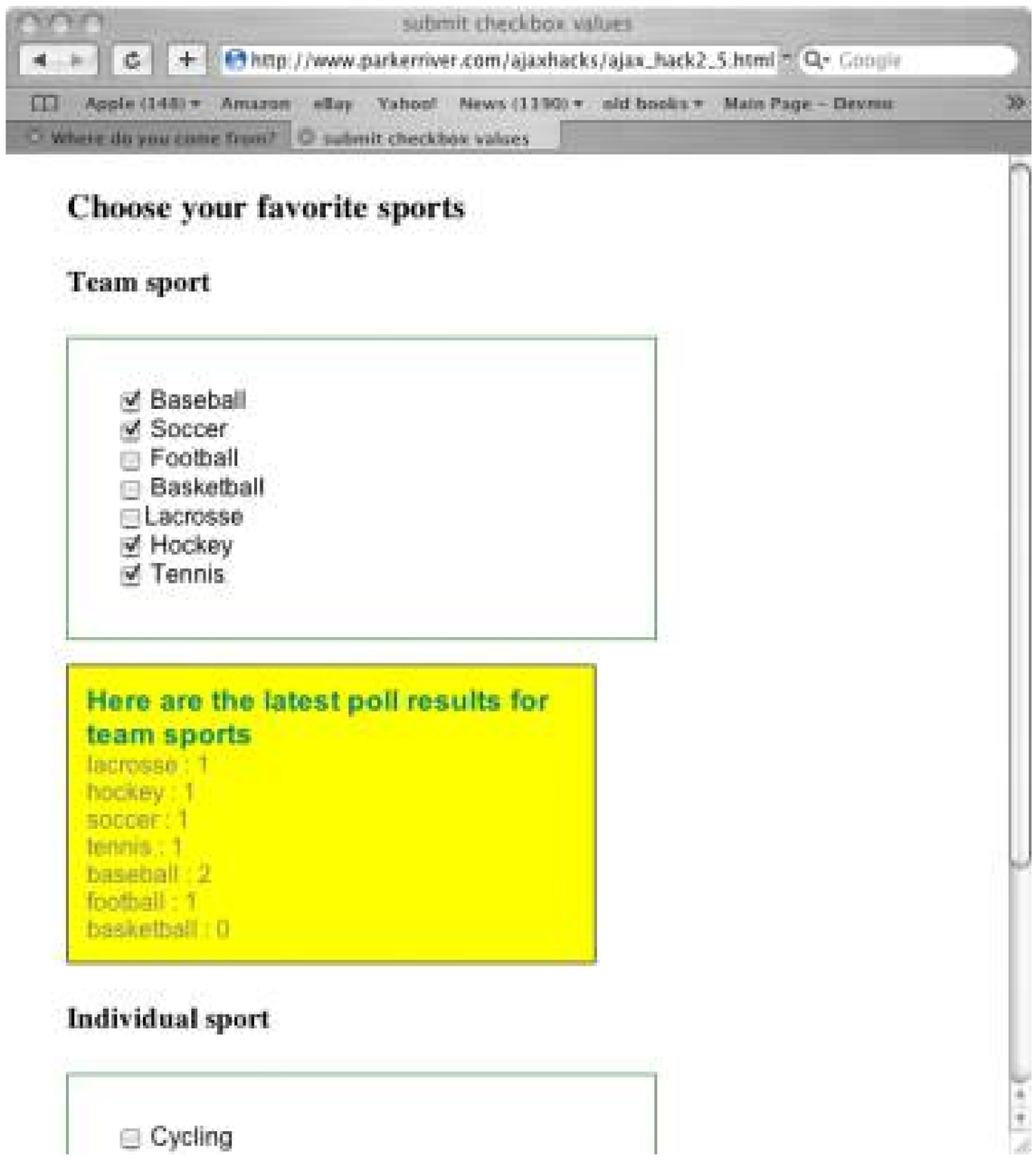
The server returns the result not as XML but in JavaScript Object Notation (JSON) format, a form of plain text that can easily be converted by JavaScript into an object. This is a useful way of enclosing the results. A typical server return value looks like:

```
{
nordic_skiing:  "0",
```

```
inline_skating: "0",
cycling: "2",
track: "2",
swimming: "0",
triathlon: "0",
running: "3"
}
```

The code uses a technique described by Hack #6 to evaluate this text as a JavaScript object. Then the code calls `displayPollResults()`, which as you probably figured out, shows the results in the browser. Figure 2-11 shows what the results look like in Safari:

Figure 2-11: Which sports are favored?



The `displayPollResults()` function uses the DOM to generate a colorful display of the results in the browser.

```
function displayPollResults(obj){
    var div = document.getElementById(sportTyp+"_poll");
    var spans = div.getElementsByTagName("span");
    for(var i = 0; i < spans.length; i++){
```

```
            if(spans[i].id.indexOf("title") != -1){
            spans[i].innerHTML = "<strong>Here are the latest poll results for "+
            sportTyp+" sports</strong>"
          }   else {
              //use the object and its properties
              var str ="&#x00A;";
              for(var prop in obj) { str += prop + " : "+
                obj[prop]+"&#x00A;";}
              spans[i].innerHTML = str;
          }
      }
      div.style.visibility="visible";
}
```

The poll results are displayed inside of`div` elements, which have ids of "team_poll" or "individual_poll." Each one of these`divs` contains two `span` elements; the `span` elements are responsible for the result titles and the actual data. At this point it is helpful to look at the CSS file that specifies various rules for the appearance of our poll results. The`divs` and their contents are initially hidden, with the`visibility` CSS property, until the user clicks a checkbox.

```
.p_title {font-size: 1.2em; color: teal }
h3 { margin-left: 5%; font-size: 1.4em; }
h4 { margin-left: 5%; font-size: 1.2em; }
div.poll { margin-left: 5%; visibility: hidden; border: thin solid black;
     padding: 2%; font-family: Arial, serif;
     color: gray; background-color: yellow}

div.team { margin-left: 5%; border: thin solid green; padding: 5%;
     font-family: Arial, serif}

div.ind { margin-left: 5%; border: thin solid green; padding: 5%;
     font-family: Arial, serif }

div { max-width: 50% }
```

One of the cool aspects of DOM and Ajax mechanisms is that CSS properties are programmable too. When the page view is ready to show the poll results, the`visibility` property of the `divs` that hold these results is set to "visible." This is accomplished with the code`div.style.visibility = "visible."`

In the `displayPollResults()` function, the code sets the `innerHTML` property for the `span` elements responsible for displaying a title about the poll results. In addition, the poll results derived from the server are stored in a `string` and displayed in this manner.

```
var str ="&#x00A;";
for(var prop in obj) { str += prop + " : "+
    obj[prop]+"&#x00A;";}
   spans[i].innerHTML = str;
```

The `obj` variable is a JavaScript object. The `for(property in object)` expression then generates a `string` that looks like `
baseball : 2
soccer : 3...`

If you keep clicking on checkboxes, you can watch the votes increment without anything else changing in the browser. This is a useful design for the applications that collect discrete feedback from users and instantaneously display the results.

Dynamically Generate A New Checkbox Group With Server Data

Let a web page's checkbox content evolve from a user's interaction with an application.

Most web forms are static, meaning the text labels and entry widgets like textareas, checkboxes, and radio buttons are hard-coded into the HTML. Lots of applications however can benefit from the ability to whip together form elements on the fly, based on a user's preferences. The content for the forms, if necessary, can even be derived from a server, such as questions for various types of quizzes and polls.

Hey, hack #14 showed how to do this with a`select` list widget, so why don't we auto-generate a bunch of checkboxes?

This hack gives the user a choice of "team sports" or "individual sports" in two radio buttons, then, when they click either button, grabs the sports categories from a server component and creates a new group of checkboxes.

Choose Your Activity

Figure 2-12 shows our barebones web page to begin with, before the DOM magic starts!

Let the form evolution begin

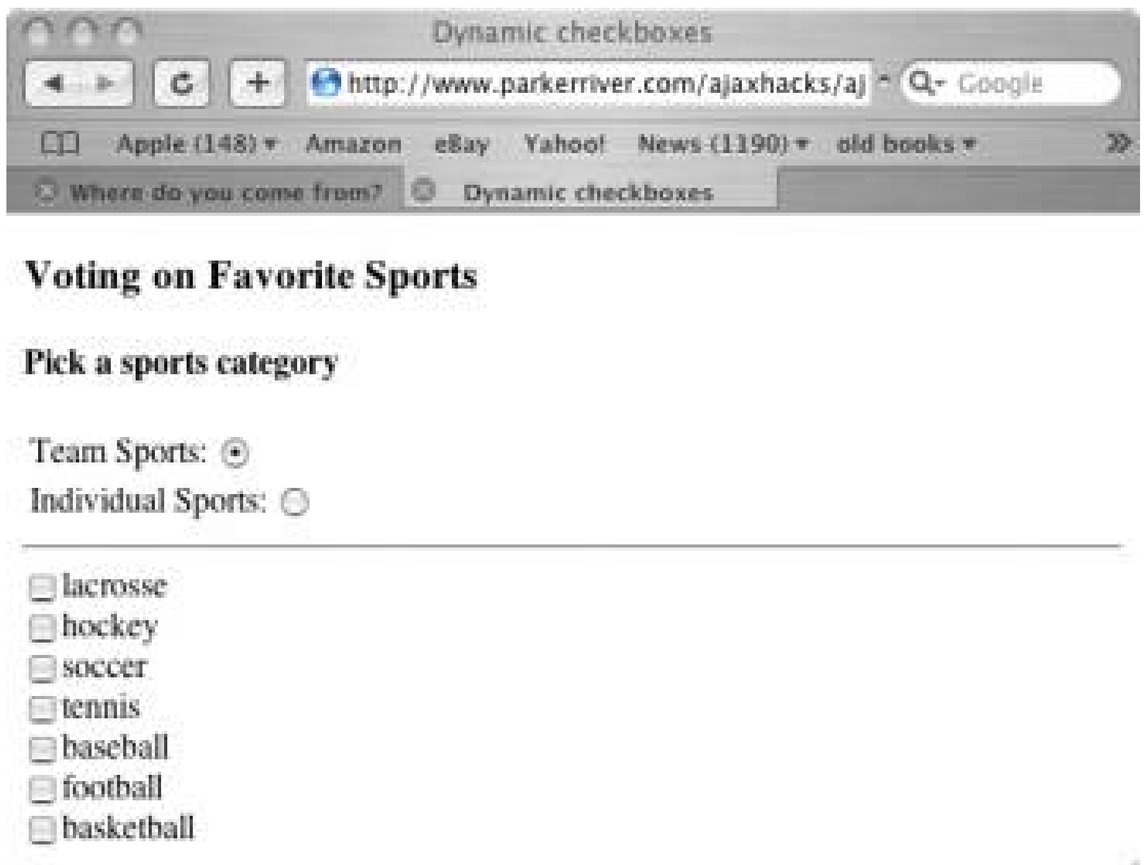
Here is the HTML for the form. The dynamic behavior for this page is all contained in the JavaScript file hacks2_7.js. The two radio buttons that the users may click to get things going are represented by the two `input` elements, and the newly generated checkboxes will appear within the `div` element with the `id` "checks."

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/2000/REC-xhtml1-20000126/DTD/xhtml1-strict.dtd">
<html  xmlns="http://www.w3.org/1999/xhtml"  xml:lang="en"  lang="en">
<head>
     <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <script type="text/javascript" src="js/hacks2_7.js"></script>
    <title>Dynamic checkboxes</title>
</head>
<body>
<h3>Voting on Favorite Sports</h3>
<h4>Pick a sports category</h4>
<form  action="javascript:void%200">
    <table border="0">
        <tr><td>
        Team Sports:
**<input type="radio" name="_sports" value="team" />**
</td></tr>
      <tr><td>  Individual Sports:
**<input type="radio" name="_sports" value="individual" />**
</td></tr>
</table>
    <hr />
<div id="checks"></div>
</form>
</body>
</html>
```

When the user clicks a checkbox, the page instantly displays either of two different sets of new checkboxes representing either individual sports or team sports. The actual lists of sports that make up the checkboxes are arrays of strings that the server returns. They obviously could be hard-coded into the JavaScript in order to prevent a network hit, but imagine that the checkbox widgets represent values that are frequently changing and/or must be derived from persistent storage on the server, such as complex multiple-choice questions in a questionnaire or product information?

Figure 2-13 shows the web page after the user has clicked a radio button. This action only submits the value associated with the checkbox that the user clicked, not the entire form.

Widgets spawning other widgets



Okay, where's the code?

Here is the JavaScript contained in the file hacks2_7.js. We have omitted the code that creates and initializes the request object, which you can review in Hack #1 and several other earlier hacks. The first thing you may notice in the code is that it assigns a function to handle the radio button's `onclick` event handler. These events are triggered by the user clicking either radio button.

> NOTE
>
> An "event handler" such as `onclick` or `onchange` is an attribute of an HTML element that can be assigned the code that will be executed whenever the user clicks that element, for example.

This assignment begins in the window's `onload` event handler. This event takes place when all of the elements in the HTML page have been loaded by the browser.

```
var sportType="";
window.onload=function(){
    var rads = document.getElementsByTagName("input");
    if(rads != null) {
        for(var i = 0; i < rads.length; i++) {
            if(rads[i].type=="radio"){ rads[i].onclick=function(){
                getSports(this)};}

        }
    }
}
```

```
function getSports(obj){
 if (obj == null ) { return; }
    var url = "";
    var val = "";
    if(obj.checked) {
        val=obj.value;
        sportType=val;
        url = "http://www.parkerriver.com/s/fav_sports"+
              "?sportType="+encodeURIComponent(val)+"&col=y";
        httpRequest("GET",url,true);
    }
}

//event handler for XMLHttpRequest
function handleResponse(){
    try{
        if(request.readyState == 4){
            if(request.status == 200){
                var resp =  request.responseText;
                if(resp != null){
                    //return value is a JSON array
                    var objt = eval(resp);
                    createChecks(objt);
                }
            } else {
                //request.status is 503
                //if the application isn't available;
                //500 if the application has a bug
                alert(
                        "A problem occurred with communicating between"+
                        " the XMLHttpRequest object and the server program.");
            }
        }//end outer if
    } catch (err)   {
        alert("It does not appear that the server "+
              "is available for this application. Please"+
              " try again very soon. \nError: "+err.message);

    }
}

function createChecks(obj){
    var _div = document.getElementById("checks");
    var el;
    //first remove all existing checkboxes
    while(_div.hasChildNodes()){
        for(var i = 0; i < _div.childNodes.length; i++){
            _div.removeChild(_div.firstChild);
        }
    }
    //obj is an array of new sports names
    for(var i=0; i < obj.length;i++) {
```

```
            el = document.createElement("input");
            el.setAttribute("type","checkbox");
            el.setAttribute("name",sportType);
            el.setAttribute("value",obj[i]);
            _div.appendChild(el);
            _div.appendChild(document.createTextNode(obj[i]));
            _div.appendChild(document.createElement("br"));
        }
}
/* httpRequest() and related code omitted for the sake of brevity... */
```

The first stage in generating the checkboxes is to send the request that fetches the values for each o these widgets. When the user clicks a radio button, the code calls`getSports()`. This function formats a URL based on the value it receives from the checkbox, then sends a request to a server component for a list of related sports.

Greetings JSON

The response comes back from the server in a`string` formatted as JavaScript Object Notation (JSON). A response might look like "["football","soccer","tennis", etc.]". We get the response from the request object's`responseText` property, then convert the response to a JavaScript`array` using the`eval()` global function. Phew, that was a mouthful! If that wasn't clear, then see the discussion o handling JSON server values in Hack #6.

> NOTE
>
> Ajax developers often advocate JSON as the format of the server return value, in the many cases where XML might be overkill on both the server and client side of things.

Once the code has this`array` of values from the server, then the code passes the`array` along to`createChecks()`. This function uses the DOM API to create the checkboxes, one checkbox for each value in the`array` (a checkbox for tennis, another for soccer, and so on). Here is the code for this function.

```
function createChecks(obj){
    var _div = document.getElementById("checks");
    var el;
    //first remove all existing checkboxes
    while(_div.hasChildNodes()){
        for(var i = 0; i < _div.childNodes.length; i++){
            _div.removeChild(_div.firstChild);
        }
    }
    //obj is an array of new sports names
    for(var i=0; i < obj.length;i++) {
        el = document.createElement("input");
        el.setAttribute("type","checkbox");
        el.setAttribute("name",sportType);
```

```
        el.setAttribute("value",obj[i]);
        _div.appendChild(el);
        _div.appendChild(document.createTextNode(obj[i]));
        _div.appendChild(document.createElement("br"));
    }
```

The function gets a reference to the `div` element on the HTML page that will enclose the checkboxes. Then the code removes any existing checkboxes, because of it didn't, then the user could keep checking the radio buttons and generate several duplicate checkboxes appended on the end of the web page; this is an outcome we want to avoid. Finally, the code creates a new `input` element for each sport, so that each of these widgets looks like:

```
<input type="checkbox" name=
"teams_sports" value="baseball" /> baseball&#x00A;
```

As soon as this function completes executing, the checkboxes appear on the web page without any visible refresh. Like magic!

Hacking the Hack

Naturally, you want the user to check these generated checkboxes for some purpose. Maybe to generate another sub-set of widgets or checkboxes? Or to send the values from the new checkboxes, when the user clicks them, to a server component? You can adapt the code from Submit Checkbox Values To The Server Without A Browser Refresh to accomplish the latter task, as well as create `onclick` event handlers for the new checkboxes (as in this hack) to give them some behavior.

Populate An Existing Checkbox Group From The Server

Dynamically add widgets to an existing group of checkboxes.

This is another type of adaptive web form, where a group of widgets can change based on the preferences of the user that accesses the web page. In Hack #16 the code submitted a clicked checkbox value right away to a server program. This hack allows users to add new choices to the same bunch of checkboxes before they choose among those widgets. The web page has a`select` list including the choices "team" or "individual." It shows two groups of checkboxes representing team sports and individual sports. Choosing either "team" or "individual" in the popup button or`select` list expands the existing checkboxes by getting new content from a server. Clicking the "restore" popup button restores the original checkboxes for either the team or individual list.

Figure 2-14 shows the web page before the user makes a choice.

Expand the offerings

Figure 2-15 depicts the same page after the user chooses "team" from the popup button at the top o the page, thus expanding the choices of team sports.

Team offerings expanded

How does it work?

We are assuming that the content for the new checkboxes must come from the server, because it changes often and/or derives from the organization's database. Otherwise, an application like this can just include a JavaScript `Array` of new content and never touch the server program. When the user makes a choice from the first popup menu or `select` list, this action sends the choice of team or individual to a server program. The code uses the request object to connect with the server the Ajax way.

The server replies with an `array` of titles for new checkboxes. Flipping the choices in the `select` list launches an `onclick` event handler in the JavaScript code, which the upcoming code sample shows.

We will not take up space with the HTML code, because the page is almost exactly the same as Hack #17. The page uses a `script` tag to import all of its Ajax-related JavaScript in a file named hacks2_8.js. You can read through the code comments right now to get a feel for what the code does.

NOTE

A single-line JavaScript comment begins with "//." A multi-line comment is bracketed by "/* */."

```
var sportTyp = "";
var checksArray = null;

window.onload=function(){
    //the 'expanding checkboxes' select popup
    var sel = document.getElementById("expand");
    //bind onclick event handler to a function
    if(sel != null){
        sel.onclick=function(){
            getMoreChoices(this)};
    }
    //the 'restoring checkboxes' select popup
    var selr = document.getElementById("restore");
    //bind onclick event handler to the function
    if(selr != null){
        selr.onclick=function(){
            restore(this)};
    }
    //Place all existing checkbox elements in two arrays
    //for restoring the original checkbox lists
    checksArray = new Object();
    checksArray.team = new Array();
    checksArray.individual = new Array();
    var ckArr = document.getElementsByTagName("input");
    populateArray(ckArr,"team");
    populateArray(ckArr,"individual");
}

function populateArray(arr,typ)  {
    var inc = 0;
    for(var i = 0; i < arr.length; i++){
        if(arr[i].type == "checkbox") {
            if(arr[i].name.indexOf(typ) != -1) {
                checksArray[typ][inc] = arr[i];
                inc++;
            }
        }
    }
}

//Return the number of input checkbox elements contained
//by a div element
function getCheckboxesLength(_sportTyp){
      var div = document.getElementById(_sportTyp+"_d");
      var len=0;
      for(var i =0; i < div.childNodes.length; i++){
         if(div.childNodes[i].nodeName == "INPUT" ||
                div.childNodes[i].nodeName == "input" ){
```

```
                len++;
            }
        }
        return len;
}
/* Use the request object to fetch an array of
titles for new checkboxes.
The obj parameter represents a select element; get the
value of this element, then hit the server with this value
to request the new titles, but only if the
checkbox hasn't already been expanded */
function getMoreChoices(obj){
    if (obj == null ) { return; }
    var url = "";
    var optsArray = obj.options;
    var val = "";
    for(var i=0; i < optsArray.length; i++){
        if(optsArray[i].selected) {
            val=optsArray[i].value;  break;
        }
    }
    sportTyp=val;
    //determine whether the checkboxes have already been expanded
    if(checksArray[sportTyp].length < getCheckboxesLength(sportTyp))  {
        return;
    }
    url = "http://www.parkerriver.com/s/expand?expType="+val;
    httpRequest("GET",url,true);
}
/* Add new checkboxes to either of the original checkbox lists.
 Only add the new checkboxes if the list hasn't been expanded yet.
 Just return from this function and don't hit the network
 if the list has already been expanded.
Parameter obj An array of new titles like
 ["Field Hockey", "Rugby"]
*/
function addToChecks(obj){
    //div element that contains the checkboxes
    var div = document.getElementById(sportTyp+"_d");
    var el = null;
    //Now add the new checkboxes derived from the server
    for(var h = 0; h < obj.length; h++){
        el = document.createElement("input");
        el.type="checkbox";
        el.name=sportTyp+"_sports";
        el.value=obj[h];
        div.appendChild(el);
        div.appendChild(document.createTextNode(obj[h]));
        div.appendChild(document.createElement("br"));
    }
}
//Restore the original list of checkboxes, using
```

```
//the checksArray object containing the
//original checkboxes.
function restore(_sel) {
    var val;
    var opts =  _sel.options;
    for (var i = 0; i < opts.length; i++){
        if(opts[i].selected) { val=opts[i].value; break;}
    }
    //Only restore if the checkboxes have
    //already been expanded
    if(checksArray[sportTyp].length < getCheckboxesLength(sportTyp)) {
        var _div = document.getElementById(val+"_d");
        if(_div != null)  {
            //rebuild the list of original checkboxes
            _div.innerHTML="";
            var tmpArr = checksArray[val];
            for(var j = 0; j < tmpArr.length; j++){
                _div.appendChild(tmpArr[j]);
                _div.appendChild(document.createTextNode(tmpArr[j].value));
                _div.appendChild(document.createElement("br"));
            }
        }
    }
}
//event handler for XMLHttpRequest
function handleResponse(){
    try{
        if(request.readyState == 4){
            if(request.status == 200){
                var resp =  request.responseText;
                if(resp != null){
                    //return value is a JSON object
                    addToChecks(eval(resp));
                }
            } else {

/* Create and initialize a Request object; not included.
See this chapter's first hack*/
```

Most of the code is involved with capturing and restoring the checkbox's original state, and the comments contained in the latter code sample should make it clear what this code is accomplishing. The reason why we have to include this code is to prevent the behavior of clicking the popup button continuously to append the same new checkboxes at the end of the list.

The code checks whether the list has already been expanded, by comparing the number of checkboxes in the cached array with the existing checkbox group. If the existing group has more checkboxes than the original group, then the list has already been expanded. If the user tries to expand the list twice, the second click will be ignored, thus sparing the network from a needless hit.

Ajax Stuff

`getMoreChoices()` makes a server request using the request object to acquire titles for new checkboxes. See Hack #1 or #3 if you have not been introduced to the request object. The first `select` list's `onclick` event handler, which is set up when the browser window first loads the web page (`window.onload`), launches this function, passing in a reference to the `select` element.

The `select` element in our page can only have the values "team" or "individual." Then the code appends the value ("team" or "individual") on to the end of the URL reflecting the server program. The `httpRequest()` function sets up and launches the request (see Hack #1 or the first hack in this chapter).

```
function getMoreChoices(obj){
    if (obj == null ) { return; }
    var url = "";
    var optsArray = obj.options;
    var val = "";
     for(var i=0; i < optsArray.length; i++){
        if(optsArray[i].selected) {
            val=optsArray[i].value;  break;
        }
    }
    sportTyp=val;
    //determine whether the checkboxes have already been expanded
    if(checksArray[sportTyp].length < getCheckboxesLength(sportTyp))  {
        return;
    }
    url = "http://www.parkerriver.com/s/expand?expType="+val;
    httpRequest("GET",url,true);
}
```

Here Comes JSON

The server sends back the HTTP response in JavaScript Object Notation (JSON) format. See Hack #6 for more details on this technique.

```
if(request.readyState == 4){
 if(request.status == 200){
     var resp =  request.responseText;
      if(resp != null){
        //return value is a JSON array
        addToChecks(eval(resp));
      }
 } else {...}
```

The return value represented by the variable `resp` is a `string` such as `["Field Hocket","rugby"]`. The code passes this string to the `eval()` global function, which returns a JavaScript `array`. The `addToChecks()` function then creates new checkboxes from this `array`.

```
function addToChecks(obj){
    //div element that contains the checkboxes
    var div = document.getElementById(sportTyp+"_d");
    var el = null;
    //Now add the new checkboxes derived from the server
    for(var h = 0; h < obj.length; h++){
        el = document.createElement("input");
        el.type="checkbox";
        el.name=sportTyp+"_sports";
        el.value=obj[h];
        div.appendChild(el);
        div.appendChild(document.createTextNode(obj[h]));
        div.appendChild(document.createElement("br"));
    }
}
```

This function uses the DOM API to create new `input` elements and add them to the end of the `div` element containing the checkboxes. the effect on the user is that they see the checkbox list grow, bu nothing else changes on the page. Nifty!

You may want to take a look at the `restore()` function, which takes an expanded checkbox list and restores it to its original content, without any network hits.

Change Unordered List Labels Using An HTTP Response

Change static unordered lists based on content derived from a server.

One of the most common tags found on web pages is the unordered list, which browsers usually render as a list of bullets accompanied by labels. This hack allows the web-page user to change an unordered list by adding additional items to it. The content for the items derives from a server program. This hack is very similar to the previous one, in which the user could add items to two lists of checkboxes. The main difference is that this hack deals with unordered lists, which are designed to display information rather than provide a selection widget (like a checkbox).

The code is also different for creating list items as opposed to checkboxes.

NOTE

Go ahead and skip this hack if you are not interested in playing with unordered lists, because the code is a revised version of the previous hack.

Figure 2-16 shows this hack's web page before the user chooses to expand either of two lists. The lists involve team sports and individual sports. When the user clicks the pop-up button at the top of the page, the list grows by a few items without anything else on the page changing. Just like the last hack, the lists can be restored to their original content by clicking the second popup button. The speed with which the lists grow and shrink is quite impressive, particularly considering that the "growth" content comes from a server.

Watch the list grow and shrink

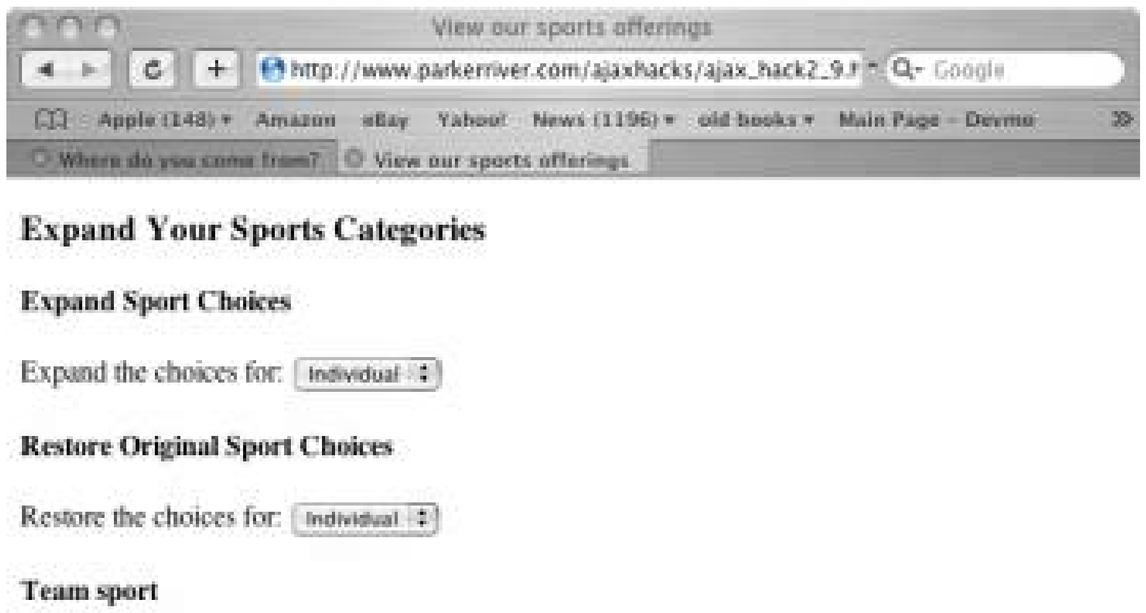


Figure 2-17 shows the web page after the user has expanded the team sports.

Expanding the menu of team sports

Here's the code for the Web page.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/2000/REC-xhtml1-20000126/DTD/xhtml1-strict.dtd">
<html  xmlns="http://www.w3.org/1999/xhtml"  xml:lang="en"  lang="en">
<head>
     <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <script type="text/javascript" src="js/request_init.js"></script>
    <script type="text/javascript" src="js/hacks2_9.js"></script>
    <title>View our sports offerings</title>
</head>
<body>
<h3>Expand  Your  Sports  Categories</h3>
<h4>Expand  Sport  Choices</h4>
<form  action="javascript:void%200">
    <div id="exp">
        Expand the choices for:
    <select name="_expand" id="expand">
        <option value="individual">Individual</option>
        <option value="team">Team</option>
    </select>
    </div>
</form>
<h4>Restore  Original  Sport  Choices</h4>
```

```
<form  action="javascript:void%200">
    <div id="rest">
        Restore the choices for: <select name=
        "_restore" id="restore">
        <option value="individual">Individual</option>
        <option value="team">Team</option>
    </select>
    </div>
</form>
<h4>Team  sport</h4>
        <ul id="team_u">
            <li>Baseball </li>
            <li>Soccer</li>
            <li>Football  </li>
            <li> Basketball  </li>
            <li>Lacrosse </li>
            <li> Hockey</li>
            <li>Tennis </li>
        </ul>
<h4>Individual  sport</h4>
        <ul id="individual_u">
            <li>Cycling</li>
            <li>Running</li>
            <li>Swimming</li>
            <li>Nordic Skiing</li>
            <li>Inline Skating</li>
            <li>Triathlon</li>
            <li>Track</li>
        </ul>
</body>
</html>
```

The key to this code is giving the `ul` tags their own `id` values. The JavaScript code can then access the tag as in the following example.

```
var ul = document.getElementById(sportTyp+"_u");
```

The `ul` elements contain the `list` items; therefore, the code increases or restores the `list` items by appending child nodes or elements to the `ul` elements. Of course, in this hack, the content for the new `list` items derives from a server. As a result, the code has to use the request object to first fetch the new values.

The web page imports two JavaScript files, request_init.js and hacks2_9.js. The first file creates and sets up the `XMLHttpRequest` object. See Hack #6 for a description of a JavaScript file that manages the request object. hacks2_9.js contains the code that grows and restores the unordered lists.

```
var sportTyp = "";
var itemsArray = null;
//define Object for caching li items
//this is a workaround for IE 6, which
```

```
//doesn't save the li element's text node
//or label when you cache it
function CachedLiItem(liElement,liLabel){
    //an li element object
     this.liElement=liElement;
    //a string representing the li text node or label
     this.liLabel=liLabel;
}
window.onload=function(){
    var sel = document.getElementById("expand");
    //bind onclick event handler to a function
    if(sel != null){
        sel.onclick=function(){
            getMoreChoices(this)};
    }
    var selr = document.getElementById("restore");
    //bind onclick event handler to a function
    if(selr != null){
        selr.onclick=function(){
            restore(this)};
    }
    //Place all existing bullet items in two arrays
    //for restoring later
    itemsArray = new Object();
    itemsArray.team = new Array();
    itemsArray.individual = new Array();
    var bulletArr = document.getElementsByTagName("li");
    populateArray(bulletArr,"team");
    populateArray(bulletArr,"individual");
}

//Create Arrays of CachedLiItem objects for
//restoring the unordered lists later
function populateArray(arr,typ)  {
    var inc = 0;
    var el = null;
    var liObj=null;
    for(var i = 0; i < arr.length; i++){
        el = arr[i].parentNode;
        if(el.id.indexOf(typ) != -1) {
            liObj=new CachedLiItem(arr[i],arr[i].childNodes[0].nodeValue);
            itemsArray[typ][inc] = liObj;
            inc++;
        }
    }
}
//Return the number of li elements contained
//by a ul element
function getULoptionsLength(_sportTyp){
    var ul = document.getElementById(_sportTyp+"_u");
    var len=0;
     for(var i =0; i < ul.childNodes.length; i++){
```

```
            if(ul.childNodes[i].nodeName == "LI" ||
               ul.childNodes[i].nodeName == "li" ){
                len++;
            }
        }
        return len;
    }
    function getMoreChoices(obj){
        if (obj == null ) { return; }
        var url = "";
        var optsArray = obj.options;
        var val = "";
         for(var i=0; i < optsArray.length; i++){
            if(optsArray[i].selected) {
                val=optsArray[i].value;  break;
            }
        }
        sportTyp=val;
        //determine whether the bullets have already been expanded
        if(itemsArray[sportTyp].length < getULoptionsLength(sportTyp))  {
            return;
        }
        url = "http://www.parkerriver.com/s/expand?expType="+val;
        httpRequest("GET",url,true);
    }
    function addToBullets(obj){
        //ul element that contains the bullet items
        var ul = document.getElementById(sportTyp+"_u");
        var el = null;
        //Now add the new items derived from the server
        for(var h = 0; h < obj.length; h++){
            el = document.createElement("li");
            el.appendChild(document.createTextNode(obj[h]));
            ul.appendChild(el);
        }
    }

    function restore(_sel) {
     var val;
        var opts =  _sel.options;

        for (var i = 0; i < opts.length; i++){
            if(opts[i].selected) { val=opts[i].value; break;}
        }
        sportTyp=val;
        //Only restore the lists if the bullets have
        //already been expanded
        if(itemsArray[sportTyp].length < getULoptionsLength(sportTyp)) {
            var ul = document.getElementById(val+"_u");
            if(ul != null)  {
                //rebuild the list of original bullets
                ul.innerHTML="";
```

```
                var tmpArr = itemsArray[val];
                var tmpLiElement = null;
                for(var j = 0; j < tmpArr.length; j++){
                    tmpLiElement=tmpArr[j].liElement;
                    //workaround for IE6
                    if(tmpLiElement.hasChildNodes()){tmpLiElement.
                            removeChild(tmpLiElement.firstChild);}
                    tmpLiElement.appendChild(document.
                            createTextNode(tmpArr[j].liLabel))
                    ul.appendChild(tmpLiElement);
                }
            }
        }
    }
    //event handler for XMLHttpRequest
    function handleResponse(){
        try{
            if(request.readyState == 4){
                if(request.status == 200){
                    var resp =  request.responseText;
                    if(resp != null){
                        //return value is a JSON array
                        addToBullets(eval(resp));
                    }
                } else {
                    //snipped for the sake of brevity
                }
            }//end outer if
        } catch (err)   {
            alert("It does not appear that the server "+
                  "is available for this application. Please"+
                  " try again very soon. \nError: "+err.message);

        }
    }
```

The `populateArray()` and `getMoreChoices()` functions are almost exactly the same as in the previous hack's code, so we will not take up space here explaining them in detail. The former functio caches the original unordered list in an array, so that it can be restored later.`getMoreChoices()` hits the server for more sport types using the request object, but only if the unordered list has not yet been expanded.

Next, the code gets the server's return value so that the code can grow either the team sport or individual-sport list.

```
var resp =  request.responseText;
if(resp != null){
//return value is a JSON array
addToBullets(eval(resp));
}
```

The return value is a `string` in `array` syntax, as in `"["Field Hockey","Rugby"]."` The code uses the `eval()` global function to convert the `string` to a JavaScript `array`. See Hack #6. Then it passes this `array` to `addToBullets()`.

```
function addToBullets(obj){
    //ul element that contains the bullet items
    var ul = document.getElementById(sportTyp+"_u");
    var el = null;
    //Now add the new items derived from the server
    for(var h = 0; h < obj.length; h++){
        el = document.createElement("li");
        el.appendChild(document.createTextNode(obj[h]));
        ul.appendChild(el);
    }
}
```

This function initiates some DOM programming to create new `list` items and append them as children of a `ul` tag. The existing `ul` tag has an `id` like "team_u." The code uses `document.getElementById(sportTyp+&_u&)` to get a reference to the `ul` tag, then appends a new `li` element to the `ul` for each value in the `array`.

`restore()` comes into play if the user wants to restore the original lists.

```
//Only restore the lists if the bullets have
//already been expanded
    if(itemsArray[sportTyp].length < getULoptionsLength(sportTyp)) {
        var ul = document.getElementById(val+"_u");
        if(ul != null)  {
            //rebuild the list of original bullets
            ul.innerHTML="";
            var tmpArr = itemsArray[val];
            var tmpLiElement = null;
            for(var j = 0; j < tmpArr.length; j++){
                tmpLiElement=tmpArr[j].liElement;
                //workaround for IE6
                if(tmpLiElement.hasChildNodes()){tmpLiElement.
                        removeChild(tmpLiElement.firstChild);}
                tmpLiElement.appendChild(document.
                        createTextNode(tmpArr[j].liLabel))
                ul.appendChild(tmpLiElement);
            }
```

This code uses a cache of original list items to rebuild the restored unordered list. When the web pag loads, the code uses a simple JavaScript object to represent each `li` element.

```
function CachedLiItem(liElement,liLabel){
    //an li element object
     this.liElement=liElement;
    //a string representing the li text node or label
     this.liLabel=liLabel;
```

```
}
```

The object has two properties: the `li` element itself, and the `string` that specifies its label (the text that you see next to the bullet). When you cache an `li` element in an `array`, for instance, Internet Explorer 6 will not save the `li` element's internal text node, so we are using this workaround object. The code empties the `ul` element first by setting its `innerHTML` property to the empty `string`. Then the code uses `appendChild()` from the DOM API to embed the original list items within this `ul` parent element.

Parting Shots

Your application never has to hit the network if it has a well-defined list of items that can just be hard-coded into the client-side JavaScript as `arrays`. But if the task calls for expanding web lists from server databases, and this persistent information changes often, then this hack's approach can come through for the developers.

Previous

Dynamically Generate An Unordered List From The Server

Show bulleted lists on the web page dynamically without any other part of the page changing.

This hack shows two radio buttons to the browser user. Each button represents a sports category. When they click on either button, the code fetches a list of titles from a server and displays a new unordered list. An unordered list looks like:

- Here is one item.
- Here is another item.

Most web pages, if they include a list of bullets, embed them statically inside the page. If the page developers want to change the content of the list, then they have to manually alter the page. This hack changes each list depending on the preferences of the user, which derive from which radio button they choose to click. In fact, to begin with, the page does not have any visible lists. Figure 2-1 shows what the web page looks like.

Generate your own list

When the user clicks either one of the radio buttons, Figure 2-19 shows an example of what they see

Get involved in a team sport

Choose Your Sports Category

Team Sports:
Individual Sports:

- nordic_skiing
- inline_skating
- cycling
- track
- swimming
- triathlon
- running

This hack uses the request object to query a server for a list of sports, so that only the bulleted list changes. Everything else on this barebones page stays the same. Here is the code for the web page. It imports two JavaScript files named http_request.js and hacks2_10.js.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/2000/REC-xhtml1-20000126/DTD/xhtml1-strict.dtd">
<html  xmlns="http://www.w3.org/1999/xhtml"  xml:lang="en"  lang="en">
<head>
     <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <script type="text/javascript" src="js/http_request.js"></script>
    <script type="text/javascript" src="js/hacks2_10.js"></script>
    <title>View our sports offerings</title>
</head>
<body>
<h3>View Our Offerings For Each Sports Category</h3>
<form  action="javascript:void%200">
    <table border="0">
        <tr><td>
            Team Sports:
**<input type="radio" name="_sports" value="team" />**</td></tr>
        <tr><td>  Individual Sports:
**<input type="radio" name="_sports" value="individual" />**
        </td></tr>
    </table>
    <div id="bullets"></div>
</form>
</body>
</html>
```

The http_request.js file includes the code for creating and setting up the request object for connecting with a server. The other JavaScript file is responsible for responding to clicks on the radio buttons, then generating the unordered list based on the content received from the server.

NOTE

Hack #3 shows the http_request.js code, so we will not duplicate that code here.

Here's the code in the hacks2_10.js file.

```
window.onload=function(){
    var rads = document.getElementsByTagName("input");
    if(rads != null) {
        for(var i = 0; i < rads.length; i++) {
            if(rads[i].type=="radio"){ rads[i].onclick=function(){
                getSports(this)};}
        }
    }
}

function getSports(obj){
    if (obj == null ) { return; }
    var url = "";
    var val = "";
    if(obj.checked) {
        val=obj.value;
        url = "http://www.parkerriver.com/s/fav_sports"+
              "?sportType="+encodeURIComponent(val)+"&col=y";
        httpRequest("GET",url,true,handleResponse);
    }
}

//event handler for XMLHttpRequest
function handleResponse(){
    try{
        if(request.readyState == 4){
            if(request.status == 200){
                var resp =  request.responseText;
                if(resp != null){
                    //return value is a JSON array
                    var objt = eval(resp);
                    createBullets(objt);
                }
            } else {
            //request.status is 503  if the application isn't available;
            //500 if the application has a bug
            alert(
                "A problem occurred with communicating between"+
                " the XMLHttpRequest object and the server program.");
            }
    }//end outer if
} catch (err)   {
    alert("It does not appear that the server "+
          "is available for this application. Please"+
```

```
                " try again very soon. \nError: "+err.message);

    }
}

function createBullets(obj){
    var _div = document.getElementById("bullets");
    var innerHTMLstr =  "<ul>";
    //obj is an array of new sports names
    for(var i=0; i < obj.length;i++) {
        innerHTMLstr += "<li>";
        innerHTMLstr += obj[i];
        innerHTMLstr += "</li>";
    }
    innerHTMLstr += "</ul>";
    _div.innerHTML= innerHTMLstr;
}
```

When the browser first loads the web page, the`window.onload` event handler is triggered.

> NOTE
>
> An event handler is code that responds to an "event" like a browser completing web-page loading or a user clicking on a radio button. You use event handlers to point to functions or blocks of code that execute whenever these events occur, in order to give your application the desired behavior. Event handling code allows the code to generate bulleted lists in response to the user choosing a radio button.

This event handler points to a function that in turns calls`getSports()`. The latter function is ultimately responsible for requesting from the server the content for all of the new bullet items. Here are both functions so that we can take a closer look.

```
window.onload=function(){
    var rads = document.getElementsByTagName("input");
    if(rads != null) {
        for(var i = 0; i < rads.length; i++) {
            if(rads[i].type=="radio"){ rads[i].onclick=function(){
                getSports(this)};}
        }
    }
}

function getSports(obj){
    if (obj == null ) { return; }
    var url = "";
    var val = "";
    if(obj.checked) {
        val=obj.value;
        sportType=val;
```

```
        url = "http://www.parkerriver.com/s/fav_sports"+
              "?sportType="+encodeURIComponent(val)+"&col=y";
        httpRequest("GET",url,true,handleResponse);
    }
}
```

The code iterates through each radio button and sets up their`onclick` event handlers to call the function `getSports()`. The parameter to this function is an object that represents the clicked radio button.

> NOTE
>
> The code uses the `this` JavaScript keyword to pass a reference to the radio button object that was clicked.

`getSports()` first checks that the radio button is selected (i.e., its`checked` property is `true`), then stores the radio button's value inside the variable`val`. A radio button has a `value` attribute, which is accessible by JavaScript.

```
<input type="radio" name="_sports" value="team" />
```

The code needs the value to request a specific sports category from the server, in the case of the example "team" sports. The code includes this value in a URL parameter, which represents the location of a server component. The `httpRequest()` function then uses the request object to grab the list of sports that the code will use to generate the new bulleted list.

Dodging Bullets

The user clicks a radio button and Presto! a new bulleted list shows up. How does the code create thi list? The second stage of dynamically generating bullets involves using the server response as ingredients for the new list.

```
var resp =  request.responseText;
if(resp != null){
   //return value is a JSON array
   var objt = eval(resp);
   createBullets(objt);
}
```

The response comes in from the server and is accessible from the request object's`responseText` property. The code uses the `eval()` function so that the program can use the response as a JavaScript array. (See Hack #6). A typical value for the response is `["cycling","track","swimming"]`. This value is then passed into the`createBullets()` function, the final step for generating the new list.

```
function createBullets(obj){
```

```
    var _div = document.getElementById("bullets");
    var innerHTMLstr =  "<ul>";
    //obj is an array of new sports names
    for(var i=0; i < obj.length;i++) {
        innerHTMLstr += "<li>";
        innerHTMLstr += obj[i];
        innerHTMLstr += "</li>";
    }
    innerHTMLstr += "</ul>";
    _div.innerHTML= innerHTMLstr;
}
```

A fairly simple method for generating web content is to use a `div` element to enclose the new tags, then rewrite the `innerHTML` property of the `div` tag. `innerHTML` takes a `string` that represents the HTML content of the tag. `createBullets()` gets a reference to the `div` by using the method `document.getElementById()`, and using the value of the `div` tag's `id` attribute. Then the code creates a new `unordered list` tag, with each new `list` item representing a team or individual sport.

An `unordered list` in HTML looks like:

```
<ul>
<li>item one</li>
<li>item two</li>
</ul>
```

To make sure the page continues to be valid XML (we are using an XHTML DTD), the code makes sure each `list` item has a closing list-item tag, as in `</li>`. Each click of the radio button generates a different bulleted list, because the server is sending back different data based on whether the code requests team or individual sports.

Plumbing the Hack

After generating the new bulleted list, the web developer will not be able to see the list's code by using `View Source` in the browser. Typically, you can peak at the source code for a web page by choosing a menu command such as View Page Source from the Firefox menu. This isn't true of new tags that are added by this hack's techniques. If you want to see these new tags, you can use Firefox's DOM Inspector, as described by Debug Ajax-Generated Tags in Firefox.

Submit Hidden Tag Values To A Server Component

Send the values of hidden form fields to the server whenever you want.

The use of hidden form fields may be less compelling these days as newer approaches have evolved such as cookies and sessions to connect one request from the same user to another. However, your application might have other reasons for using an`<input type="hidden">` element.

> NOTE
>
> A hidden field contains a value that the user does not see, unless they peak at the page's source code. It can be used to send the server some extra identifying information along with the rest of the form input.

This hack sends only the value of this field to a server component, when the browser is finished loading the web page.

Let's say you're testing the number of times users open up a page even if the page derives from thei client-side cache. Unless the server specifies with various HTTP response headers that the web page shouldn't be cached, the browser will cache a requested page or keep a local copy of it. The purpose of this caching strategy is to improve performance and prevent unnecessary network requests for the same page if it hasn't changed. This hack however will send a server component the value of a hidden input field whenever the page is loaded into a browser.

> NOTE
>
> You could use such a strategy for web user testing within an application. However, you would probably bombard a network with many wasteful requests if you included this with a production application.

Dynamo

The value of the hidden field is dynamically generated when the browser loads the page. Here is the HTML for the page. The code is minimal, but it includes a hefty hidden value tucked inside of it!

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/2000/REC-xhtml1-20000126/DTD/xhtml1-strict.dtd">
<html  xmlns="http://www.w3.org/1999/xhtml"  xml:lang="en"  lang="en">
<head>
    <script type="text/javascript" src="js/ http_request.js.js"></script>
    <script type="text/javascript" src="js/hacks2_11.js"></script>
```

```
     <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <title>hidden hacks</title>
</head>
<body>
<h3>Delving into some navigator properties</h3>
<div id="content">
These include:   navigator.appName, navigator.platform,
navigator.language, and navigator.userAgent.
</div>
<form  action="javascript:void%200">
    <script type="text/javascript" src="js/innerInput.js"></script>
</form>
</body>
</html>
```

When the page is loaded into the browser, the JavaScript file innerInput.js dynamically creates a hidden `input` tag, mainly to give it a value that's interesting, such as the name of the page, when it was accessed, the computing platform of the user, the default language for the browser, as well as the `User Agent` string associated with the browser they are using. Code can access most of these properties via the `navigator` client-side object in JavaScript. For example, accessing `navigator.platform` returns MacPPC for my computer; `navigator.userAgent` provides the content of the `User Agent` request header from this browser.

Now the `hidden` tag has a lot of meaningful information for its `value` attribute. inner_input.js contains:

```
var delim = ":::";
 document.write(
      "<input type=\"hidden\" id=\"hid\" name=\"data\"  value=\""+
      location.pathname+delim+new Date()+
      delim+navigator.appName+delim+navigator.platform+
      delim+navigator.language+delim+navigator.userAgent+"\" />");
```

The `document.write()` method can dynamically write part of the page as the browser loads the HTML. The latter code creates a `hidden` tag with the `id hid`. The user does not see the value of this tag, but the value is available to JavaScript code. All the different properties such as `navigator.userAgent` are separated by the characters ":::." An example is:

```
/ajaxhacks/ajax_hack2_11.html:::
Thu Oct 27 2005 10:37:15 GMT-0400:::
Netscape:::MacPPC:::en:::Mozilla/5.0 (Macintosh; U;
PPC Mac OS X; en) AppleWebKit/412.6
(KHTML, like Gecko) Safari/412.2
```

Notifying Home

We want to take this information and send it to a server, so that it can be logged. For this task the application requires more JavaScript. The page imports with `script` tags two more JavaScript files.

http_request.js sets up the request object to talk with the server. Hack #3 describes the latter code. The file hacks2_11.js contains the code that accesses the`input` tag's value and sets up a request to POST it to the server as soon as the browser loads the page.

```
window.onload=function(){
    var hid = document.getElementById("hid");
    var val = "navprops="+encodeURIComponent(hid.value);
    url = "http://www.parkerriver.com/s/hid";
    httpRequest("POST",url,true, handleResponse,val);
}
//event handler for XMLHttpRequest
function handleResponse(){
    try{
        if(request.readyState == 4){
            if(request.status == 200){
                //Commented out now: alert(
                 "Request went through okay...");
            } else {
                //request.status is 503
                //if the application isn't available;
                //500 if the application has a bug
                alert(
                  "A problem occurred with communicating between"+
                  " the XMLHttpRequest object and "+
                  "the server program.");
            }
        }//end outer if
    } catch (err)   {
        alert("It does not appear that the server "+
              "is available for this application. Please"+
              " try again very soon. \nError: "+err.message);

    }
}
```

The code gets the value of the `input` element and encodes the value's characters so that they can be properly transferred over the network. Then the code sends a POST request, because of the volume of this server message.

NOTE

A GET request appends the parameters to the end of the URL, whereas a POST request sends the parameter data as a block of characters following the request headers.

The `httpRequest()` function is a wrapper around the code that sets up an `XMLHttpRequest` object and sends the message.

The `httpRequest()` function does a browser compatibility check as in Hack #1. This function also checks for any data that is designed to be posted. This data would appear as the fifth parameter to

the function.

> **NOTE**
>
> JavaScript allows code to define a function and then client code may pass variable arguments to the function. These parameters can be accessed within the defined function as part of an `arguments array`, which every JavaScript function has built-in. Therefore, `arguments[4]` represents the fifth parameter passed into a function.

http_request.js uses the request object's `setRequestHeader()` function to convey to the server component the content type of the sent data. The HTTP POST request will not succeed with Firefox fc instance unless you include this request header. See Hack #2.

```
request.setRequestHeader("Content-Type",
   "application/x-www-form-urlencoded;  charset=UTF-8");
```

Logging

The server component can take the posted data and log it, or whatever the application calls for. Here is an example log entry after a couple of requests with Firefox and Apple Safari (with some of the logged text removed and/or edited for readability).

```
/ajaxhacks/ajax_hack2_11.html:::
Thu Oct 27 2005 10:37:15 GMT-0400:::
Netscape:::MacPPC:::en:::
Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en) AppleWebKit/412.6
(KHTML, like Gecko) Safari/412.2

/ajaxhacks/ajax_hack2_11.html:::
Thu Oct 27 2005 10:49:24 GMT-0400 (EDT):::
Netscape:::MacPPC:::en-US:::Mozilla/5.0 (Macintosh; U;
PPC Mac OS X Mach-O; en-US; rv:1.7.12)
Gecko/20050915  Firefox/1.0.7
```

You can see that the file name is included, followed by the date and time when it was requested, the some browser-specific data such as the default locale (US English) and the value of the `User Agent` request header.

> **NOTE**
>
> In JavaScript, the `User Agent` header data is accessed from the `navigator.userAgent` property.

p(synopsis).

E B V N

We are Vietnames

Chapter 4. Direct Web Remoting (DWR) for Java Jocks

Direct Web Remoting (DWR) for Java Jocks

Previous

E B V N
We are Vietnames

Introduction

Perhaps you want to work with Ajax but not even deal with programming the`XMLHttpRequest` object. An open-source kit called Direct Web Remoting (DWR) is a software layer built on top of this object, completely insulating Web page developers from directly programming the request object.

One advantage of DWR is that you can forget about the boilerplate code we have been using in other hacks to get the HTTP request object working in Microsoft and Mozilla-based browsers. This framework also includes easy techniques for populating web-page widgets with server data, but largely removing the required knowledge of Document Object Model (DOM) programming. The one caveat with DWR is that you must use a Java-based server-side solution, because DWR works with Java servlets and objects behind the scenes.

DWR provides a neat mapping between Java objects and JavaScript code. In other words, you can setup the logic for your application using Java objects on the server, then call those object's methods with JavaScript code when need be. This is called remoting your objects, or making remote Java method calls with JavaScript objects that are bound to the Java objects on the server. The next hack explains the process for setting up DWR and integrating it into a web application.

Integrate DWR Into Your Java Web Application

Design your Ajax application around a JavaScript framework bound to Java objects on the server.

The Direct Web Remoting code comes in the form of an archived or zipped Java Archive (JAR) file. The download address is: http://www.getahead.ltd.uk/dwr/download.html.

> NOTE
>
> The top-level Web page for this free, open source software is: http://www.getahead.ltd.uk/dwr/. Check out the license details for more information while you are visiting this page.

Place the dwr.jar JAR file in the `/WEB-INF/lib` directory of your Java web application, then restart or reload the application.

> NOTE
>
> For those not familiar with Java Web applications, they all have a top-level directory named `WEB-INF`. Inside of `WEB-INF` are XML configuration files, the main one being web.xml. `WEB-INF` also contains a directory named lib, which encloses code libraries or JAR files that the application depends on, such as database drivers and helper classes. The dwr.jar file goes in this `lib` directory.

Configure the application

To get DWR going with your JavaScript, you have to declare in web.xml a Java servlet that DWR uses and add your own DWR-related XML file to `/WEB-INF/lib`.

Here is the chunk of code that you have to add to web.xml. If web.xml already includes registered servlets, then nest this newly declared servlet in with the existing ones. The same goes for the `servlet-mapping` element.

```
<servlet-name>dwr-invoker</servlet-name>
  <servlet-class>uk.ltd.getahead.dwr.DWRServlet</servlet-class>
  <init-param>
     <param-name>debug</param-name>
     <param-value>true</param-value>
  </init-param>
</servlet>
```

```
<servlet-mapping>
  <servlet-name>dwr-invoker</servlet-name>
  <url-pattern>/dwr/*</url-pattern>
</servlet-mapping>
```

> NOTE
>
> You may have to restart the Java web application for the servlet container to create a new instance of this DWR-related servlet.

You also have to create a simple XML file declaring the Java classes that you want to use from your client-side JavaScript code. Don't worry, we'll show you how to use the JavaScript objects that are bound to Java classes shortly! Place this XML file in `/WEB-INF/lib`.

```
<dwr>
  <allow>
    <create creator="new" javascript="JsDate">
      <param name="class" value="java.util.Date"/>
    </create>
     <create creator="new" javascript="JsBean">
      <param name="class" value="com.parkerriver.BikeBean"/>
    </create>
  </allow>
</dwr>
```

This XML states that the client-side JavaScript can use two Java classes remotely. The JavaScript objects that bind the client-side code remotely to the Java classes are named `JsDate` and `JsBean`. As part of the server-side preparations, you had to have already developed the Java class `com.parkerriver.BikeBean` and installed it in your application. `java.util.Date` is part of the Java Software Development kit; it's not your own custom class. `Date` is already available as part of the Java virtual machine your server component is using.

> NOTE
>
> The `BikeBean` class file would typically be stored in `/WEB-INF/classes`, as in `/WEB-INF/classes/com/parkerriver/BikeBean.class`.

This XML file binds the two JavaScript names to the `Date` and `BikeBean` objects, so that these objects will be available to use in your client-side JavaScript. This means that JavaScript code can call all of the public methods of these Java objects. But how is the JavaScript in the local web page connected to the remote Java instances running on the server?

The web page that will use Direct Web Remoting contains these `script` tags, which connect the JavaScript code via the DWR servlet to the server code.

```
<script type="text/javascript" src=
"/[name of web app]/dwr/interface/**JsBean.js**">
```

```
</script>
<script type="text/javascript" src=
"/[name of web app]/dwr/interface/**JsDate.js**">
</script>
<script type="text/javascript" src=
"/[name of web app]/dwr/**engine.js**"></script>
<script type="text/javascript" src=
"/[name of web app]/dwr/**util.js**"></script>
```

Do you recall the simple XML file that we just added to the web application? The first two `script` tags reference the JavaScript names we bound to the Java classes that we want to remote: `JsBean` and `JsDate`. The XML file configured certain Java classes to be used with these names in JavaScript code. Remember the dwr.jar file that we installed in the web application? It contains two JavaScript libraries, engine.js and util.js. The first of these files is required to use DWR; the second is optional and contains a bunch of DWR functions that our client-side code can use.

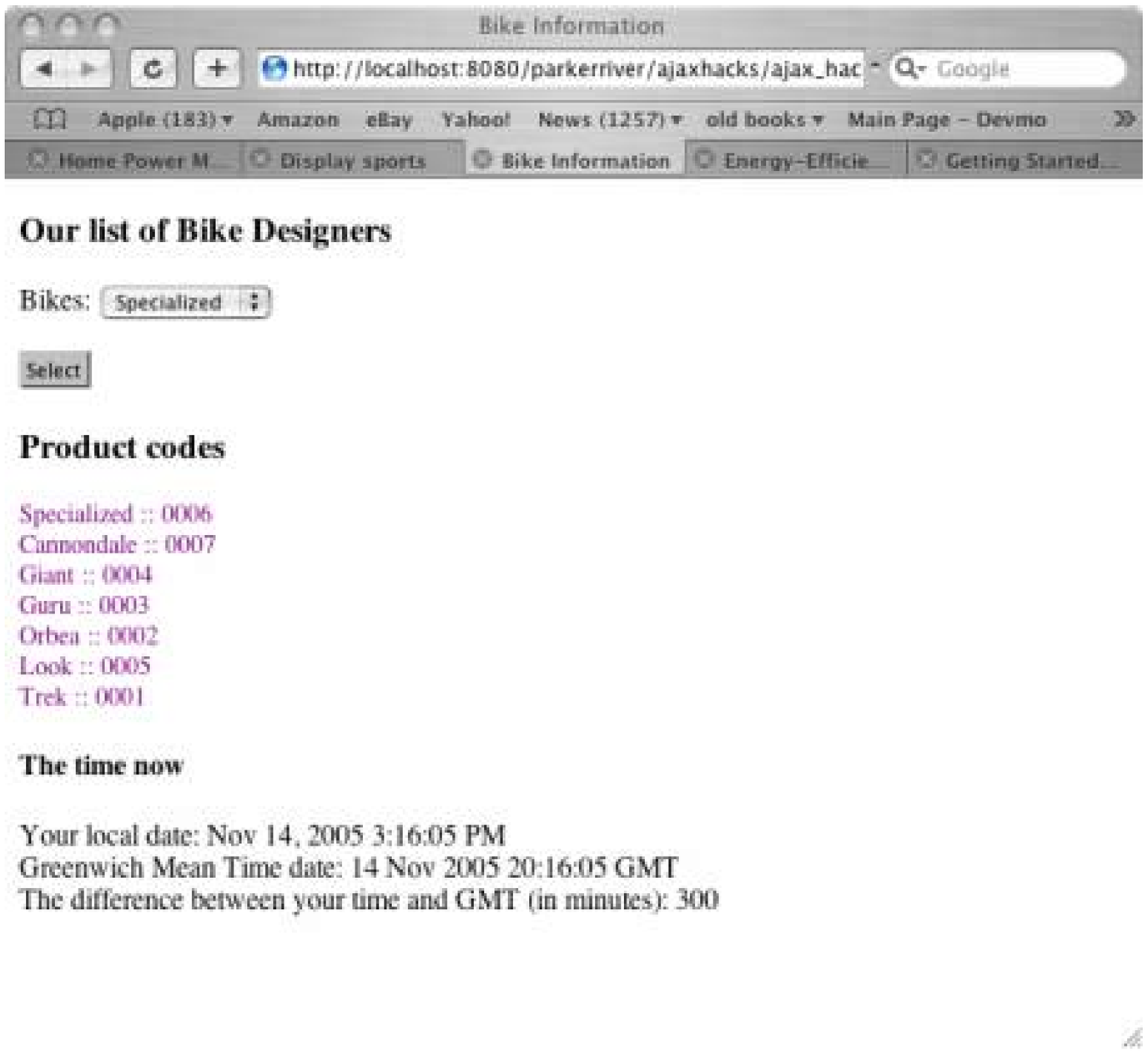
The URL that the `script` tag uses, such as `/parkerriver/dwr/interface/JsBean.js`, connects to the special DWR servlet that we enabled. The servlet in turn makes available to our code the public methods of the Java classes that we configured in XML. The next few hacks will use these classes and functions.

Use DWR To Populate A Select List From A Java Array

Remotely get an `array` return value from a Java object and use the data to populate a `select` list.

Sounds awesome, huh? You can take existing Java objects that have methods returning Java `arrays`, and use those return values to populate a `select` list on a web page. First, figure 5-1 shows the web page that we will use in the next few hacks. The page lists some bike manufacturers in a pop-up widget, a few product codes associated with those companies, then some date/time values. This hack fills the first popup or `select` tag with its values when the page is first loaded.

Dynamically fill a select list with server values



The page imports several JavaScript files using `script` tags. The first four files allow the application to use DWR; the last one contains the code for our application. Here is the underlying web-page code.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/2000/REC-xhtml1-20000126/DTD/xhtml1-strict.dtd">
<html  xmlns="http://www.w3.org/1999/xhtml"  xml:lang="en"  lang="en">
<head>
     <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <script type="text/javascript" src=
"/parkerriver/ajaxhacks/js/hacks5_1.js"></script>
    <script type="text/javascript" src=
"/parkerriver/dwr/interface/JsBean.js"></script>
    <script type="text/javascript" src=
"/parkerriver/dwr/interface/JsDate.js"></script>
    <script type="text/javascript" src=
"/parkerriver/dwr/engine.js"></script>
    <script type="text/javascript" src=
"/parkerriver/dwr/util.js"></script>
    <title>Bike Information</title>
</head>
<body>
<h3>Our  list  of  Bike  Designers</h3>
<form  action="javascript:void%200">
    <p>
        Bikes: **<select id="bikes"></select>**
    </p>
    <p>
        <button type="button" name="selection" value=
                "Select">Select</button>
    </p>
</form>
<h3>Product  codes</h3>
<div id="prodCodes"></div>
<h4>The  time  now</h4>
<div id="showDates"></div>
</body>
</html>
```

To use one of these bound JavaScript objects in your code, you have to set up the server component in the way the previous hack described, then use a `script` tag with the following syntax.

```
<script type="text/javascript" src=
"/[name-of-your-web-app]/dwr/interface/JsBean.js"></script>
```

Substitute `[name-of-your-web-app]` with the name of your web application or "context root" in Java web parlance. In addition, every Ajax application using DWR has to import the engine.js using similar syntax.

```
<script type="text/javascript" src=
"/[name-of-your-web-app]/dwr/engine.js"></script>
```

util.js is optional but contains a lot of useful JavaScript functions, a few of which the upcoming hacks

use.

> NOTE
>
> You can use the utility functions included in util.js standalone, without using the other aspects of DWR such as JavaScript/Java object binding.

Getting an Array From the Server

The hack's code will populate the `select` list using a Java `array` value it receives from a server component. The component is a Java servlet that this chapter's first hack installed, and the `array` source is a `JavaBean` instance we have running on the server. Here is the code for the `BikeBean` class. The `array` derives from this class' `getdesignerInfo()` method. This method returns all of the keys (as in "Trek" or "Cannondale") contained in a `HashMap`, which is a Java object that represents a hash table or associative array, named `bikeInfo`.

```
package com.parkerriver;

import java.util.Map;
import java.util.HashMap;
import java.util.Collections;

public class BikeBean {
       private static Map bikeInfo;
       static{
           bikeInfo = Collections.synchronizedMap(new HashMap());
           bikeInfo.put("Trek","0001");
           bikeInfo.put("Orbea","0002");
           bikeInfo.put("Guru","0003");
           bikeInfo.put("Giant","0004");
           bikeInfo.put("Look","0005");
           bikeInfo.put("Specialized","0006");
           bikeInfo.put("Cannondale","0007");
       }

    public String[] getDesignerInfo(){
        return (String[])bikeInfo.keySet().toArray(new String[]{});
    }

    public static Map getBikeInfo() {
        return bikeInfo;
    }
}
```

This `BikeBean` object is loaded and stored into the server's memory (specifically, inside the Java Virtual machine that the server is using). How does the JavaScript code running inside a distant user's browser get access to the Java object's methods? The XML configuration that the chapter's

first hack explained bound a JavaScript name (`JsBean`) to the `BikeBean` object. The DWR servlet and the engine.js file that the web page imports handles the intermediate magic that connects the browser code to the server code. Here is the JavaScript code in hacks5_1.js that gives the `select` list its value.

```
window.onload=function(){
    setupSelect();
    setupMap();
    setupDates();};

function setupSelect(){
    JsBean.getDesignerInfo(populate);
}
function populate(list){
    DWRUtil.removeAllOptions("bikes");
    DWRUtil.addOptions("bikes", list);
}
/* CODE SNIPPED FOR:
setupMap();
setupDates();
*/
```

When the browser finishes loading the web page, the `window.onload` code calls three different functions. This hack deals with `setupSelect()` (this chapter's upcoming hacks feature the other two functions). `setupSelect()` remotely calls via `JsBean` the `getdesignerInfo()` method. This method returns an `array` of `strings` that represent some names of bike manufacturers. These names will end up as the labels for a `select` list (see figure 5-1).

NOTE

The DWR servlet returns Java values in JSON format. See Hack #6. So a `HashMap` in Java is returned as:

NOTE

```
{  Trek:0001,Specialized:0005,...}
```

DWR uses a callback design pattern as one of the options for working its magic. When the code calls Java methods from JavaScript, an additional parameter representing a callback function is added at the end of the method's parameter list (or is the only parameter in terms of methods that are not defined in Java as having any parameters).

The only parameter to `geTDesignerInfo()` is the name of a function that will handle the Java method's return value (an `array`). The callback function's name is `populate()`, and its parameter is the returned array, here represented by the `list` variable. This code could also have passed in as a

function literal instead of a function name to `getdesignerInfo()`, as in:

```
JsBean.getDesignerInfo(
    function(list){
      DWRUtil.removeAllOptions("bikes");
      DWRUtil.addOptions("bikes", list);
    }
);
```

The code is in essence saying "I'm calling this Java method remotely, and here is the JavaScript function that will handle the return value."

Eccentric Utility

The rest of the code takes this `array` of bike-maker names and dynamically fills a `select` list with them using a couple of DWR's utility functions. The web page made these functions available by importing util.js using a `script` tag, as earlier explained in this hack.

`DWRUtil.removeAllOptions()` takes the `id` of a `select` list as a parameter, then removes all of the options (a logical first step before you change the options in the list). The web page's `select` list looks like:

```
<select id="bikes"></select>
```

`DWRUtil.addOptions()`, on the other hand, takes the `id` of a `select` list as the first parameter, and an `array` as its second parameter. The `array` members then become the options or labels of the `select` list. You might recall that the `list` variable contains the `array` returned by the Java method to which our JavaScript code is bound. Again, our code looks like:

```
DWRUtil.addOptions("bikes", list);
```

If you are a Java web developer, this is cool stuff. The next hack populates a `select` list from a Java `Map` type such as `java.util.HashMap`.

Use DWR To Populate A Selection List From A Java Map

Create a selection list with the Map keys as option values and the Map values as the option content.

A selection list is a popup button on a web page. This element involves a`select` tag with one or more ne following cases, not United Kingdom or France.

```
<select><opton value="uk">United Kingdom</option>
<option value="fr">France</option></select>
```

This hack uses DWR to generate a pop-up from a Java`Map` , using `Map` keys as the values of the `option` HTML code, including the `script` tags that import various JavaScript libraries and the `select` tag itself.

```
<head>
     <meta http-equiv="content-type" content="text/html; charset=utf-8" />
      <script  type="text/javascript"  src="/parkerriver/ajaxhacks/js/hacks5_3.js">
    </script>
    <script type="text/javascript" src="/parkerriver/dwr/interface/JsBikeBean.js">
    </script>
    <script type="text/javascript" src="/parkerriver/dwr/engine.js"></script>
    <script type="text/javascript" src="/parkerriver/dwr/util.js"></script>
    <title>Bike Information</title>
</head>
<!--snipped... -->
<p>
Bikes: **<select id="bikes"></select>**
</p>
```

engine.js is a required JavaScript code library for web pages that are using the DWR framework.`JsBikeE`

Code Ahead

Here is the code in hacks5_3.js for making a remote Java call and loading up the`select` list.

```
window.onload=function(){
    //return a JSON value of a HashMap;
    //populate is the function that will handle
    //the return value
    JsBikeBean.toJSON(populate);};

//map represents the HashMap in JSON format
function populate(map){
    //DWR utility function in util.js
```

```
        DWRUtil.removeAllOptions("bikes");
        //add the Map values to the select tag
        //with id "bikes"
        addOptionValues("bikes", map);
    }
    function addOptionValues(_id,_map){
        //handle the return value as a JS object
        var jMap = new Function("return "+_map)();
        var sel = document.getElementById(_id);
        var opt = null;
        if(sel != null){
             for(var prop in jMap) {
                 opt=document.createElement("option");
                 opt.setAttribute("value",jMap[prop]);
                 opt.appendChild(document.createTextNode(prop));
                 sel.appendChild(opt);
             }
        }
    }
```

Ponder `window.onload`, which points to the function that the browser's JavaScript runtime calls when tl

```
JsBikeBean.toJSON(populate);
```

This function in turn removes any existing `options` from the `select` element, and then creates new `opti` variable refers to a JavaScript object to which the server component's return value was converted.

NOTE

See `Receive Data as a JavaScript Object`.

In the `for/in` loop, `prop` represents the name of each of the original `Map` keys, as in "Cannondale" or "Tr

NOTE

In the code, `jMap` is a JavaScript object as in

NOTE

```
{"Trek":"0006"}
p(note). Using the syntax jMap["Trek"] returns that property value, as in "0006."

The options that this code creates look like <literal>&lt;option value="0006"&gt;T
```

```
h1. Display The Keys/ Values From A Java Hash Map On A Web Page id. 31634 role. ha

div(synopsis).
Connect to a Java object running on the server and use JavaScript to display a <lite
div(synopsis).

This hack takes a <literal>java.util.HashMap</literal> containing the names of bik

p(note). A <literal>java.util.HashMap</literal> in Java is a hash table structure th

The place on the web page where we want to display these values looks like this in t

<h3>Product  codes</h3>
<div id="prodCodes"></div>
```

This code represents a subheading and a `div` element with the `id prodCodes` . When the web page loads
The server component and web page are set-up and configured just as in`Integrate DWR into your Ja`

```
<script  type="text/javascript"  src="/parkerriver/ajaxhacks/js/**hacks5_1.js**"></scri
<script type="text/javascript" src="/parkerriver/dwr/interface/**JsBean.js**"></script>
<script type="text/javascript" src="/parkerriver/dwr/interface/**JsDate.js**"></script>
<script type="text/javascript" src="/parkerriver/dwr/**engine.js**"></script>
<script type="text/javascript" src="/parkerriver/dwr/**util.js**"></script>
```

The code from hacks5_1.js calls a `JsBean` method to display the converted `HashMap's` values inside the

```
private static Map bikeInfo;
       static{
           bikeInfo = Collections.synchronizedMap(new HashMap());
           bikeInfo.put("Trek","0001");
           bikeInfo.put("Orbea","0002");
           bikeInfo.put("Guru","0003");
           bikeInfo.put("Giant","0004");
           bikeInfo.put("Look","0005");
           bikeInfo.put("Specialized","0006");
           bikeInfo.put("Cannondale","0007");
       }

    public static Map getBikeInfo() {
        return bikeInfo;
    }
```

The `getBikeInfo()` method simply returns the `Map` with all of these values.

NOTE

A comprehensive real-world application rather than our hack might be returning a`Map` derived from

Traveling by the speed of light from the server to the browser code, here is the web page's underlying J

```
//This method is called by the window.onload event handler
function setupMap(){
    JsBean.getBikeInfo(setProdCodes);
}
//"data" is the JS object representation of a HashMap
function setProdCodes(data){
    var div = document.getElementById("prodCodes");
    //remove old messages
    div.innerHTML="";
    div.style.color="purple";
    div.style.fontSize="0.9em";
     var tmpText;
    for(var prop in data) {
       tmpText = prop + " :: "+ data[prop];
       div.appendChild(document.createTextNode(tmpText));
       div.appendChild(document.createElement("br"));
    }
}
```

`getBikeInfo()` returns the `HashMap` value and passes it as the parameter to the `setProdCodes()` functi

```
JsBean.getBikeInfo(setProdCodes);
...
function setProdCodes(data){...}
```

`setProdCodes()` represents the callback mechanism that DWR uses to exchange data between the serv

NOTE

The JavaScript code passes a callback function name as a parameter to the Java method. Make sur

NOTE

```
JsBean.getBikeInfo(setProdCodes());
```

The `HashMap` that originated on the server manifests as the callback function's parameter. The`data` para font size and color of the text.

```
div.style.color="purple";
div.style.fontSize="0.9em";
```

The DWR framework does a lot of useful work for a script and Ajax developer. The framework returns th

NOTE

See Hack #6 for more information on JavaScript Object Notation (JSON). The DWR framework retu

NOTE

```
{  Trek:0001,Specialized:0005,...}
```

```
for(var prop in data) {
       tmpText = prop + " :: "+ data[prop];
       div.appendChild(document.createTextNode(tmpText));
       div.appendChild(document.createElement("br"));
}
```

The code writes the bike maker names and product codes by displaying the key and value followed by a

```
/* i.e., data["Trek"]returns "0001" */
tmpText = prop + " :: "+ data[prop];
div.appendChild(document.createTextNode(tmpText));
div.appendChild(document.createElement("br"));
```

tmpText contains the line of text that the web page displays, as in "Trek :: 0001." During each iteration

---

Use DWR To Populate An Un/ Ordered List From A Java Array

Use a framework to dynamically populate a web-page widget from values derived from a Java object

This hack automatically (you might say auto magically) generates an ordered or unordered list using server content, such as a list of high-end bike makers. A typical list on a web page is hard-coded intc the HTML, or a web page's code. It looks like a series of bullets or numbers, each accompanied by a label. These list types involve content that never or hardly ever changes. On the other hand, some lists must be dynamically generated from a server object, based on persistent information such as that contained in a database.

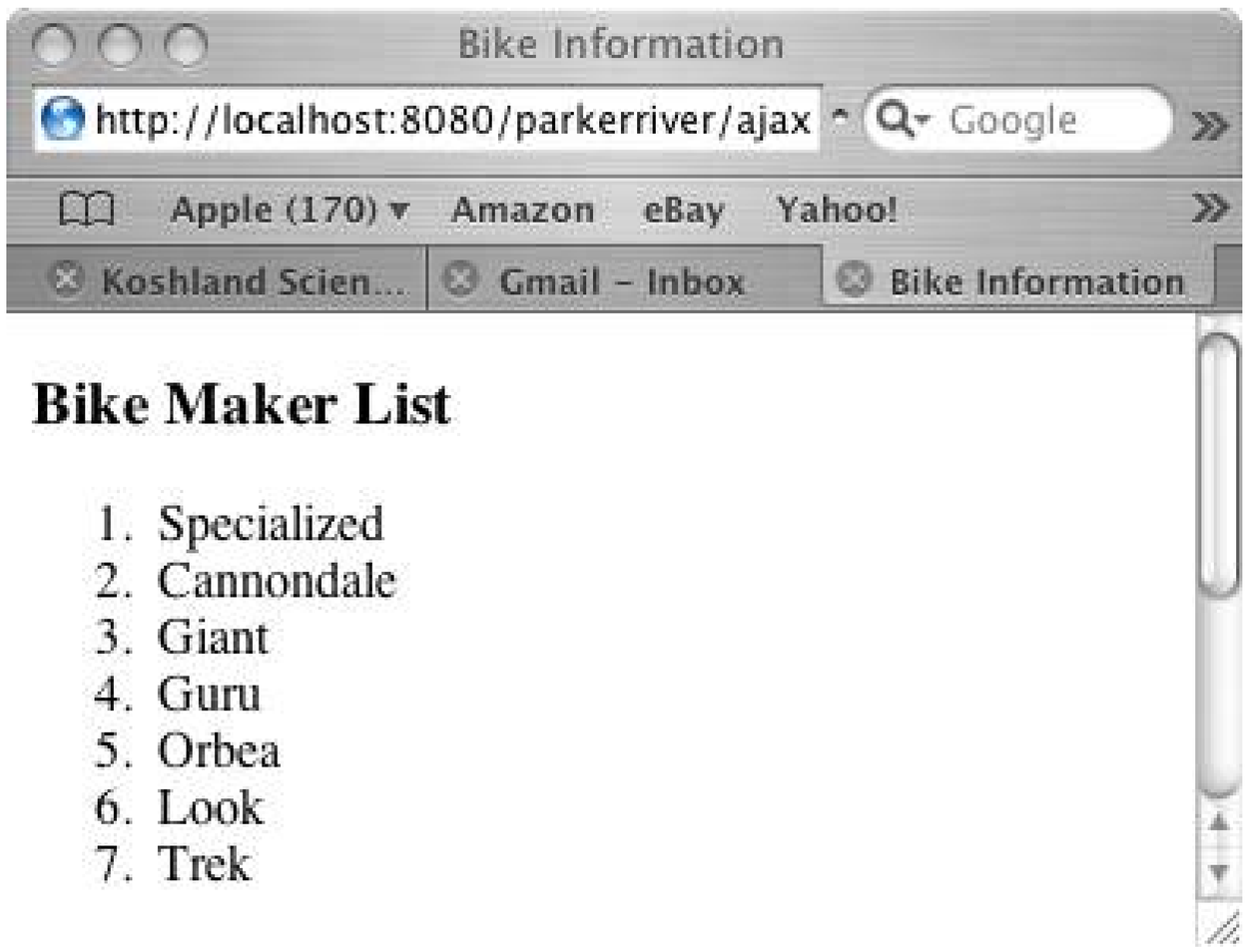
> NOTE
>
> A dynamically generated list is only necessary for persistent information that is updated frequently, such as a store that is constantly adding new products and/or changing product attributes.

Think of a bike shop constantly adding new products and their attributes to their online store.

The web page code in this hack derives the content by calling a Java method via Direct Web Remoting. DWR is designed to bind JavaScript objects to Java objects running on the server. The firs hack in this chapter sets up and configures the Java application on the server end, which is the first step to running this hack.

This hack generates an ordered list on the same web page that other hacks in this chapter have used. This is an `ol` tag that contains a numbered list of bike makers. We include the option to generate an unordered list, a `ul` tag containing bullets to the left of the labels. When the web page loads, its underlying code automatically fetches an `array` of bike-maker names from a server and generates the list. Figure 5-2 shows what the web page looks like.

5-2. A list of bike makers



The web page imports all of the necessary JavaScript files with`script` tags, and includes the list within a `div` tag with `id` "orlist."

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/2000/REC-xhtml1-20000126/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
     <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <script type="text/javascript" src=
"/parkerriver/ajaxhacks/js/hacks5_1.js"></script>
    <script type="text/javascript" src=
"/parkerriver/dwr/interface/JsBean.js"></script>
    <script type="text/javascript" src=
"/parkerriver/dwr/interface/JsDate.js"></script>
    <script type="text/javascript" src=
"/parkerriver/dwr/engine.js"></script>
    <script type="text/javascript" src=
"/parkerriver/dwr/util.js"></script>
    <title>Bike Information</title>
</head>
<body>
<!--SNIPPED -->
<p**><input id="hid" type="hidden" value="ordered" />**
</p>
<!--SNIPPED -->
<div id="orlist"></div>
<!-- ... -->
</body>
</html>
```

hacks5_1.js includes our custom code, which we'll show in a moment. All of the other imported JavaScript files are DWR-related. engine.js is required if your code is using the DWR framework; util.js is an optional library of utility functions, one of which we use in this hack.`JsBean` is a JavaScript class that is bound to a Java object. Here is the hacks5_1.js code this hack uses.

```
window.onload=function(){ callSetups()};

function callSetups(){
    setupSelect();
    setupMap();
    setupList(document.
     getElementById("hid").value);
    setupDates();
}
function setupList(typ){
    JsBean.getDesignerInfo(function(list) {
        var div = document.getElementById("orlist");
        var el = null;
        if(div != null){
            //remove any existing lists
            div.innerHTML="";
            if(typ.indexOf("un") == -1) {
                //create an ordered list
                el=document.createElement("ol");
            }   else {
                //create an unordered list
                el=document.createElement("ul");
            }
            el.setAttribute("id","servlist");
            div.appendChild(el);
            //create li elements from server information
            DWRUtil.addOptions("servlist", list);
        }
    });
}
//Rest of code snipped...
```

One salient code aspect is that it does not require`XMLHttpRequest` or our custom library for using the request object, http_request.js (see hack #3). The DWR framework takes care of its remote binding between JavaScript and Java.

An event handler linked to `window.onload` calls a `setupList()` function. `setupList()` has a `string` specifying "ordered" or "unordered" as a parameter. The code gets this value from a`hidden` element on the web page, so that a web page designer or writer can specify the type of list. Inside of `setupList()` the code calls the Java method `geTDesignerInfo()`, via its client-side proxy `JsBean`. This method returns an `array` of bike maker names.

NOTE

DWR returns JSON values from Java objects (see Hack #6), unless you configure a

different "converter" DWR can use when setting up the framework. See http://getahead.ltd.uk/dwr/documentation,

The way DWR works when remotely calling Java methods is that a parameter representing a function for handling the Java return value is added to the method call, as in `JsBean.callFoo(function(returnValue){//handle callFoo return value})`. The latter example, as in the web-page code, handles the return value with a function literal. You could alternatively use `JsBean.callFoo(myFunc(returnValue))`, then define `myFunc()` somewhere. The framework passes the Java-method return value to this handler function as its parameter.

In our code, the function literal that handles the `geTDesignerInfo()` return value looks like this.

```
function(list) {
    var div = document.getElementById("orlist");
    var el = null;
    if(div != null){
       //remove any existing lists
       div.innerHTML="";
       /* The function literal has access to the
          type parameter of the outer function; typ
           can be "ordered" or "unordered" */
       if(typ.indexOf("un") == -1) {
          //create an ordered list
          el=document.createElement("ol");
        }   else {
          //create an unordered list
          el=document.createElement("ul");
        }
          el.setAttribute("id","servlist");
          div.appendChild(el);
          //create li elements from server information
          DWRUtil.addOptions("servlist", list);
        }
})
```

list is the array returned from the server, which looks like `["value1","value2"]`. First the code determines whether to create an ordered or unordered list. Then the code appends the new element with `id` "servlist" as a child within an existing `div` element. Finally, the function uses a DWR utility function to generate the new list.

```
DWRUtil.addOptions("servlist", list);
```

Figure 5-3 shows what the browser looks like after generating an unordered list.

```
<head>
    <meta http-equiv="content-type" content=
     "text/html; charset=utf-8" />
    <script type="text/javascript" src=
"/parkerriver/ajaxhacks/js/hacks5_1.js"></script>
    <script type="text/javascript" src=
```

```
"/parkerriver/dwr/interface/JsBean.js"></script>
    <script type="text/javascript" src=
"/parkerriver/dwr/interface/JsDate.js"></script>
    <script type="text/javascript" src=
"/parkerriver/dwr/engine.js"></script>
    <script type="text/javascript" src=
"/parkerriver/dwr/util.js"></script>
    <title>Bike Information</title>
</head>
```

Creating an unordered list from remote Java

NOTE

The content on the bottom of figure 5-3 relates to calling a built-in Java object using DWR. Our last hack covers this mechanism.

This function takes the `id` of the list as the first argument, and the `array` of values as the second. If the code has to remove existing options from a list first, one option would be

`DWRUtil.removeOptions(&servlist&)`. `addOptions()` is the same function as Hack #50 used to populate a `select` list.

Previous

Access A Custom Java Object With Java Script

Receive a serialized Java object via Ajax then use that object with JavaScript.

The programming model for a number of Java applications involves generating JavaBeans that represent data. A JavaBean is an object representation of a concrete thing like a bicycle, with its wheels, pedals, seat, chain rings, and other components as object properties. The purpose of a JavaBean is to represent these concrete entities for a software program that accomplishes a set of practical tasks involving the entity data type, such as an e-commerce site that sells bikes. Therefore, it is natural that some Ajax applications will receive data from a server component in the form of JavaBeans.

This hack uses the Direct Web Remoting (DWR) framework to access a JavaScript representation of a Java object from the server. Then the hack displays the object on a web page.

The Big Set-up@@

To use DWR with Ajax, you have to set it up on the server first. Integrate DWR into your Java Web Application described this process in detail, so we will not repeat it here, except to show this hack's XML configuration file. On the server end, this file must be stored in`/WEB-INF/lib`. The file gives DWR its instructions for creating an instance of the Java class that your application calls remotely from JavaScript.

```
<dwr>
  <allow>
    <create creator="new" javascript="JsBikeBean">
      <param name="class" value="com.parkerriver.BikeJavaBean"/>
    </create>
  </allow>
</dwr>
```

As specified in this configuration file, the JavaScript name your code uses for the remote method call is `JsBikeBean`. Figure 5-4 shows the web page when it's first requested. The underlying code requests a serialized version of the `BikeJavaBean` object when the web page is first loaded. Then it displays this object as a `string` in an alert window.

Voila, serialized Java object

Here is the code for the `BikeJavaBean` class, for which `JsBikeBean` is remoted.

```
package com.parkerriver;

import java.util.Map;
import java.util.HashMap;
import org.json.JSONObject;

public class BikeJavaBean {
    private Map bikeInfo;

    public BikeJavaBean(Map bikeInfo) {
        this.bikeInfo = bikeInfo;
    }

    public BikeJavaBean() {
        bikeInfo = new HashMap();
        bikeInfo.put("Trek","0001");
        bikeInfo.put("Orbea","0002");
        bikeInfo.put("Guru","0003");
        bikeInfo.put("Giant","0004");
        bikeInfo.put("Look","0005");
        bikeInfo.put("Specialized","0006");
        bikeInfo.put("Cannondale","0007");
    }

    public String[] getbikeMakers(){
        return (String[])bikeInfo.keySet().
        toArray(new String[]{});
    }

    public Map getBikeInfo() {
        return bikeInfo;
    }
```

```
    public String toJSON(){
        /* There are different ways to serialize a Java object
         using a JSONObject constructor; here we are constructing
         a JSONObject using the Java object's HashMap */
        JSONObject jo = new JSONObject(getBikeInfo());
        return jo.toString(4);
    }
}
```

This is an object that contains a hash table structure involving the names of bike makers keyed to some imaginary product codes. Our JavaScript object named `JsBikeBean` (check out the earlier configuration) is bound to this java object. Pay special attention to the `toJSON()` method. This is the method that our Ajax code will call to access a serialized version of the JavaBean.

The code uses a `JSONObject` type, which derives from the Java API for JavaScript Object Notation (JSON). This rather dense acronym list relates to the convenient JSON return values that several of our hacks have dealt with (See `Receive Data as a JavaScript Object`). We have bundled this API and related classes in with the rest of our server-side Java classes. The Java API for JSON simply offers Java classes that make it easier to return JSON formatted values to Ajax applications.

The purpose of returning JSON formats to Ajax is that they can easily be converted to JavaScript objects, which often makes it easier for Ajax to work with the data.

NOTE

See `Receive Data as a JavaScript Object` and http:///www.json.org.

The bean's code creates a `JSONObject` by passing into the `JSONObject's` constructor the bean's `HashMap` of bike-maker data. This code essentially wraps the bean's data inside this special object.

```
JSONObject jo = new JSONObject(getBikeInfo());
return  jo.toString(4);
```

Then the code calls the `JSONObject's toString()` method, which returns the `string` version of the bike-maker names and product codes that show up in the browser `alert` window.

NOTE

In the programmer world, representing an instance of an object in a different format, and preserving its internal state or property values, is sometimes called marshalling. So in this case we're marshalling a Java object into JSON format. Going the other way, from say XML back into a Java object, is called unmarshalling.

Here is the HTML code for the web page. As usual, the key parts of this page are the `script` tags that import the necessary JavaScript libraries.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/2000/REC-xhtml1-20000126/DTD/xhtml1-strict.dtd">
<html xmlns=
"http://www.w3.org/1999/xhtml"  xml:lang="en"  lang="en">
<head>
     <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <script type="text/javascript" src=
"/parkerriver/ajaxhacks/js/hacks5_5.js"></script>
    <script type="text/javascript" src=
"/parkerriver/dwr/interface/JsBikeBean.js"></script>
    <script type="text/javascript" src=
"/parkerriver/dwr/engine.js"></script>
    <script type="text/javascript" src=
"/parkerriver/dwr/util.js"></script>
    <title>Bike Information</title>
</head>
<body>
<h3>Our list of Bike Designers</h3>
<div id="bean"></div>
</body>
</html>
```

The two highlighted `script` tags import the JavaScript libraries that are required to use DWR: JsBikeBean.js, which in our case binds a JavaScript object of that name to the JavaBean running on the server, and `engine.js`, which is the framework code. hacks5_5.js represents the code for this hack, and util.js is an optional library that contains several useful functions.

> NOTE
>
> See `Integrate DWR into your Java Web Application` for more details on setting up the DWR framework on the server.

When the user dismisses the `alert` window, the web page's code uses the returned `JavaBean` object (in `JSON` format) to display the object's data on the page. Figure 5-5 shows this page.

Displaying a serialized JavaBean

You are probably curious by now what the code does in hacks5_5.js. How does the web page code display the JavaBean information that the server component returns? How does the web page make the request in the first place?

```
window.onload=function(){
    JsBikeBean.toJSON(function(javaStr){
        alert(javaStr);
        var div = document.getElementById("bean");
        //remove old content
        div.innerHTML="";
        //convert the return value to a Java object
        var javaObj = new Function("return "+javaStr)();
        var   innerHt="<p>Property names and product codes:</p>";

        for(var propName in javaObj) {
            innerHt += "<p>";
            innerHt += "<strong>";
            innerHt += propName;
            innerHt += "</strong> : ";
            innerHt += javaObj[propName];
            innerHt += "</p>";
        }
        div.innerHTML=innerHt;
    });
};
```

The framework takes care of making the HTTP request, so the code does not contain any references to `XMLHttpRequest` or the `httpRequest()` function that we have seen in other hacks. The `JsBikeBean.toJSON()` function is a remote method call that returns the serialized (or JSONized) JavaBean. DWR uses the callback mechanism, where the argument to the remote method is a

function that will handle the server's return value. That function, in turn, has the return value as its lone argument. Our code uses a function literal, in which the entire function definition is passed into the remote method call.

First, an `alert` window shows the returned `string`. Then the code converts the JSON-formatted `string` into a JavaScript object using a special technique.

> NOTE
>
> `Receive Data as a JavaScript Object` describes this technique, a line of code that makes JavaScript interpret the JSON-formatted string as an object.

In the code, the variable `javaObj` now represents a plain old JavaScript object that the code easily explores with a `for/in` loop. This loop builds a `string`, which displays the object's values inside a `div` element.

```
for(var propName in javaObj) {
   innerHt += "<p>";
   innerHt += "<strong>";
   innerHt += propName;
   innerHt += "</strong> : ";
   innerHt += javaObj[propName];
   innerHt += "</p>";
}
div.innerHTML=innerHt;
```

Hacks like this can easily integrate existing JavaBeans that various server components might use. Using the Java API for JSON is just a matter of downloading and the compiling the source code for objects such as `JSONObject` and `JSONArray`.

> NOTE
>
> See http://www.crockford.com/JSON/java/index.html/.

Call A Built-in Java Object From Java Script Using DWR

Extend your code's reach by calling built-in Java objects remotely.

What if you had to read a file like a log on the server from a JavaScript object on the client browser? You might want to use the `java.io.FileReader` class on the server. This class is part of the Java 2 Standard Edition, a fancy way of saying that `FileReader` is built-in to Java but not JavaScript. The DWR framework allows you to easily call standard Java methods from your JavaScript. This hack displays some date information on a web page. The data derives from remote method calls using the `java.util.Date` object.

> NOTE
>
> JavaScript has a robust `Date` object and several associated methods, which you would use in most real-world production applications that display dates on a web page. It's still nice to know, from at least a hack writer's perspective, that a great variety of standard Java objects and their methods are available from JavaScript. At the very least, you can adapt these techniques to several other similar situations.

The code displays the current date, and compares this data to the Greenwich Mean Time date. Figure 5-3 in Use DWR to populate an Un/Ordered List From a Java Array shows what the `Date` information looks like on the web page.

Setting up this code involves a little server configuration, as this chapter's first hack explained. Here is the configuration file on the server.

```
<dwr>
  <allow>
    <create creator="new" javascript="JsDate">
      <param name="class" value="java.util.Date"/>
    </create>
  </allow>
</dwr>
```

This XML file binds the JavaScript name `JsDate` to a corresponding Java `Date` object. In a Java web application, this XML file belongs in `/WEB-INF/lib`. Make sure dwr.jar is also in `/WEB-INF/lib`. If you're still setting up DWR on the server, check back to Hack #48 for a summary of the required steps.

The next step on the client-side is to import all of the necessary JavaScript libraries into the web page that is calling the Java object remotely.

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

```
"/parkerriver/ajaxhacks/js/hacks5_1.js"></script>
    <script type="text/javascript" src=
"/parkerriver/dwr/interface/JsDate.js"></script>
    <script type="text/javascript" src=
"/parkerriver/dwr/engine.js"></script>
    <script type="text/javascript" src=
"/parkerriver/dwr/util.js"></script>
    <title>Bike Information</title>
</head>
```

The first imported script, hacks5_1.js, includes the code for our application. The second highlighted `script` tag involves the `JsDate` object, which DWR binds to the `java.util.Date` object. We can use this JavaScript object to call methods on the Java `Date` object. The next two imported libraries, engine.js and util.js, represent a required library for using DWR and an optional utilities library, respectively.

The hack's web page includes an `h3` subheading tag and a `div` for containing the `Date` information.

```
<h4>The time now</h4>
<div id="showDates"></div>
```

Here is the code for remotely calling the `Date` object.

```
window.onload=function(){setupDates();};

function setupDates(){
    var div = document.getElementById("showDates");
    //remove old messages
    div.innerHTML="";
    //define callback function for displaying a local date
    JsDate.toLocaleString(function(dateString){
        div.appendChild(document.createTextNode(
        "Your local date: "+dateString));
        div.appendChild(document.createElement("br"));
    });
    //define callback function for displaying
    //Greenwich Mean Time
    JsDate.toGMTString(
     function(dateString){
       div.appendChild(document.createTextNode(
           "Greenwich Mean Time date: "+
            dateString));
       div.appendChild(document.createElement("br"));});

JsDate.getTimezoneOffset(
   function(dateString){
     div.appendChild(document.createTextNode(
     "The difference between your time and GMT (in minutes): "+
     dateString));
        }
```

```
            );
}
```

This code displays the date information as part of the`window.onload` event handler, which the browser's JavaScript implementation calls when the browser finishes loading the web page. `setupDates()` then displays different elements of the current time by calling three Java`Date` methods remotely.

- `toLocaleString()` generates a current time and date string, as in `Nov 21, 2005 7:58:16 AM`.
- `toGMTString()` displays the same kind of string but in Greenwich Mean Time.
- `getTimezoneOffset()` displays the number in minutes representing the difference between the user's current time and GMT. For example, my time in Massachusetts is 300 minutes or six hours behind GMT time.

The code uses the `JsDate` object to remotely call these Java methods. As part of the DWR mechanism, the lone parameter for these method calls is a function that handles the Java return value, in these cases, various date/time strings. For example, here is the function that handles the `toLocaleString()` return value.

```
function(dateString){
  div.appendChild(document.createTextNode(
  "Your local date: "+dateString));
  div.appendChild(document.createElement("br"));
}
```

The `dateString` parameter represents the actual `string` returned by remotely calling `java.util.Date.toLocaleString()`. The `div` tag our page uses for displaying this information creates a new text node representing this `string` followed by a line-break `br` tag.

NOTE

For information on all of the different options for making Java remote method calls from JavaScript see this DWR page: http://getahead.ltd.uk/dwr/browser/intro.

After initially loading the web page, the user can refresh the page and the date/time strings will change reflecting the most current time of day locally and in terms of GMT.

NOTE

You could Hack the Hack by including a Refresh button with an`onclick` event handler that updates the date information, without refreshing the entire page.

E B V N

We are Vietnames

Chapter 86. To Come

Idtxt A

Previous

E B V N
We are Vietnames

Idtxt A

TK .

Previous

E B V N
We are Vietnames