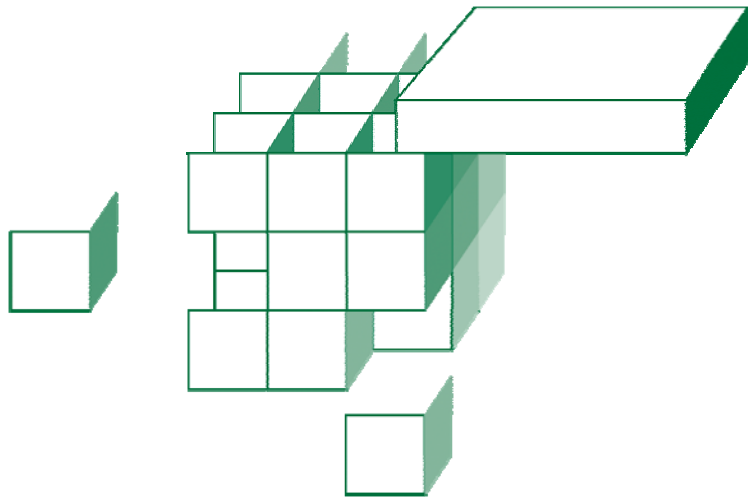


CHAPTER 6

PROTOTYPE



OBJECTIVES

After completing "Prototype," you will be able to:

- Describe the usefulness of JavaScript libraries, including Prototype and Dojo, for Ajax and other purposes.
- Understand the importance of server-side Ajax frameworks, such as **DWR**, **ASP.NET**, and various JSF component libraries.
- Use Prototype's **Ajax.Request** object to streamline the coding of Ajax request/response processing.
- Use Prototype convenience functions such as **\$**, **\$F**, and **\$A**.
- Practice algorithmic programming using Prototype's **Array** functions, including **find**, **findAll**, and **each**.

Making Things Easier

- JavaScript does not come with a standard class library.
- As practiced using the techniques we've seen so far, Ajax programming suffers some limitations and frustrations:
 - **Remote requests** using XHR are possible, but the process is clunky and error-prone.
 - The basics of **user interface** design are there, but again the programming model is awkward, and it's a lot of work to get even a simple, attractive form pulled together, let alone more sophisticated **widgets** and **visual effects**.
 - The **event model** leaves much to be desired: one piece of code can unintentionally detach another piece of code as an event handler, and browsers are not consistent in their implementations.
- This threatens to leave the Ajax programmer in the position of re-inventing the wheel – or many different wheels.

Script Libraries

- Fortunately JavaScript offers a handful of simple, powerful re-use mechanisms, including simply referring to a source file from an HTML page, thus making functions and object types from that source file usable from the page.
- **Script libraries**, then, can implement common functionality and make it easy to tap it for an individual page.
- Two of the most popular Ajax script libraries are:
 - **Prototype**, which is minimal but on which many more ambitious libraries are built, including **script.aculo.us**
 - **Dojo**
- We'll study Prototype in this chapter, and Dojo in the next one.
- Dojo is larger and more feature-rich than Prototype proper, but there are many areas of overlap as well.
- We'll consider a few features of each library, aiming to get familiar with each while also developing an appreciation of the value of any high-quality script library.

Ajax Issues

- Ajax issues don't end with the mechanics of getting a remote request and response to travel to and from the server.
- There are deeper questions:
 - How will the server handle Ajax requests – as distinct from traditional page requests?
 - We'd like to have something that takes full advantage of business and maybe presentation logic already defined in the application. How can we cause the page and Ajax requests to converge as early as possible?
 - How might we share the semantics of the business domain between the server and the client?

Ajax Frameworks

- Many Ajax frameworks are available that go beyond client-side code libraries – though they may well include some JavaScript source files of their own.
 - **RMI-style frameworks** such as **Direct Web Remoting** expose the public interface of a server-side class as a more or less ordinary JavaScript object type, automatically piping remote requests to that server-side object.
 - In the Java world, **JavaServer Faces** provides a high-level abstraction of web-request handling, and **JSF component libraries** use this to streamline both Ajax and ordinary page processing.
 - **ASP.NET** includes a high-level Ajax framework.
 - Frameworks for **other .NET languages** are available as well, from various providers including Microsoft.
- **Server-side Ajax frameworks are beyond our scope.**
 - If nothing else, we'd need to commit to a server-side platform (Java, .NET, etc.) before we could go any further.

Prototype

- Prototype is implemented entirely in JavaScript – it doesn't rely on any other server-side or client-side code.
- It is an open-source project, currently at version 1.6.
- It is refreshingly lightweight: it exists as a single file, **prototype.js**.
 - A little over **125k**
 - **4183** lines of code
- Get Prototype at the following URL:

`http://www.prototypejs.org/download`

- You get the **prototype.js** file – and that's it!
 - Put it in your project's web root or anywhere else you like.
- Include Prototype in your own HTML pages using a `<script>` reference:

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

- Several of our upcoming examples use Prototype, and we'll visit one of them just to get familiar with the library:

Examples/Fetch.

- A quick scan of **prototype.js** will reveal the code for several of Prototype's most popular features:
 - Types **Ajax.Request**, **Ajax.Response**, **Ajax.Updater**, etc. – this is Prototype's **remoting library**, which is a thin wrapper around XHR.
 - Several shortcut or **convenience functions**, the most beloved of which, in certain quarters, is **\$**.
 - That's right: the function name is **\$**, and it's an alias for the very common **document.getElementById**:

```
function $(element) {  
  if (arguments.length > 1) {  
    for (var i = 0, elements = [], length =  
      arguments.length; i < length; i++)  
      elements.push($(arguments[i]));  
    return elements;  
  }  
  if (Object.isString(element))  
    element = document.getElementById(element);  
  return Element.extend(element);  
}
```

- Some of these functions are just aliases, and some offer additional convenience over some standard JavaScript or DOM function.

Ajax.Request

- Prototype encapsulates its Ajax support in the type **Ajax**.
- The nested type **Ajax.Request** models an Ajax request.
 - By creating an object of this type, you trigger the actual request.
 - Pass the **request URL** and an **options object** – a data structure that allows you to customize the request and response handling.

```
var options =  
{  
  method: "POST",  
  onSuccess: myHandlerFunction;  
  onFailure: function() { alert("Oh, no!"); }  
};  
new Ajax.Request("someAjaxService", options);
```

- It's typical to create the options object using JSON syntax, as shown above.
- There is an **Ajax.Response** type as well, which aggregates the underlying **XMLHttpRequest** object itself.
 - This is passed to the response handler function.
 - It can be used more or less as XHR is used.
 - But the handler function will only be called if the **readyState** is **4** and the **status** is **200**; so these checks no longer need to be performed in your code.
- Note that **Ajax.Request** is naturally thread-safe, since a new instance must be created in order to send the request.
 - This instance wraps its own, unique XHR object.

The Options Object

- The options object includes the following properties:

Property	Default	Meaning
parameters	null	URL-encoded string containing the request parameters
method	"POST"	HTTP method for the request
onSuccess	null	Function to be called on successful request/response
onFailure	null	Function to be called on an error response (HTTP 400 or 500)
onException	null	Function to be called if the request failed to complete
onComplete, onLoaded, onInteraction	null	Functions to be called during request state changes up to but excluding readyState==4
on404, onXXX		Functions to be called for the corresponding HTTP response code
asynchronous	true	If false , the browser blocks the UI thread until completion
requestHeaders	null	Array of HTTP headers to be sent in the request
postBody	null	Content to be passed in the body of an HTTP POST (apart from request parameters)

- Another variant of our most basic Ajax page is found in **Examples/Fetch** that uses Prototype and **Ajax.Request**.
- See **prototype.html**:
 - **handleResponse** uses the **\$** function (more soon) to find the target element, and **XHR.responseText** to get the HTML fragment:

```
function handleResponse(XHR)
{
  println("handleResponse() called.");
  $("dataArea").innerHTML = XHR.responseText;
  showOutput(buffer);
}
```

- **sendRequest** builds an options object identifying **handleResponse** as the handler function. It then creates the request object, which triggers the actual HTTP request:

```
function sendRequest()
{
  println("sendRequest() called.");
  var options =
  {
    onSuccess: handleResponse,
    onFailure: function(resp) { alert
      ("Couldn't retrieve data using Prototype!"); }
  };
  new Ajax.Request("getData.html", options);
  showOutput(buffer);
}
```

Using Ajax.Request

EXAMPLE

- The results are similar to what we've seen in other variants of this dead-simple Ajax example.
- Open the page in your browser and click the second **Get Data** button:

Using Prototype

Get Data from server using a page request

Get Data from server using Ajax.Request

Clear Data

NEW JERSEY		
1	Robert E. Andrews	Haddon Heights
2	Frank A. LoBiondo	Millville
3	Jim Saxton	Mount Holly
4	Christopher H. Smith	Robbinsville
5	Marge Roukema	Ridgewood
6	Frank Pallone, Jr.	Long Branch
7	Mike Ferguson	Warren Township
8	Bill Pascrell, Jr.	Paterson
9	Steven R. Rothman	Fair Lawn
10	Donald M. Payne	Newark
11	Rodney P. Frelinghuysen	Morristown
12	Rush D. Holt	Hopewell Township

```
sendRequest() called.  
handleResponse() called.
```

The \$ Function

- Prototype addresses more than just Ajax.
- In fact it is mostly focused on more fundamental facilities.
- It is probably best known for the **\$** function, which mostly functions as an alias for **document.getElementById**:

```
var someElement = $("#someID");
```

- We saw this function in use in the previous example.
- It can also return an array of element references, if passed additional IDs as arguments.

- The source code is much admired for its simplicity by experienced JavaScript coders:

```
function $() {  
    var elements = new Array();  
  
    for (var i = 0; i < arguments.length; i++) {  
        var element = arguments[i];  
        if (typeof element == 'string')  
            element = document.getElementById(element);  
  
        if (arguments.length == 1)  
            return element;  
  
        elements.push(element);  
    }  
    return elements;  
}
```

Other Convenience Functions

- Prototype is full of little conveniences along those same lines.
- A bit more verbose in its name is **getElementsByClassName**, which is available on the “extended” **document** object:

```
var allRefs =  
  document.getElementsByClassName("ref");
```

- This looks like a regular DOM method, but no: it is added by Prototype.
- It will be available to any code that follows the inclusion of **prototype.js**, which extends – you guessed it! – the prototype of the document object.
- Another variant of the function limits the scope of the search by specifying the ID of an ancestor element:

```
document.getElementsByClassName("ref", section2);
```

- Another handy one is **\$F**, which takes an ID and returns the value of the form field with the corresponding ID.

```
var lastName = $F("lastNameField");
```

- **\$A** converts an “array-like” object, such as a DOM **NodeList**, to a true JavaScript array:

```
var pArray = $A(document.getElementsByTagName("p"));
```

- The win here is that the array is also extended to include all the Prototype **Array** methods – we’ll see these in action later in the chapter.

String Functions

- **Prototype** provides enhanced string manipulation through a set of global functions:
 - **stripTags** leaves the character content but removes all markup from a string.
 - **escapeHTML** translates markup characters to the corresponding XML/HTML character entities – for instance `<` to `<`;
 - **toQueryParams** converts an object to a query string representing each of the object's properties as an HTTP request parameter.
 - **camelize** converts text to a “camel case” identifier.

- We'll get a bit more exercise with Prototype by refactoring one of the pages of the Flights application.
 - Do your work in **Demos/Prototype**.
 - The completed demo is found in **Examples/Flights/Ajax/Step3**.

1. Open **docroot/airports.jsp** and find the `<script>` near the bottom of the page.

2. Remove the global variable **baseId**. With Prototype (and the use of JavaScript closures) we have better control over this value and don't have to leave it hanging out there as a global.

3. Instead, give the **sendRequest** function a parameter of that name:

```
function searchForAirport(baseId)
```

4. At the top of **sendRequest**, add an **options** variable, which will be passed to a new **Ajax.Request** object presently. For starters, set the **method** and **parameters** properties.

```
var options =  
{  
  method: "GET",  
  parameters: "?address=" +  
              $(baseId + "-location").value  
}
```

- **parameters** is just what **queryString** was for the XHR implementation; you can copy that string expression and get rid of the **queryString** variable.

5. Delete the rest of the original **sendRequest** implementation.

6. Add a property **onSuccess** to the **options** object, as follows:

```
parameters: "?address=" +
            $(baseId + "-location").value,
onSuccess:
  function(XHR)
  {
    $(baseId + "-code").value =
      XHR.responseText;
  }
};
```

- The function body is just the meat of the original **receiveAirportCode** function, with the **readyState** and **status** checking left out.

7. Add a similar **onFailure** function – the only difference is that it sets “???” into the same HTML element:

```
},
onFailure:
  function(XHR)
  {
    $(baseId + "-code").value = "???" ;
  }
};
```

8. Remove the original **receiveAirportCode** function.
9. At the bottom of your new **sendRequest** implementation, create your request object:

```
...
new Ajax.Request("AirportSearch", options);
}
```

10. Find the two invocations of **sendRequest** in the HTML body, and refactor them to take advantage of the new **baseUrl** parameter. The first of these two occurrences is shown below:

```
<input
  type="button"
  id="origin-cmd"
  class="tableButton"
  value="Search"
  onclick="searchForAirport ('origin');"
/>
```

11. Change the script reference just above the main `<script>` to bring Prototype into play:

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

12. Build and deploy the Flights application anew:

`ant`

13. Test at the following URL, and be sure that you're still getting the airport-code search behavior as before:

`http://localhost:8080/Flights`

- Enter a city name or fragment thereof, and click **Search**:



The screenshot shows a web application titled "Airline Reservations" with a blue sky background. It features two search rows. The first row is labeled "Flying from:" and contains a text input field with "BOS", a label "(or search for airports near:", a text input field with "osto", and a "Search" button. The second row is labeled "Flying to:" and contains a text input field with "LAX", a label "(or search for airports near:", a text input field with "Angel", and a "Search" button. A mouse cursor is pointing at the second "Search" button. Below these rows is a large "Find Flights" button.

Algorithmic Programming

- Prototype extends the normal JavaScript array as the **Array** type.
- This brings a nice set of utility functions into play, such as:
 - **find**
 - **findAll**
 - **each**
- These and several others accept functions as parameters.
- This fosters an **algorithmic programming** style by which the process of iterating over the array is separated from the action that should be taken on a given array element.
 - Closures make algorithmic programming quite natural in JavaScript; in many other OO languages it's possible, but essentially artificial, for lack of a good way to pass behavior definitions (functions) as a method argument.

- In **Examples/Table** we experiment with algorithmic programming in Prototype.
- **index.html** displays a simple table of 100 numbers, 10-by-10.
- A `<script>` includes three functions that operate on the table.
- **highlightPrimes** finds all `<td>` tags in the document, and tests each to see if its value is a prime number:

```
function highlightPrimes()
{
  var cells = document.getElementsByTagName("td");
  $A(cells).findAll(function(element)
  {
    var number = parseInt(element.innerHTML);
    for (var factor = 2;
         factor < number / 2; ++factor)
      if (number % factor == 0)
        return false;
    return true;
  })
}
```

- Note the use of the **\$A** function to convert a DOM **NodeList** to a JavaScript array, so that we can call the **findAll** function.
- The resulting node list is processed, so that each table cell is re-styled to bold text and a new background color:

```
.each(function(element)
{
  element.style.fontWeight = "bold";
  element.style.background = "#cff";
});
}
```

- **markFactorsOf100** takes a similar approach – using the same algorithms but plugging in different filter criteria and styling:

```
function markFactorsOf100()
{
  var cells = document.getElementsByTagName("td");
  $A(cells).findAll(function(element)
  {
    return 100 % parseInt(element.innerHTML) == 0;
  })
  .each(function(element)
  {
    element.innerHTML = "F";
    element.style.fontWeight = "bold";
    element.style.color = "#080";
  });
}
```

- Finally, **reset** just puts the styling and content back for all cells:

```
function reset()
{
  var number = -1;
  var cells = document.getElementsByTagName("td");
  $A(cells).each(function(element)
  {
    element.innerHTML = ++number;
    element.style.fontWeight = "normal";
    element.style.color = "#000";
    element.style.background = "#fff";
  });
}
```

Table Transformations

EXAMPLE

- Each of three form buttons triggers a different function.
- Test the page in your browser and see the effects:

Algorithmic Programming

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

Highlight Primes Mark Factors of 100 Reset

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

Highlight Primes Mark Factors of 100 Reset

Table Transformations

EXAMPLE

0	F	F	3	F	F	6	7	8	9
F	11	12	13	14	15	16	17	18	19
F	21	22	23	24	F	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
F	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

Highlight Primes Mark Factors of 100 Reset

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

Highlight Primes Mark Factors of 100 Reset

Suggested time: 30 minutes

In this lab you will refactor the **flights.jsp** page from the Flights application to use Prototype. Some of this will be similar to the previous demo in which we refactored **airports.jsp**. But you will also take advantage of Prototype's **Array** methods to process the user's radio-button selections more elegantly.

Detailed instructions are found at the end of the chapter.

Evaluated Only

SUMMARY

- JavaScript, the DOM, CSS, and XHR give us the critical tools to build effective and attractive user interfaces.
- But script libraries such as Prototype fill a critical void:
 - With raw JavaScript, the programming model is awkward at best.
 - Browsers are not consistent in their implementations of the DOM, CSS, or XHR.
- Prototype’s Ajax support is simple and elegant – like much of the rest of the library.
 - It takes the truly boilerplate coding off of our hands.
 - It implements consensus best-practice, such as thread safety via distinct XHR instances.
 - It still allows customization and flexibility.
- Prototype is quite lightweight, by itself; but remember that it is intentionally “thin” and meant to support additional layers, such as **script.aculo.us**.
- It’s convenience functions, aliases, and especially the marvelous use of JavaScript prototypes to extend basic types such as **Element** and **Array**, make for a sort of “second skin” over JavaScript itself – a smoother, higher-level programming model.