THE EXPERT'S VOICE® IN WEB DEVELOPMENT

Practical Prototype and script.aculo.us

Learn every major facet of Prototype and script.aculo.us from one of the core team developers.

Andrew Dupont

Apress®

Practical Prototype and script.aculo.us

Andrew Dupont

Apress®

Practical Prototype and script.aculo.us

Copyright © 2008 by Andrew Dupont

ISBN-13 (pbk): 978-1-59059-919-8

ISBN-10 (pbk): 1-59059-919-5

ISBN-13 (electronic): 978-1-4302-0502-9

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editors: Clay Andres, Tony Campbell, Jason Gilmore, Chris Mills
Technical Reviewer: Aaron Gustafson
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Beth Christmas
Copy Editor: Damon Larson
Associate Production Director: Kari Brooks-Copony
Production Editor: Laura Esterman
Compositor: Linda Weidemann, Wolf Creek Press
Proofreader: Erin Poe
Indexer: Broccoli Information Management
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `http://www.apress.com/info/bulksales`.

The source code for this book is available to readers at `http://www.apress.com`.

Contents at a Glance

Contents

About the Author

ANDREW DUPONT is a UI developer living and working in Austin, Texas. He is a member of the core development team for Prototype, the popular JavaScript toolkit. He has contributed to Prototype in many different ways: writing code and documentation, offering support, and evangelizing to colleagues. In addition, Andrew has spoken about JavaScript at South by Southwest Interactive and other tech industry conferences.

Andrew received liberal arts and journalism degrees from the University of Texas at Austin. He occasionally attended classes, but much preferred the time between classes, during which he experimented with web design and learned about the emerging web standards movement.

About the Technical Reviewer

After getting hooked on the Web in 1996 and spending several years pushing pixels and bits for the likes of IBM and Konica Minolta, **AARON GUSTAFSON** founded his own web consultancy: Easy! Designs. Aaron is a member of the Web Standards Project (WaSP) and the Guild of Accessible Web Designers (GAWDS). He also serves as a technical editor for A List Apart, is a contributing writer for Digital Web Magazine and MSDN, and has built a small library of writing and editing credits in the print world, including contributions to *AdvancED DOM Scripting* (friends of ED, 2007), *Accelerated DOM Scripting with Ajax, APIs, and Libraries* (Apress, 2007), and *Web Design in a Nutshell, Third Edition* (O'Reilly, 2006). Aaron has graced the stage at numerous conferences, including An Event Apart, COMDEX, South by Southwest, the Ajax Experience, and Web Directions, and he is frequently called on to provide web standards training in both the public and private sectors.

He blogs at `http://easy-reader.net`.

Acknowledgments

A number of forces conspired to help me finish this book, despite my best efforts not to. Aaron Gustafson was a great help as technical editor and gave thoughtful code critiques. The patient, stoic folks at Apress gave me constructive encouragement throughout the slow, arduous process. Christophe Porteneuve, having written a book on the same subject, gave me tips on several occasions.

I am also grateful to Sam Stephenson, both for creating Prototype and for inviting me to be a part of Prototype Core. Other team members gave critical moral support: Thomas Fuchs, Tobie Langel, Mislav Marohnic, and Justin Palmer.

Objects, as well as people, were instrumental in the completion of this book: a MacBook Pro, TextMate, Parallels Desktop, and sugar-free Red Bull. I thank them for their support.

Introduction

I wrote this book for people who have some experience with JavaScript and no experience with Prototype. I mean for "experience with JavaScript" to cast a wide net: you may love JavaScript, or hate it, or love the language but hate browser scripting, or love both, or hate both.

Because the book assumes some JavaScript experience, it does not cover the most basic parts of the JavaScript language itself, nor does it cover the DOM. There are many books that can get you started on that path, but the best free resource is Quirks Mode (`www.quirksmode.org`), the authoritative and exhaustive reference created by Peter-Paul Koch.

This book is also meant to appeal to those who have some experience with Prototype but don't consider themselves experts. Many have worked with Prototype indirectly through Ruby on Rails or a similar framework. Many have used a third-party script that depended on Prototype, but treated the code as a black box.

In the first chapter of the book, we'll look at some aspects of the JavaScript language that novice users may not know about. Feel free to spend as much time on Chapter 1 as you need, since it's crucial that you understand these concepts if you want to use Prototype effectively.

The screenshots in this book show Firefox running on Windows XP, but the code examples are designed to work in all major browsers and on all major platforms. Prototype boasts official support for Firefox (versions 1.5 and above), Internet Explorer (6 and above), Safari (2 and above), and Opera (9.2 and above).

I welcome your feedback, observations, and ridicule. I can be reached at `book@andrewdupont.net`.

PART 1

Prototype

CHAPTER 1

What You Should Know About Prototype, JavaScript, and the DOM

Before jumping into the deep end, you should learn about where Prototype comes from—its purpose, origin, and philosophy. We'll also discuss what differentiates Prototype from other libraries.

First, though, we need to make sure we're on the same page. This book assumes a basic familiarity with JavaScript and the DOM, but that's a vague prerequisite, and JavaScript is a language both broad and deep.

In case you need a refresher, here's a crash course in topics that will be built upon in the chapters to follow.

About JavaScript

George Orwell once wrote that writing JavaScript "is a horrible, exhausting struggle, like a long bout of some painful illness. One would never undertake such a thing if one were not driven on by some demon whom one can neither resist nor understand."

OK, that's a lie. He actually said that about writing *books*. Keep in mind, though, that JavaScript had not yet been created in Orwell's time, and that a modern-day Orwell might have eschewed prose in favor of *programming*, a far higher artistic pursuit.

I feel this way about JavaScript, at the very least. It's a brilliant language with very public flaws. It was created hastily and standardized prematurely. The JavaScript environments within browsers vary wildly in spec compliance, language features, and speed. It's a language whose ideals are compromised by the imperfect state of today's Web.

We'll talk about ways to mitigate these flaws. But first let's look at some of the things that make JavaScript brilliant.

Everything Is an Object

The sooner you embrace this concept, the more quickly you'll understand Prototype: everything in JavaScript is an object. This has several different meanings and several different implications, which are outlined in the following subsections.

All Data Types Have Instance Methods

Like other languages that embrace object orientation, every object can have instance methods. This allows for flexible syntax and makes code easier to read.

```
["foo", "bar", "baz"].join(' ');
//=> "foo bar baz"
"foo bar baz".split(' ');
//-> ["foo", "bar", "baz"]
```

It also ensures that functions don't clutter up the global namespace. What's the use of a generic `join` function, for instance, if it works only on arrays? Or a generic `split` that works only on strings?

All Data Types Inherit from Object

JavaScript boasts half a dozen native data types: `Object`, `String`, `Array`, `RegExp` (for regular expressions), `Boolean` (true or false), and `Date`.

I place `Object` first because it's the root data type. When I say that everything in JavaScript is an object, I also mean that everything in JavaScript is an `Object`. Confused? Let me explain.

`Object` can be thought of as a *blank* data type, the empty canvas that all other types start with. There is nothing `Object` does that another type can't do, but then that's the point: `Objects` can bend to your will.

But back to the main point: Everything in JavaScript is an `Object`. We can verify this with the `instanceof` operator on some core JavaScript data types:

```
Array instanceof Object;
//-> true
RegExp instanceof Object;
//-> true
Date instanceof Object;
//-> true
String instanceof Object;
//-> true
Function instanceof Object;
//-> true
```

These are the constructors for data types. But instances of these data types also inherit from `Object`:

```
['foo', 'bar'] instanceof Object; // (Array literal)
//-> true
/.*/ instanceof Object;           // (RegExp literal)
//-> true
new Date instanceof Object;
//-> true
```

But here's where it gets tricky. The typical "primitives" in a programming language—strings, numbers, and Booleans—are *both* primitives *and* objects in JavaScript. They're treated as one or the other depending on context.

```
"foo" instanceof Object;      //-> false
new String instanceof Object; //-> true

5 instanceof Object;             //-> false
new Number(5) instanceof Object; //-> true

true instanceof Object;              //-> false
new Boolean(true) instanceof Object; //-> true
```

This is confusing at first, but ends up being quite helpful. It allows these types to behave like primitives when they need to (they're passed by value, instead of by reference), but they can still reap the benefits of JavaScript's object-oriented functionality (instance methods, local scope, etc.).

All Objects Have Prototypes

Although JavaScript is most certainly object oriented, it's likely not the sort of object orientation you're used to. Strictly speaking, there is no concept of a "class" in JavaScript—instead of outlining an abstract definition of an object, you can simply make a copy of an existing object. Remember what we just found out:

```
Array instanceof Object;     //-> true
new Array instanceof Object; //-> true
```

There is no technical distinction between `Array` and instances of `Array`. You can do the same sorts of things to either one.

Because there are no classes in JavaScript, inheritance is based on the concept of prototypes. Each object has its own `prototype` property, which serves as a template for any new instances (copies) made from that object.

This behavior isn't limited to user-defined objects—it can be applied to the built-ins as well. It's quite simple to add instance methods to arrays, strings, or any other native types:

```
Array.prototype.double = function() {
  var newArray = [];
  // inside this function, "this" refers to the array
  for (var i = 0, length = this.length; i < length; i++) {
    newArray.push(this[i]);
    newArray.push(this[i]);
  }
  return newArray;
};

var exampleArray = new Array(1, 2, 3);
// inherits all the properties of Array.prototype
exampleArray.double();
//-> [1, 1, 2, 2, 3, 3]
```

Since `exampleArray` is an instance of `Array`, it looks to `Array.prototype` for inheritance. So instead of defining a double function in the global namespace, we can define it as an instance method on arrays. This is a big win. It makes user-written code mesh better with native code, it reduces verbosity and the risk of naming collisions, and it lets the code demonstrate its meaning far more plainly.

Those of you more at home with class-based inheritance need not bother with any of this: Prototype includes a more conventional system of OOP that allows for the distinction between classes and instances—*and* makes dealing with prototypes unnecessary, if that's your bag. Prototypal inheritance isn't something you need to be afraid of, but you don't need to invite it over for a beer, either.

Any Object Can Have Arbitrary Properties Set

All objects are mutable in JavaScript: they can be changed, augmented, and pruned at any time. You can assign any property to an object, even one that already exists.

```
var object = new Object();
object.foo = "foo";
object.bar = "bar";
for (var i in object) console.log(i);
//-> foo
//-> bar
```

```
// Numbers have a native "toString" method
var number = 5;
number.toString(); //-> "5"
Number.prototype.toString = function() {
  return String(this + 1);
};
number.toString(); //-> "6"
```

That last example should frighten you—yes, JavaScript's dynamism lets you do vacuously stupid things. Remember that you're working in a shared environment—don't do things that will wantonly break other scripts on the page.

ABOUT CONSOLE.LOG

Many of the code examples in this book will use `console.log`, a function for writing to the browser console. Supported in Firefox (through the excellent Firebug extension) and Safari, it's far less invasive than the popular `alert` function—there are no pop-up dialogs to dismiss. Think of it as JavaScript's `print` statement. You won't ever use it in production code, but it's handy for learning and debugging.

Even Functions Are First-Class Objects

JavaScript functions are not the downtrodden plebeians of PHP or Java. They enjoy full citizenship. A function can be assigned to a variable, can be passed as an argument in another function, and can be returned from a function. (You can have a function that accepts a function as an argument and *returns* a function! Madness!)

You're probably familiar with this syntax for defining functions:

```
function lambda() {
  return "foo";
}
```

But here's a lesser-used syntax that's nearly equivalent:

```
var lambda = function() {
  return "foo";
};
```

In both examples, you're describing a function and assigning it a name of `lambda`. In both examples, the function can be called by writing `lambda()`. And in both examples, you can use `lambda` as flexibly as you'd use any other data type.

The second example uses a *function literal* (or *anonymous function*): a special, flexible syntax for creating a new function. Just like strings (quotation marks), arrays (brackets), objects (braces), and regular expressions (forward slashes), functions have a literal notation. (Notice also how the second example ends with a semicolon after the closing brace, since the braces aren't a control structure like they are in the first example.)

Prototype uses function literals all over the place. You'll see them passed as arguments into methods for working with collections. You'll see them used as callbacks for Ajax requests. You'll see them used to define methods on objects.

About the DOM

The Document Object Model (DOM), an interaction model defined by an ambitious and far-reaching set of W3C specifications, lays out the ideal set of objects, methods, and properties to change an HTML or XML document programmatically. The segmentation of the DOM into many different "levels" and "modules" means that the browsers you write code for will have wildly varying levels of DOM support.

There are four major implementations of JavaScript and the DOM: Mozilla's Spider-Monkey, Internet Explorer's JScript, Opera's linear_b, and Safari's JavaScriptCore. While each has its strengths and weaknesses, you will discover very quickly that they are not created equal.

It's Hard to Write Multiplatform JavaScript

Most developers have the luxury of writing code toward one target: a compiler or canonical interpreter that implements the given language perfectly. Writing JavaScript for modern browsers will make you realize how much this luxury is taken for granted. It can be an excruciating, wailing-and-teeth-grinding experience.

Most of these pains will come not from the language itself, but from what other languages would call the standard library—the APIs we have that connect JavaScript to a web environment.

One nagging pain, for instance, comes from the different levels of DOM support. Let's call these "capabilities." They're things that some browsers can do, but that others cannot.

A sharper, more chronic pain comes from outright incompatibilities between browsers. The most notable is between Internet Explorer and all other browsers. In all modern versions of Internet Explorer (currently versions 6 and 7), DOM support is minimal; the gaps are filled in by Internet Explorer's legacy API, which predates the DOM.

The *most agonizing* pain, akin to a daily kick in the stomach, comes from what I'll call "quirks." Quirks are a diplomatic term for *bugs*: misimplementations of the DOM spec, memory leaks, and other irrational behavior. These are the largest obstacles to developer happiness; they're hard to diagnose and hard to work around. They'll leave you staring at your monitor past midnight, bleary eyed and out of ideas.

JavaScript frameworks exist to smooth out these cracks. Wherever possible, Prototype provides a unified API that handles all the ugliness under the hood. Prototype won't solve all your problems, but it can definitely make the process of writing JavaScript far less unpleasant.

It's Hard to Debug Multiplatform JavaScript

The tools that server-side developers have come to rely on—loggers, debuggers, and the like—are not readily available on the client side. Among the major browsers, you'll find that some are far more helpful than others when you need to fix problems in your code.

The best developer tool for JavaScript authors is Firebug, an extension for Mozilla Firefox. Firebug is a dream come true: it provides a logger, a debugger, a DOM inspector, a CSS editor, a rendered source tree, and a code profiler. It is the single biggest reason why Firefox is the browser of choice for JavaScript development.

Of course, you'll also need to test in Internet Explorer, Opera, and Safari. But you'll likely find it easiest to use Firefox while you write your code. When you get it working, you can test in other browsers to find out if you need to make changes.

About This Book

You won't need a legend or translation table to read this book. But there are a few things you should know up front so that you can enjoy this book to the fullest.

Firefox Is Used for Nearly All Examples

The things that make Firefox the best browser for client-side web development also make it the best browser for an interactive teaching process. In this book, we'll spend a lot of time in Firefox.

One major reason is Firebug (`www.getfirebug.com/`), the definitive Firefox extension for web developers. You'll come to love how the Firebug console speeds up trial-and-error development and lets you learn by experimentation. But another reason is the stability of SpiderMonkey (Mozilla's JavaScript engine) and its broad support for the DOM.

Of course, this book is also about writing JavaScript in the real world, so we'll also be testing our examples in other browsers. But we'll be following the approach outlined earlier: develop in Firefox, and *then* make it work everywhere else.

First Theory, Then Practice

You'll notice that the first half of the book is heavy on *abstract* examples, and the second half is heavy on *concrete* examples. This division mirrors the division between Prototype and script.aculo.us: the former builds a rich set of APIs, while the latter uses those APIs to address specific UI needs.

Even so, there are a few meanderings. Each chapter of Part 1 will highlight specific ways that Prototype can improve the code you write. Each chapter of Part 2 will give you specific ways to apply the examples to your own web application.

About Prototype

The origins of Prototype are shrouded in mystery—like Stonehenge, the Dead Sea scrolls, and *the universe itself*. Well, that may be overstating it.

Version 1.0 of Prototype was released in March of 2005. An early README file described the framework this way:

> *Prototype is a JavaScript library that aims to ease development of dynamic web applications. Its development is driven heavily by the Ruby on Rails framework, but it can be used in any environment.*

Though Prototype has evolved considerably over the last three years, this summary is still stunningly applicable.

Prototype's Philosophy

The first version of Prototype was very small. It contained some of the convenience methods that Prototype has become famous for, but most of its 335 lines of code were designed to simplify Ajax and form interaction.

Since then, its scope has widened, but its philosophy has not wavered. There is neither a manifesto nor a mission statement, but a handful of principles guide the project:

> *The principle of least surprise*: Popularized by Yukihiro Matsumoto, the creator of Ruby, it states that a language, framework, or library should always do the "least surprising" thing; it should behave the way the user expects it to. Prototype is meant to be intuitive and easy to learn.

The 80 percent rule: As an informal corollary to the famous 80/20 rule, Prototype pledges to solve the common problems that are shared by the vast majority of developers. Proposed additions to the framework need to demonstrate widespread need. But the common 80 percent shouldn't exclude the unique 20 percent, either—Prototype is designed to be easily extensible and customizable.

Self-documenting code: In the README file of Prototype 1.0, Sam Stephenson states, "Prototype is embarrassingly lacking in documentation." In lieu of providing plain English documentation, Sam did the next best thing: he made the code itself easy to read and understand. These days, Prototype enjoys exhaustive documentation, both official and unofficial. But readability, intuitive naming schemes, and cleanliness are still virtues of Prototype's source code.

Prototype's Purpose and Scope

Prototype is all about the abstract rather than the concrete. It isn't a widget toolkit, or a graphing library, or a form validation utility—but all these things can be built *atop* Prototype. Most famously, script.aculo.us (which we'll cover in Part 2) uses Prototype as a basis for a rich library of effects and UI controls.

Many JavaScript toolkits, script.aculo.us included, allow you to do complex things without knowing much JavaScript. That's not what Prototype is for. If you don't want to get elbow-deep in JavaScript development, you won't like Prototype very much.

Prototype's Web Site

You can get Prototype, learn more about it, and read current project news at Prototype's web site, at `www.prototypejs.org/`. You'll also learn where to go if you get stuck, how to contribute to Prototype, and who uses Prototype in the real world.

Prototype's download page always features the latest *stable* version, plus instructions on how to build a bleeding-edge version if you're feeling particularly daring.

Contributing to Prototype

Prototype is an open source project that bears the MIT License. In other words, do whatever you like with the code and use it wherever you please, as long as you give credit. "Giving credit" is done automatically—at the top of the source code is a long comment describing the framework and its authors.

Like other open source projects, Prototype relies heavily on the developer community. Bug reports and patches are enthusiastically welcomed. The code itself lies in the

Ruby on Rails Subversion repository; you can browse the source code and the outstanding issues at `http://dev.rubyonrails.org/`.

Proposed additions to Prototype are discussed on the Prototype Core mailing list. Instructions for joining this list can be found on Prototype's web site.

Getting Started with Prototype

Prototype can be thought of as a JavaScript supplement. Most languages have a "standard library" of code that simplifies common tasks. For instance, Ruby developers can use `require 'rexml'` to help with XML parsing, and Python developers can use `import gzip` to enable gzip compression in their scripts.

Similarly, you can include Prototype on a web page by loading the `prototype.js` script near the beginning of the file. So let's try it out.

Creating a Page

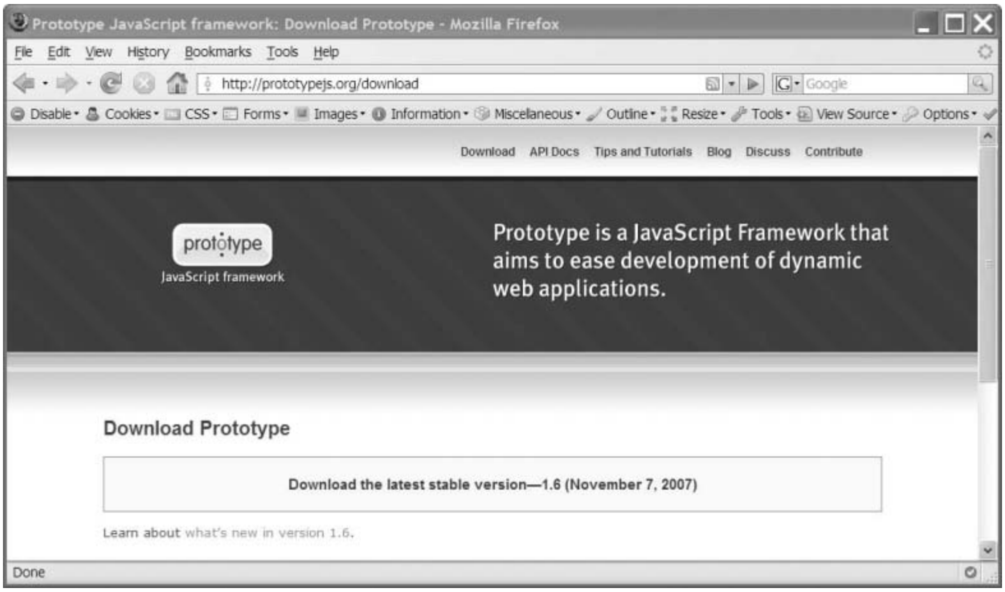
First, create an empty folder to hold your HTML and JavaScript files. Using your favorite text editor, create a boilerplate HTML file like so:

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

  <body>
    <h1>Blank Page</h1>
  </body>
</html>
```

Not all that interesting, but it suits our needs. Save this file as `index.html`.

Downloading Prototype

Fire up a web browser and go to `http://prototypejs.org/`. Here you'll see an overview of the Prototype library and a prominent Download link. Click the link, and then click the green box to get the latest version of Prototype (version 1.6.0.1 at press time). You should see a page that looks like Figure 1-1.


Prototype JavaScript framework: Download Prototype - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://prototypejs.org/download
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tools View Source Options
Download API Docs Tips and Tutorials Blog Discuss Contribute
prototype
JavaScript framework
Done

Prototype is a JavaScript Framework that aims to ease development of dynamic web applications.

Download Prototype

Download the latest stable version—1.6 (November 7, 2007)

Learn about what's new in version 1.6.

Figure 1-1. *The download page on Prototype's web site*

Clicking the link will likely take you to a page full of JavaScript code. This is Prototype. You can save a copy of this file through the File menu; you can also select all the text and paste it into an empty text file. Either way, save the file as `prototype.js` in the same directory in which you placed `index.html`.

Including Prototype

To load Prototype into the page environment, you need to include a `script` tag. Add the following line to `index.html`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.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="prototype.js"></script>
    <title>Blank Page</title>
  </head>
  <body>
    <h1>Blank Page</h1>
  </body>
</html>
```

Keep a few things in mind about `script` tags:

- You can include JavaScript on a page directly with the `script` tag, but it's usually a better idea to place your code in a file that gets loaded onto the page with the `src` attribute instead. It makes the code more portable.
- Scripts are evaluated in the order in which they occur in the page. For this reason, any scripts you write that rely on Prototype *must* be included *after* `prototype.js` is loaded. For this reason, try to ensure that `prototype.js` is the very first script loaded on any page.
- We're placing this `script` tag in the head of the HTML, the same place we'd put style sheets and define other page metadata. Because the code is evaluated as soon as it's parsed, it's a bad idea for scripts you include to take *immediate action*, since they'd be acting on a document that's only partially rendered. Instead, have your scripts define functions, and then set up those functions to run on page load or as the result of user action.

Testing It Out

Before you take a look at this page, make sure you've installed Mozilla Firefox and the Firebug extension. If you haven't, you can visit two easy-to-remember URLs: `http://getfirefox.com` and `http://getfirebug.com`. Both tools are free, easy to install, and available on all major platforms.

Save `index.html`, and then open it in Firefox. (You can simply drag the file into a browser window, if you like.) It won't look like much, but since you've included Prototype on this page, you can use it as a playground.

Click the icon at the bottom-right side of the screen; it should be a green circle with a check box. (If it's a gray circle, you should click it and select "Enable Firebug for local files.") Firebug will pop up a windowpane with several tabs. The one you're interested in right now is Console; it acts as a JavaScript shell, letting you run code in the context of the current page (see Figure 1-2).

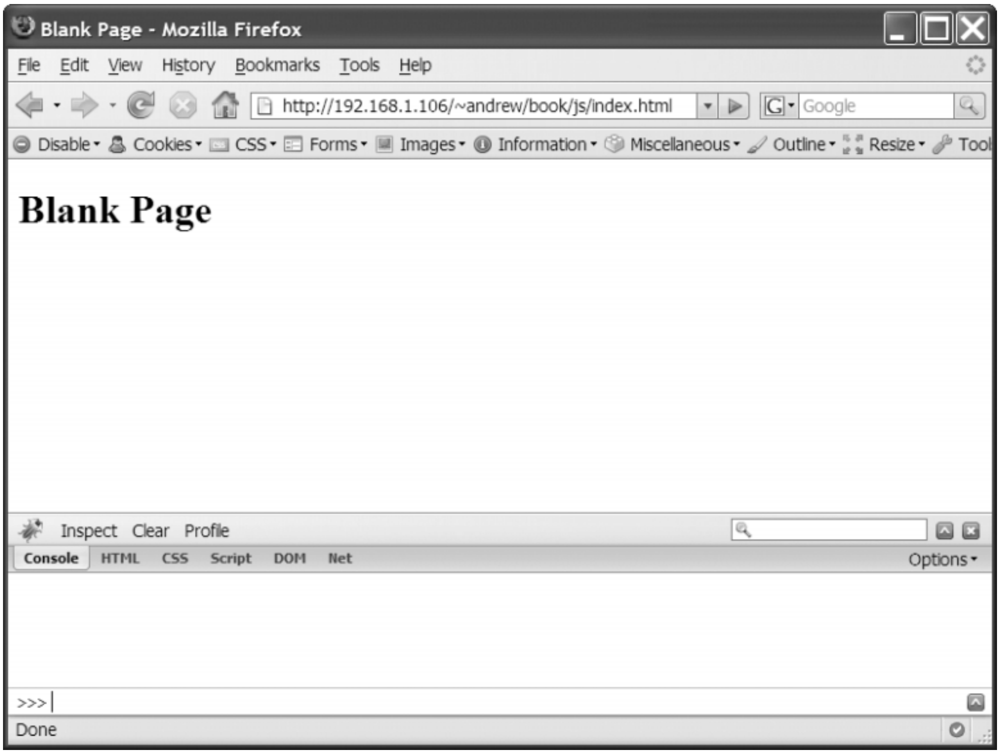
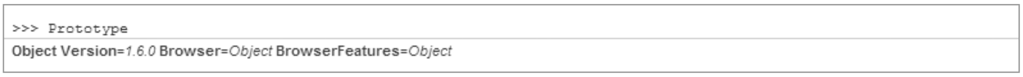

Blank Page - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/book/js/index.html
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tool
Blank Page
Inspect Clear Profile
Console HTML CSS Script DOM Net
Options
>>>
Done

Figure 1-2. *Firebug lets you run code interactively on the current page.*

Let's try a couple of statements. Type in `Prototype`, then press Enter (see Figure 1-3).



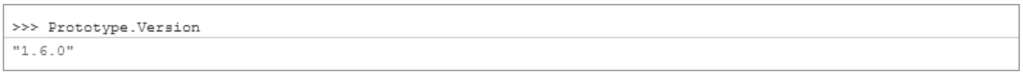
```
>>> Prototype
Object Version=1.6.0 Browser=Object BrowserFeatures=Object
```

Figure 1-3. *The Prototype object in Firebug*

The Prototype library defines a global object called `Prototype`, which holds metadata about the library itself. The most reliable way to determine whether the Prototype library has been loaded into a page is to check the type of the `Prototype` object; if it hasn't been defined on the page, then the type will be *undefined*. A library that depends on Prototype might do something like this:

```
// raise an error if Prototype isn't loaded
if (typeof Prototype === "undefined") {
  throw new Error("This script relies on the Prototype JavaScript framework.");
}
```

You'll notice that one of the properties is called `Version`. Type in `Prototype.Version` and press Enter (see Figure 1-4).



```
>>> Prototype.Version
"1.6.0"
```

Figure 1-4. *The version number of Prototype in string form*

Naturally, `Prototype.Version` refers to the version of the library that's been loaded onto the page. A lot of the code we'll be writing in this book won't work in versions of Prototype prior to 1.6. Many libraries, script.aculo.us among them, check this property and raise a helpful error message if the included Prototype version is too old.

Summary

We've just completed a high-level overview of JavaScript, the browser environment, and the Prototype project. Now we're ready to jump headfirst into the code. On the other side of this chapter break, we'll pick back up with a quick survey of the most important Prototype functions and methods.

CHAPTER 2

Prototype Basics

JavaScript libraries don't start out as libraries. They start out as "helper" scripts.

It's possible, but impractical, to do DOM scripting without the support of a library. Little things will start to annoy you from day one. You'll get tired of typing out `document.getElementById`, so you'll write an alias. Then you'll notice that Internet Explorer and Firefox have different event systems, so you'll write a wrapper function for adding events. Then you'll become irritated with a specific oversight of the language itself, so you'll use JavaScript's extensibility to write some code to correct it.

Then one day you'll notice your "helper script" is 35 KB. When organized and divided into sections, you'll realize you've got a library on your hands. All JavaScript libraries start out as caulk between the cracks.

For this reason, a lesson in using Prototype begins not with an in-depth look at any particular portion, but rather with a whirlwind tour of many of the problem-solving constructs you'll use most often. In this chapter we'll take that tour.

Getting Started

Let's keep using the web page we wrote in the previous chapter. Open up `index.html` and add some content to the page's body:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.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="prototype.js"></script>
    <title>Blank Page</title>
  </head>
  <body>
    <h1>Blank Page</h1>
    <ul id="menu">
      <li id="nav_home" class="current"><a href="/">Home</a></li>
      <li id="nav_archives"><a href="/archives">Archives</a></li>
      <li id="nav_contact"><a href="/contact">Contact Me</a></li>
      <li id="nav_google"><a href="http://google.com"
       rel="external">Google</a></li>
    </ul>
  </body>
</html>
```

We're adding an unordered list with links as list items—the typical structure for a web site's navigation menu. Each `li` element has an ID. The final `a` element has a `rel` attribute with a value of `external`, since it links to an external site. This convention will be quite useful later on.

We'll add more markup to this page over the course of the chapter, but this is enough for now. Nearly all of the code examples from this chapter can be used on this page with the Firebug shell, so feel free to follow along.

The $ Function

DOM methods have intentionally verbose names for clarity's sake, but all that typing gets tiresome very quickly. Prototype's solution is simple but powerful: it aliases the oft-used DOM method `document.getElementById` to a function simply named `$`. For example, instead of

```
document.getElementById('menu'); //-> <ul id="menu">
```

with Prototype, you can write

```
$('menu'); //-> <ul id="menu">
```

Like `document.getElementById`, `$` finds the element on the page that has the given `id` attribute.

Why is it called `$`? Because you'll use it often enough that it needs to have a short name. It will be the function you'll use most often when scripting with Prototype. In addition, the dollar sign is a valid character in object names and has little chance of having the same name as another function on the page.

It's far more than a simple alias, though. There are several things you can do with `$` that you can't do with `document.getElementById`.

$ Can Take Either Strings or Nodes

If the argument passed to `$` is a string, it will look for an element in the document whose ID matches the string. If it's passed an existing node reference, though, it will return that same node. In other words, `$` lets you deal with string ID pointers and DOM node references nearly identically. For example

```
var menuElement = document.getElementById('menu');
Element.remove(menuElement);
// can also be written as...
Element.remove('menu');
```

Either way, the element is removed from the page. Why? Listing 2-1 shows how `Element.remove` is defined in the Prototype source code.

Listing 2-1. *Prototype Source Code*

```
Element.remove = function(element) {
  element = $(element);
  element.parentNode.removeChild(element);
  return element;
};
```

The highlighted line is used quite often in Prototype. If the `element` argument is a string, `$` converts it to a DOM node; if it's a node already, then it just gets passed back. It makes code very flexible at a very small cost.

All of Prototype's own DOM manipulation methods use `$` internally. So any argument in any Prototype method that expects a DOM node can receive *either* a node reference *or* a string reference to the node's ID.

$ Can Take Multiple Arguments

Normally, `$` returns a DOM node, just like `document.getElementById`. It returns just one node because an ID is supposed to be unique on a page, so if the browser's DOM engine finds an element with the given ID, it can assume that's the only such element on the page.

That's why `document.getElementById` can take only one argument. But `$` can take *any* number of arguments. If passed just one, it will return a node as expected; but if passed more than one, it will return an *array* of nodes.

```
var navItems = [document.getElementById('nav_home'),
               document.getElementById('nav_archives'),
               document.getElementById('nav_contact')];
//-> [<li id="nav_home">, <li id="nav_archives">, <li id="nav_contact">]

var navItems = $('nav_home', 'nav_archives', 'nav_contact');
//-> [<li id="nav_home">, <li id="nav_archives">, <li id="nav_contact">]
```

Prototype's constructs for working with DOM collections represent a large portion of its power. In Chapter 3, you'll learn about how useful this can be.

$ Enhances DOM Nodes with Useful Stuff

Core JavaScript data types can be augmented with user-defined functions. All JavaScript objects, even built-ins, have a `prototype` property that contains methods that should be shared by all instances of that object. For instance, Prototype defines a method for stripping leading and trailing whitespace from strings:

```
String.prototype.strip = function() {
  return this.replace(/^\s+/, '').replace(/\s+$/, '');
};

"  Lorem ipsum dolor sit amet     ".strip();
//-> "Lorem ipsum dolor sit amet"
```

All strings get this method because they're all instances of the `String` object.

In an ideal world, it would be this easy to assign user-defined functions to DOM objects:

```
HTMLElement.prototype.hide = function() {
  this.style.display = 'none';
};
```

This example works in Firefox and Opera, but fails in some versions of Safari and all versions of Internet Explorer. There's a gray area here: the DOM API is designed to be language independent, with its JavaScript version just one of many possible implementations. So some browsers don't treat DOM objects like `HTMLElement` the same way as built-ins like `String` and `Array`. To get this sort of thing to work across browsers requires a bit of voodoo.

Prototype takes care of this behind the scenes by defining custom element methods on `HTMLElement.prototype` in browsers that support it, and copying these instance methods to nodes on demand in browsers that don't. Any Prototype method that returns DOM nodes will "extend" these nodes with instance methods to enable this handy syntactic sugar.

Once a node has been extended once, it does not need to be extended again. But to extend all nodes on page load would be prohibitively costly, so Prototype extends nodes on an as-needed basis.

Let's illustrate this in code:

```
var firstUL = document.getElementsByTagName('ul')[0];
firstUL.hide();
//-> Error: firstUL.hide is not a function
```

You'll get this error in Internet Explorer. A node must have been extended by Prototype before you can be sure it has these instance methods. So there are a few options here:

- Use the generic version of the method. Every instance method of a DOM node is also available on the `Element` object:

  ```
  var firstDiv = document.getElementsByTagName('ul')[0];
  Element.hide(firstUL);
  ```

- Instead of using the native DOM method, use a Prototype method that does the same thing.

  ```
  var firstUL = $$('div')[0];
  firstUL.hide();
  ```

- Extend the node just to be safe, using `Element.extend` or `$`.

  ```
  $(document.getElementsByTagName('ul')[0]).hide();
  ```

In other words, `$` isn't just an alias for `document.getElementById`, it's also an alias for `Element.extend`, the function that adds custom instance methods to DOM nodes. You'll learn much more about this in Chapter 6.

Object.extend: Painless Object Merging

The object literal is part of JavaScript's terseness and expressive power. It allows one to declare an object with any number of properties very easily.

```
var data = {
  height: "5ft 10in",
  weight: "205 lbs",
  skin:   "white",
  hair:   "brown",
  eyes:   "blue"
};
```

But in JavaScript, it's possible to add any number of properties to an existing object at any time. So what happens when we want to extend this object?

```
if (person.country == "USA") {
  data.socialSecurityNumber = "456-78-9012";
  data.stateOfResidence = "TX";
  data.standardTaxDeduction = true;
  data.zipCode = 78701;
}
```

We can't define new properties en masse—we have to define them one by one. It gets even more frustrating when extending built-in classes, as Prototype does:

```
String.prototype.strip = function() {
  // ...
};

String.prototype.gsub = function() {
  // ...
};

String.prototype.times = function() {
  // ...
};

String.prototype.toQueryParams = function() {
  // ...
};
```

This is a direct road to carpal tunnel syndrome. There's got to be a better way—we need a function for merging two different objects.

Prototype gives us `Object.extend`. It takes two arguments, `destination` and `source`, and both objects, and loops through all the properties of `source`, copying them over to `destination`. If the two objects have a property with the same name, then the one on `destination` takes precedence.

```
if (person.country == "USA") {
  Object.extend(data, {
    socialSecurityNumber: "456-78-9012",
    stateOfResidence:     "TX",
    standardTaxDeduction: true,
    zipCode:              78701
  });
}
```

Since objects are passed by reference, not value, the `source` object is modified in place.

`Object.extend` also solves our typing woes when extending built-ins:

```
Object.extend(String.prototype, {
  strip: function() {
    // ...
  },

  gsub: function() {
    // ...
  },

  times: function() {
    // ...
  },

  toQueryParams: function() {
    // ...
  }
});

for (var i in String.prototype)
  console.log(i);
//-> "strip", "gsub", "times", "toQueryParams"
```

That's one annoyance out of the way. This construct cuts down on redundancy, making code both smaller and easier to read. Prototype uses `Object.extend` all over the place internally: extending built-ins, "mixing in" interfaces, and merging default options with user-defined options.

WHY NOT USE OBJECT.PROTOTYPE.EXTEND?

If we were steadfastly abiding by JavaScript's object orientation, we'd define `Object.prototype.extend`, so that we could say the following:

```
var data = { height: "5ft 10in", hair: "brown" };
data.extend({
  socialSecurityNumber: "456-78-9012",
  stateOfResidence:     "TX"
});
```

This may appear to make things easier for us, but it will make things much harder elsewhere. Because properties defined on the prototypes of objects are enumerated in a `for...in` loop, augmenting `Object.prototype` would "break" hashes:

```
for (var property in data)
  console.log(property);
//-> "height", "hair", "socialSecurityNumber", "stateOfResidence", "extend"
```

There are ways around this, but they all involve changing the way we enumerate over objects. And we'd be breaking a convention that's relied upon by many other scripts that could conceivably exist in the same environment as Prototype. In the interest of "playing well with others," nearly all modern JavaScript libraries abide by a gentleman's agreement not to touch `Object.prototype`.

$A: Coercing Collections into Arrays

Oftentimes in JavaScript, you'll have to work with a collection that *seems* like an array but really isn't. The two major culprits are DOM `NodeLists` (returned by `getElementsByTagName` and other DOM methods) and the magic `arguments` variable within functions (which contains a collection of all the arguments passed to the function).

Both types of collections have numeric indices and a `length` property, just like arrays—but because they don't inherit from `Array`, they don't have the same methods that arrays have. For most developers, this discovery is sudden and confusing.

`$A` provides a quick way to get a true array from any collection. It iterates through the collection, pushes each item into an array, and returns that array.

The arguments Variable

When referenced within a function, `arguments` holds a collection of all the arguments passed to the function. It has numeric indices just like an array:

```
function printFirstArgument() {
  console.log(arguments[0]);
}
printFirstArgument('pancakes');
//-> "pancakes"
```

It *isn't* an array, though, as you'll learn when you try to use array methods on it.

```
function joinArguments() {
  return arguments.join(', ');
}
joinArguments('foo', 'bar', 'baz');
//-> Error: arguments.join is not a function
```

To use the `join` method, we first need to convert the `arguments` variable to an array:

```
function joinArguments() {
  return $A(arguments).join(', ');
}
joinArguments('foo', 'bar', 'baz');
//-> "foo, bar, baz"
```

DOM NodeLists

A DOM `NodeList` is the return value of any DOM method that fetches a collection of elements (most notably `getElementsByTagName`). Sadly, DOM `NodeLists` are nearly useless. They can't be constructed manually by the user. They can't be made to inherit from `Array`. And the same cross-browser issues that make it hard to extend `HTMLElement` also make it hard to extend `NodeList`.

Any Prototype method that returns a collection of DOM nodes will use an array. But native methods (like `getElementsByTagName`) and properties (like `childNodes`) will return a `NodeList`. Be sure to convert it into an array before you attempt to use array methods on it.

```
// WRONG:
var items = document.getElementsByTagName('li');
items = paragraphs.slice(1);
//-> Error: items.slice is not a function
```

```
// RIGHT:
var items = $A(document.getElementsByTagName('li'));
items = items.slice(1);
//-> (returns all list items except the first)
```

$$: Complex Node Queries

The richness of an HTML document is far beyond the querying capabilities of the basic DOM methods. What happens when we need to go beyond tag name queries and fetch elements by class name, attribute, or position in the document?

Cascading Style Sheets (CSS) got this right. CSS, for those of you who don't have design experience, is a declarative language for defining how elements look on a page. The structure of a CSS file consists of selectors, each with a certain number of rules (i.e., "The elements that match *this* selector should have *these* style rules."). A CSS file, if it were obsessively commented, might look like this:

```
body { /* the BODY tag */
  margin: 0;  /* no space outside the BODY */
  padding: 0; /* no space inside the BODY */
}

a { /* all A tags (links) */
  color: red; /* links are red instead of the default blue */
  text-decoration: none; /* links won't be underlined */
}

ul li { /* all LIs inside a UL */
  background-color: green;
}

ul#menu { /* the UL with the ID of "menu" */
  border: 1px dotted black; /* a dotted, 1-pixel black line around the UL */
}

ul li.current {
  /* all LIs with a class name of "current" inside a UL */
  background-color: red;
}
```

To put this another way, one side of our problem is already solved: in CSS, there exists a syntax for describing specific groups of nodes to retrieve. Prototype solves the other side of the problem: writing the code to parse these selectors in JavaScript and turn them into collections of nodes.

The $$ function can be used when simple ID or tag name querying is not powerful enough. Given any number of CSS selectors as arguments, $$ will search the document for all nodes that match those selectors.

```
$$('li'); // (all LI elements)
//-> [<li class="current" id="nav_home">, <li id="nav_archives">,
      <li id="nav_contact">, <li id="nav_google">]

$$('li.current'); // (all LI elements with a class name of "current")
//-> [<li class="current" id="nav_home">]

$$('#menu a'); // (all A elements within something with an ID of "menu")
//-> [<a href="/">, <a href="/archives">, <a href="/contact">,
      <a href="http://www.google.com" rel="external">]
```

There are two crucial advantages $$ has over ordinary DOM methods. The first is brevity: using $$ cuts down on keystrokes, even for the simplest of queries.

```
// BEFORE:
var items = document.getElementsByTagName('li');
// AFTER:
var items = $$('li');
```

As the complexity of your query increases, so does the savings in lines of code. $$ can be used to fetch node sets that would take many lines of code to fetch otherwise:

```
// find all LI children of a UL with a class of "current"
// BEFORE:
var nodes = document.getElementsByTagName('li');
var results = [];
for (var i = 0, node; node = nodes[i]; i++) {
  if (node.parentNode.tagName.toUpperCase() == 'UL' &&
      node.className.match(/(?:\s*|^)current(?:\s*|$)) {
    results.push(node);
  }
}
// AFTER:
var results = $$('ul > li.current');
```

The second advantage is something we've talked about already: the nodes returned by `$$` are already "extended" with Prototype's node instance methods.

If you're a web designer, you're likely familiar with CSS, but the power of `$$` goes far beyond the sorts of selectors you're likely accustomed to. `$$` supports virtually all of CSS3 syntax, including some types of selectors that you may not have encountered:

- Querying by attribute:
  - `$('input[type="text"]')` will select all text boxes.
  - `$$('a[rel]')` will select all anchor tags with a `rel` attribute.
  - `$$('a[rel~=external])` will select all `a` elements with the word "external" in the `rel` attribute.
- Querying by adjacency:
  - `$$('ul#menu > li') li') selector>` will select all `li` elements that are direct children of `ul#menu`.
  - `$$('li.current + li')` will select any `li` sibling that directly follows a `li.current` in the markup.
  - `$$('li.current ~ li')` will select all the following siblings of a `li.current` element that are `li` elements themselves.
- Negation:
  - `$$('ul#menu li:not(.current)')` will select all `li` elements that *don't* have a class name of `current`.
  - `$$('ul#menu a:not([rel])')` will select all `a` elements that *don't* have a `rel` attribute.

These are just some of the complex selectors you can use in `$$`. For more information on what's possible, consult the Prototype API reference online (`http://prototypejs.org/api/`). We'll encounter other complex selectors in some of the code we'll write later in this book.

Summary

You can do far more with Prototype than what I've just described, but the functions in this chapter are the ones you'll use most often. And although they solve common problems, they also form the foundation for a general scripting philosophy: one that espouses fewer lines of code, separation of content and behavior, and the principle of least surprise. Later on, you'll learn how to use these functions within a set of conventions to make your DOM scripting experience far more pleasant.

CHAPTER 3

Collections (Or, Never Write a for Loop Again)

Collections are at the heart of DOM scripting—arrays, hashes, DOM `NodeLists`, and various other groups of items. Nearly all your scripts will do some form of iteration over an array. So why is iteration so bland in JavaScript?

Prototype sports a robust library for dealing with collections. It makes arrays astoundingly flexible (and invents `Hash`, a subclass of `Object`, for key/value pairs), but can also be integrated into any collections you use in your own scripts.

The Traditional for Loop

Amazingly, the first version of JavaScript didn't even support arrays. They were added soon after, but with only one real enhancement over a vanilla `Object`—a magic `length` property that would count the number of numeric keys in the array. For example

```
var threeStooges = new Array();
threeStooges[0]  = "Larry";
threeStooges[1]  = "Curly";
threeStooges[2]  = "Moe";
console.log(threeStooges.length);
//-> 3
```

The `length` property and the ubiquitous `for` looping construct result in a simple, low-tech way to loop over an array's values: start at `0` and count up to the value of `length`.

```
for (var i = 0; i < threeStooges.length; i++) {
  console.log(threeStooges[i] + ": Nyuk!");
}
```

This is a fine and decent way to loop, but JavaScript is capable of so much more! A language with JavaScript's expressive power can embrace *functional programming* concepts to make iteration smarter.

Functional Programming

JavaScript is a multi-paradigm language. It can resemble the imperative style of C, the object-oriented style of Java, or the functional style of Lisp. To illustrate this, let's define a function and see what it can do:

```
function makeTextRed(element) {
  element.style.color = "red";
}
```

This function expects a DOM element node and does exactly what it says: it turns the node's enclosed text red. To apply this function to an entire collection of nodes, we can use the venerable `for` loop:

```
var paragraphs = $$('p');
for (var i = 0; i < elements.length; i++)
  makeTextRed(elements[i]);
```

But let's look at this from another angle. You learned in Chapter 1 that functions in JavaScript are "first-class objects," meaning that they can be treated like any other data type, like so:

```
typeof makeTextRed //-> "function"
makeTextRed.constructor; //-> Function
var alias = makeTextRed;
alias == makeTextRed; //-> true
```

In short, anything that can be done with strings, numbers, or other JavaScript data types can be done with functions.

This enables a different approach to iteration: since functions can be passed as arguments to other functions, you can define a function for iterating over an array. Again, this is easier to explain with code than with words:

```
function each(collection, iterator) {
  for (var i = 0; i < collection.length; i++)
    iterator(collection[i]);
}
```

The `iterator` argument is a function. The `each` method we just wrote will loop over an array's indices and call `iterator` on each, passing into it the current item in the array.

Now we can iterate thusly:

```
var paragraphs = $$('p');
each(paragraphs, makeTextRed);
```

Also remember from Chapter 1 that functions have a literal notation—you don't have to name a function before you use it. If we won't use the `makeTextRed` function anywhere else in the code, then there's no reason to define it beforehand.

```
each(paragraphs, function(element) {
  element.style.color = "red";
});
```

We can make one more improvement to our code. Since each is a method made to act on arrays, let's make it an instance method of all arrays:

```
Array.prototype.each = function(iterator) {
  for (var i = 0; i < this.length; i++)
    iterator(this[i]);
};
```

Remember that `this` refers to the execution scope of the function—in this case, it's the array itself. Now we can write the following:

```
paragraphs.each(function(element) {
  element.style.color = "red";
});
```

To look at this more broadly, we've just abstracted away the implementation details of iteration. Under the hood, we're calling the same old `for` loop, but because we've built a layer on top, we're able to define other functions that *involve* iterating but do much more than the preceding `each` example.

ABOUT FUNCTION NOTATION

This book uses a common notation to distinguish between *static* methods and *instance* methods. Static methods are marked with a dot—for example, `Array.from` refers to the `from` method on the `Array` object. Instance methods are marked with an octothorpe: `Array#each` refers to the `each` method on `Array.prototype`—that is, a method on an *instance* of `Array`.

Prototype's Enumerable Object

Prototype defines a handful of functions in an object called `Enumerable`. Anything that is "enumerable" (anything that can be iterated over) can use these methods.

These functions include `each` (much like the one we defined previously) and many other methods that all hook into `each` internally. These methods aren't specific to arrays—they can be used on *any* collection, as long as we tell them how to enumerate the items therein.

Prototype automatically extends `Enumerable` onto `Array`. At the end of the chapter, you'll learn how to implement `Enumerable` in your own classes, but for now we'll use arrays for all our examples.

Using Enumerable#each

`Enumerable#each` is the foundation that the rest of `Enumerable` relies upon, so let's take a closer look at it.

I've been defaming `for` loops for several pages now, but they do have one critical advantage over functional iteration: they let you short-circuit the iteration flow by using the keywords `break` (abort the loop) and `continue` (skip to the next item in the loop). We need a way to emulate these keywords if we want to match the feature set of traditional loops.

```
var elements = $$('.menu-item');
// find the element whose text content contains "weblog"
for (var i = 0, element; element = elements[i]; i++) {
  if (!element.id) continue;
  if (element.innerHTML.include('weblog')) break;
}
```

Simulating `continue` is easy enough—an empty return within a function will do the trick:

```
var elements = $$('.menu-item'), weblogElement;
elements.each( function(element) {
  if (!element.id) return;
  /* ... */
});
```

But a `break` equivalent takes a bit of voodoo. Prototype makes smart use of exceptions to pull this off. It creates a `$break` object that can be thrown within loops to exit immediately.

```
var elements = $$('.menu-item'), weblogElement;
elements.each( function(element) {
  if (!element.id) return;
  if (element.innerHTML.include('weblog')) {
    weblogElement = element;
    throw $break;
  }
});
```

In this example, we "throw" the `$break` object as though it were an exception. It interrupts the execution of the function and gets "caught" higher in the call stack, at which point the `each` method stops iterating and moves on.

Now we're even. It's rare that you'll need to use `$break`—most of the use cases for breaking out of loops are addressed by other `Enumerable` methods—but it's comforting to know it's there.

Finding Needles in Haystacks: detect, select, reject, and partition

The code pattern we used in the last section—finding one needle in a haystack—is a common one, but we can express it more concisely than through an `each` loop. The function we pass into `each` serves as an item manipulator, but we can also use that function as a litmus test to let us know whether an item matches our needle. The next four methods do just this.

Using Enumerable#detect

`Enumerable#detect` finds and returns one item in your collection. It takes a function as an argument (one that returns `true` or `false`) and will return the first item in the collection that causes the function to return `true`.

```
function isEven(number) {
  return number % 2 == 0;
}

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10].detect(isEven);
```

//-> 2

If there are no matches, `detect` will return `false`.

Using Enumerable#select

What if we need to find *several* needles in a haystack?

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10].select(isEven);
//-> [2, 4, 6, 8, 10]
```

Just like `detect`, `select` tests each item against the given function. But it doesn't stop after the first match—it will return *all* items in the collection that match the criteria.

```
["foo", 1, "bar", "baz", 2, null].select( function(item) {
  return typeof item === "string";
});
//-> ["foo", "bar", "baz"]
```

Unlike `detect`, which is guaranteed to return only one item, `select` will always return an array. If there are no matches, it will return an empty array.

Using Enumerable#reject

Nearly identical to `select`, `reject` will return all the items that *fail* a particular test.

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10].reject(isEven);
//-> [1, 3, 5, 7, 9]
```

Using Enumerable#partition

When you need to separate a collection into two groups, use `Enumerable#partition`. It returns a two-item array: the first an array of all items that passed the test, and the second an array of all items that failed the test.

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10].partition(isEven);
//-> [ [2, 4, 6, 8, 10], [1, 3, 5, 7, 9] ]
```

Sorting Collections: min, max, and sortBy

The next three `Enumerable` methods—`min`, `max`, and `sortBy`—address common situations of arranging collections by value.

Using Enumerable#min and #max

Much like `Math.min` and `Math.max`, which identify the smallest and largest values of all the arguments passed to them, `Enumerable#min` and `#max` will find the smallest and largest values in an existing group:

```
Math.min(1, 4, 9, 16, 25); //-> 1
Math.max(1, 4, 9, 16, 25); //-> 25

var squares = [1, 4, 9, 16, 25];
squares.min(); //-> 1
squares.max(); //-> 25
```

In this example, it's easy to figure out what the minimum and maximum values are—numbers are directly comparable. For trickier collections, though, you'll need to pass in a function to identify exactly *what* you want the maximum or minimum of:

```
var words = ["flowers", "the", "hate", "moribund", "sesquicentennial"];
words.max( function(word) { return word.length; } ); //-> 16
words.min( function(word) { return word.length; } ); //-> 3
```

Comparing on string length, we get `3` and `16` as the min and max, respectively—the lengths of the shortest and longest words in the array.

Using Enumerable#sortBy

JavaScript has a built-in sorting method: `Array#sort`. Why do we need another?

Let's illustrate. If we try to use `Array#sort` on an example set of numbers, we'll be in for a surprise:

```
[2, 5, 4, 8, 9, 1, 3, 10, 7, 6].sort();
//-> [1, 10, 2, 3, 4, 5, 6, 7, 8, 9]
```

As it happens, `sort` called with no arguments will coerce the array items into strings before it compares them. (`10` is greater than `2`, but `"10"` is less than `"2"`.) If we want to compare the numbers directly, we must pass a function argument into `sort`:

```
[2, 5, 4, 8, 9, 1, 3, 10, 7, 6].sort(function(a, b) {
  return a - b;
});
//-> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

The passed function tells `sort` how to compare any two items in the array. In the preceding example, if `a` is greater than `b`, then the return value will be positive, indicating that `a` should *follow* `b`. If `b` is greater than `a`, then the return value will be negative, and `a` will *precede* `b`. (If the return value is `0`, then the two are equal, of course.)

This is nuts—or, at the very least, surprising. We need a better sort function.

`Enumerable#sortBy` works a little differently. It, too, takes a function argument, but the function is used only to translate a given item to a comparison value:

```
var words = ["aqueous", "strength", "hated", "sesquicentennial", "area"];

// sort by word length
words.sortBy( function(word) { return word.length; } );
//-> ["area", "hated", "aqueous", "strength", "sesquicentennial"]

// sort by number of vowels in the word
words.sortBy( function(word) { return word.match(/[aeiou]/g).length; } )
//-> ["strength", "hated", "area", "aqueous", "sesquicentennial"]
```

As you can see, the comparison function takes one argument, rather than two. Most developers will find this far more intuitive.

Advanced Enumeration: map, inject, invoke, and pluck

The next four `Enumerable` methods carry more cryptic names, but are every bit as useful as the methods described previously.

Using Enumerable#map and Enumerable#inject

The `map` method performs parallel transformation. It applies a function to every item in a collection, pushes each result into an array, and then returns that array.

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10].map( function(num) { return num * num; } );
//-> [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

The `inject` method returns an accumulated collection. Think of it as a hat being passed around to each of the items—each item throws a different quantity into the hat.

```
var words = ["aqueous", "strength", "hated", "sesquicentennial", "area"];
var totalLength = words.inject(0, function(memo, string) {
  return memo + string.length;
});
// -> 40
```

Since we need to keep track of `memo`—the variable that stores our running total—the arguments for `inject` are slightly different. The first argument of `inject` is our starting value—`0`, in this case, since we're dealing with a numeric property. The second argument is the function we're using against the collection's items.

This inner function itself takes two arguments: our running total (`memo`) and the item we're working with (`string`). The function's return value will be used as the memo for the next item in the collection.

This can be a bit confusing, so let's add a logging statement to the inner function to illustrate what's going on:

```
var totalLength = words.inject(0, function(memo, string) {
  console.log('received ' + memo + '; added ' + string.length);
  console.log('returning ' + (memo + string.length));
  return memo + string.length;
});

//-> received 0; added 7
//-> returning 7
//-> received 7; added 8
//-> returning 15
//-> received 15; added 5
//-> returning 20
//-> received 20; added 16
//-> returning 36
//-> received 36; added 4
//-> returning 40

//-> 40
```

Now that you can see each step in the enumeration, the behavior of `inject` should be easier to follow.

Using Enumerable#pluck and Enumerable#invoke

These two `Enumerable` methods are somewhat special. They take a string as their first argument instead of a function.

The `pluck` method collects individual properties on each of the objects on the collection:

```
var words = ["aqueous", "strength", "hated", "sesquicentennial", "area"];
words.pluck('length');
//-> [7, 8, 5, 16, 4]
```

Note that this example code is equivalent to

```
words.map( function(word) { return word.length; } );
```

but is shorter and more meaningful.

The `invoke` method is similar: it calls the specified instance method on each item. Let's illustrate by using one of Prototype's string methods:

```
"   aqueous ".strip(); //-> "aqueous"
var paddedWords = ["   aqueous ", "strength ", "     hated   ",
 "sesquicencennial", " area    "];
words.invoke('strip');
//-> ["aqueous", "strength", "hated", "sesquicentennial", "area"]
```

This code is equivalent to

```
words.map( function(word) { return word.strip(); } );
```

but `invoke` can also pass arguments along to the instance method. Simply add the required number of arguments *after* the first:

```
"swim/swam".split('/'); //-> ["swim", "swam"]
"swim/swam".replace('/', '|'); //-> "swim|swam"

var wordPairs = ["swim/swam", "win/lose", "glossy/matte"];
wordPairs.invoke('split', '/');
//-> [ ["swim", "swam"], ["win", "lose"], ["glossy", "matte"] ]
wordPairs.invoke('replace', '/', '|');
//-> ["swim|swam", "win|lose", "glossy|matte"]
```

The `map`, `inject`, `pluck`, and `invoke` methods greatly simplify four very common code patterns. Become familiar with them and you'll start to notice uses for them all over the code you write.

Other Collections That Use Enumerable

Two other Prototype classes that make use of `Enumerable` are `Hash` and `ObjectRange`. Together they serve as great examples of how to use `Enumerable` with other types of collections.

Hash

There is no built-in facility in JavaScript for setting key/value pairs—the construct that's known as a *hash* (in Ruby), a *dictionary* (in Python), or an *associative array* (in PHP). There is, of course, an ordinary object, and this suffices for most cases.

```
var airportCodes = {
  AUS: "Austin-Bergstrom Int'l",
  HOU: "Houston/Hobby",
  IAH: "Houston/Intercontinental",
  DAL: "Dallas/Love Field",
  DFW: "Dallas/Fort Worth"
};

for (var key in airportCodes) {
  console.log(key + " is the airport code for " + airportCodes[key] + '.');
}

>>> AUS is the airport code for Austin-Bergstrom Int'l.
>>> HOU is the airport code for Houston/Hobby.
>>> IAH is the airport code for Houston/Intercontinental.
>>> DAL is the airport code for Dallas/Love Field.
>>> DFW is the airport code for Dallas/Fort Worth.
```

We can declare an object and iterate over its properties quite easily. This doesn't get us everything a hash would, but it comes very close.

Eventually, however, we'll run into two major problems.

Objects Have No Key Safety

An object is not a blank slate when it's declared. It has native properties and methods with names that may conflict with the names you'd want to use for your keys.

```
var obj = {};
obj.constructor; //-> Object

obj = {
  name: "Statue of Liberty",
  constructor: "Frédéric Bartholdi"
};

obj.constructor; //-> "Frédéric Bartholdi"
```

In this example, a built-in property (`constructor`) that has special meaning in JavaScript is being shadowed by a property of the same name that we assign on the object instance. Similar collisions can occur with `toString`, `valueOf`, and other built-in properties. The safety of any arbitrary key cannot be guaranteed.

These might seem like edge cases, but key safety is especially important when you're building a hash in which the key names depend on user input, or on some other means that isn't planned beforehand by the developer.

The Object.prototype Problem

As we briefly covered in Chapter 2, JavaScript suffers from a flaw caused by two of its features stepping on one another. In theory, we can define properties on `Object.prototype` and have them propagate to every instance of `Object`. Unfortunately, when properties are enumerated in a `for...in` loop, anything that's been defined on `Object.prototype` will get picked up.

```
Object.prototype.each = function(iterator) {
  for (var i in this)
    iterator(i, this[i]);
};
var obj = {
  name: "Statue of Liberty",
  constructor: "Frédéric Bartholdi"
};

obj.each(console.log); // (pass the key and value as arguments to console.log)
>>> "name" "Statue of Liberty"
>>> "constructor" "Frédéric Bartholdi"
>>> "each" "function(iterator) { for (var i in this) iterator(i, this[i]); }"
```

Regrettably, there's no way to suppress this behavior. We could get around it by avoiding ordinary `for...in` loops altogether, wrapping code around them that ensures we enumerate only properties that exist on the *instance*, but then we've only solved the problem for our *own* scripts. Web pages often pull in scripts from various sources, some of which may be unaware of each other's existence. We can't expect all these scripts to boil the ocean just to make our lives a little easier.

The Solution

There's a way around all this, although it may not be ideal: creating a new "class" for creating true hashes. That way we can define instance methods on its prototype without encroaching on `Object.prototype`. It also means we can define methods for getting and setting keys—internally they can be stored in a way that won't collide with built-in properties.

Prototype's `Hash` object is meant for key/value pairs. It is designed to give the syntactic convenience of `Enumerable` methods without encroaching on `Object.prototype`.

To create a hash, use `new Hash` or the shorthand `$H`:

```
var airportCodes = new Hash();
```

To set and retrieve keys from the hash, use `Hash#set` and `Hash#get`, respectively:

```
airportCodes.set('AUS', 'Austin-Bergstrom Int'l');
airportCodes.set('HOU', 'Houston/Hobby');

airportCodes.get('AUS'); //-> "Austin-Bergstrom Int'l"
```

Ick! Do we really have to set keys individually like that? Is this worth the trade-off?

Luckily, we don't have to do it this way. We can pass an object into the `Hash` constructor to get a hash with the key/value pairs of the object. Combine this with the `$H` shortcut, and we've got a syntax that's almost as terse as the one we started with:

```
var airportCodes = $H({
  AUS: "Austin-Bergstrom Int'l",
  HOU: "Houston/Hobby",
  IAH: "Houston/Intercontinental",
  DAL: "Dallas/Love Field",
  DFW: "Dallas/Fort Worth"
});
```

You can also add properties to a hash en masse at any time using `Hash#update`:

```
var airportCodes = new Hash();
airportCodes.update({
  AUS: "Austin-Bergstrom Int'l",
  HOU: "Houston/Hobby",
  IAH: "Houston/Intercontinental",
  DAL: "Dallas/Love Field",
  DFW: "Dallas/Fort Worth"
});
```

This code gives the same result as the preceding example.

You can get a hash's keys or values returned as an array with `Hash#keys` and `Hash#values`, respectively:

```
airportCodes.keys(); //-> ["AUS", "HOU", "IAH", "DAL", "DFW"]
airportCodes.values();
//-> ["Austin-Bergstrom Int'l", "Houston/Hobby", "Houston/Intercontinental",
//->  "Dallas/Love Field", "Dallas/Fort Worth"]
```

Finally, you can convert a hash back to a plain `Object` with `Hash#toObject`:

```
airportCodes.toObject();
//-> {
//->   AUS: "Austin-Bergstrom Int'l",
//->   HOU: "Houston/Hobby",
//->   IAH: "Houston/Intercontinental",
//->   DAL: "Dallas/Love Field",
//->   DFW: "Dallas/Fort Worth"
//-> }
```

Enumerable Methods on Hashes

Enumerable methods on hashes work almost identically to their array counterparts. But since there are two parts of each item—the key and the value—the object passed into the iterator function works a bit differently:

```
airportCodes.each( function(pair) {
  console.log(pair.key + ' is the airport code for ' + pair.value + '.');
});
```

```
>>> AUS is the airport code for Austin-Bergstrom Int'l.
>>> HOU is the airport code for Houston/Hobby.
>>> IAH is the airport code for Houston/Intercontinental.
>>> DAL is the airport code for Dallas/Love Field.
>>> DFW is the airport code for Dallas/Fort Worth.
```

The `pair` object in the preceding example contains two special properties, `key` and `value`, which contain the respective parts of each hash item. If you prefer, you can also refer to the key as `pair[0]` and the value as `pair[1]`.

Keep in mind that certain `Enumerable` methods are designed to return arrays, regardless of the original type of the collection. `Enumerable#map` is one of these methods:

```
airportCodes.map( function(pair) {
  return pair.key.toLowerCase();
});
//-> ["aus", "hou", "iah", "dal", "dfw"]
```

In general, these methods work the way you'd expect them to. Be sure to consult the Prototype API documentation if you get confused.

ObjectRange

Prototype's `ObjectRange` class is an abstract interface for declaring a starting value, an ending value, and all points in between. In practice, however, you'll be using it almost exclusively with numbers.

To create a range, use `new ObjectRange` or the shorthand `$R`.

```
var passingGrade = $R(70, 100);
var teenageYears = $R(13, 19);
var originalColonies = $R(1, 13);
```

A range can take three arguments. The first two are the start point and endpoint of the range. The third argument, which is optional, is a Boolean that tells the range whether to exclude the end value.

```
var inclusiveRange = $R(1, 10);       // will stop at 10
var exclusiveRange = $R(1, 10, true); // will stop at 9
```

Ranges are most useful when you need to determine whether a given value falls within some arbitrary boundaries. The `include` instance method will tell you whether your value is included in the range.

```
if (passingGrade.include(studentGrade))
  advanceStudentToNextGrade();
```

But ranges can also use any method in `Enumerable`. We can take advantage of this to simplify our example code from earlier in the chapter.

```
function isEven(num) {
  return num % 2 == 0;
}
var oneToTen = $R(1, 10);
oneToTen.select(isEven); //-> [2, 4, 6, 8, 10]
oneToTen.reject(isEven); //-> [1, 3, 5, 7, 9]
var firstTenSquares = oneToTen.map ( function(num) { return num * num; } );
//-> [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Turning Collections into Arrays

`Enumerable` also boasts a generic `toArray` method that will turn any collection into an array. Obviously, this isn't very useful for arrays themselves, but it's a convenience when working with hashes or ranges.

```
$H({ foo: 'bar', baz: 'thud' }).toArray();
//-> [ ['foo', 'bar'], ['baz', 'thud'] ]
$R(1, 10, true).toArray();
//-> [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Keep in mind that using `$A`, the array-coercion shortcut function, will produce the same result. If an object has a `toArray` method, `$A` will use it.

Using Enumerable in Your Own Collections

The fun is not confined to arrays, hashes, and ranges. `Enumerable` can be "mixed into" any class—all it needs to know is how to enumerate your collections.

Let's take a look at the source code for `Enumerable#each`:

```
// Excerpt from the Prototype source code
each: function(iterator) {
  var index = 0;
  try {
    this._each(function(value) {
      iterator(value, index++);
    });
  } catch (e) {
    if (e != $break) throw e;
  }
  return this;
}
```

The nonhighlighted portion manages the boring stuff: keeping track of the index value and catching our handy `$break` exception. The actual enumeration is delegated to a method called `_each`. This is where the magic happens.

`Enumerable` needs the `_each` method to tell it how to enumerate. For example, `Array.prototype._each` looks like this:

```
// Excerpt from the Prototype source code

each: function(iterator) {
  for (var i = 0, length = this.length; i < length; i++)
    iterator(this[i]);
}
```

So, we haven't gotten rid of the `for` loop entirely—we've just stashed it away in a function you'll never call directly.

Here's a horribly contrived example to illustrate all this. Let's say we want a specific kind of array that will enumerate only its even indices, skipping over the odd ones. Writing the constructor for this class is easy enough:

```
var EvenArray = function(array) {
  this.array = array;
};
```

We can feed it any ordinary array by calling this constructor with an existing array:

```
var even = new EvenArray(["zero", "one", "two", "three", "four", "five"]);
```

Now let's define an `_each` method on our class's prototype:

```
EvenArray.prototype._each = function(iterator) {
  for (var i = 0; i < this.array.length; i += 2)
    iterator(this.array[i]);
};
```

Notice that we're incrementing two at a time, skipping all the odd values of `i`.

Now we can extend `Enumerable` onto our class:

```
Object.extend(EvenArray.prototype, Enumerable);
```

Remember how `Object.extend` works—we're taking each property on `Enumerable` and copying it onto `EvenArray.prototype`.

And we're done. No, really. Try it out:

```
even.each( function(item) { console.log(item); });
//-> zero
//-> two
//-> four
even.map( function(item) { return item.toUpperCase(); });
//-> ["ZERO", "TWO", "FOUR"]
```

By defining one method (`_each`), we're now able to use any `Enumerable` method on instances of `EvenArray`.

So, to review, mixing `Enumerable` into your own classes is a two-step process. First, define the `_each` method. Next, copy the `Enumerable` methods onto your class with `Object.extend`. When using Prototype, you'll find there is rarely a step three.

Summary

Playing with `Enumerable` methods is a good way to dip your toe into the Prototype pool. You'll use them in many different contexts throughout your code. And because they're abstract, they're mercifully free of the annoying cross-browser issues that cast their shadow over nearly every other area of browser-based JavaScript development. Before you forge ahead, make sure you're very comfortable with everything we've covered in this chapter.

CHAPTER 4

Ajax: Advanced Client/Server Communication

By now, you're almost certainly familiar with *Ajax* as a buzzword. Technically, it's an acronym—Asynchronous JavaScript and XML—and refers specifically to JavaScript's `XmlHttpRequest` object, which lets a browser initiate an HTTP request outside the confines of the traditional page request.

Yawn. The technology isn't the exciting part. Ajax is huge because it pushes the boundaries of what you can do with a web UI: *it lets you reload part of a page without reloading the entire page*. For a page-based medium like the Web, this is a seismic leap forward.

Ajax Rocks

`XmlHttpRequest` (XHR for short) is a JavaScript interface for making arbitrary HTTP requests. It lets a developer ask the browser to fetch a URL in the background, but without any of the typical baggage of a page request—the hourglass, the new page, and the re-rendering.

Think of it as an HTTP library for JavaScript, not unlike Ruby's `Net::HTTP` class or PHP's `libcurl` bindings. But because it lives on the client side, it can act as a scout, marshaling requests between client and server in a much less disruptive way than the typical page request.

The difference is crucial—the user has to wait around for a page request, but XHR doesn't. Like the acronym says, Ajax allows for *asynchronous* communication—the JavaScript engine can create a request, send it off, and then do other things until the response comes back. It's far better than making your *users* do other things until the response comes back.

Ajax Sucks

It's not all sunshine and rainbows, though. Ajax is much easier to talk about than it is to *do*.

The problem that afflicts JavaScript in general applies to Ajax in particular: the `XmlHttpRequest` object has its own set of bugs, inconsistencies, and other pitfalls from browser to browser. Created by Microsoft and first released as part of Internet Explorer 5, the XHR object gained popularity once it was implemented by the other major browser vendors—even though there was no formal specification to describe how it ought to work. (The W3C has since started an XHR specification, currently in "Working Draft" status.)

For this reason, it's painful and frustrating to work with XHR without some sort of wrapper library to smooth out the rough edges. Prototype takes the awkward, unintuitive API of `XmlHttpRequest` and builds an easy-to-use API around it.

Prototype's Ajax Object

Let's set up an environment to play around with Ajax. In a text editor, create a file named `ajax.js` and place some JavaScript content inside. This will be the file we load with Ajax (see Listing 4-1).

Listing 4-1. *The ajax.js File*

```
alert('pancakes!');
```

Create a directory for this file and save it.

■ **Caution** Since Ajax is an HTTP request interface, these examples require a web server to communicate with. Opening these examples straight from the local disk (using the `file:` protocol) will yield mixed results. Try running them on a local installation of Apache—or on space you control on a remote web server.

Now we need a page to make the Ajax request *from*. Create an empty HTML page, call it index.html, and save it in the same directory as ajax.js. Listing 4-2 shows the index.html file.

Listing 4-2. *The index.html File*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <title>Blank Page</title>

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

  <body>
    <h1>Blank Page</h1>
  </body>
</html>
```

Notice how we load Prototype by including it in the head of our document via a script tag. You'll need to place a copy of prototype.js in the same directory as index.html.

Now open index.html in Firefox. We'll use Firefox for these examples so that we can execute commands on the fly in the Firebug interactive shell. Make sure the Console tab is focused, as shown in Figure 4-1.

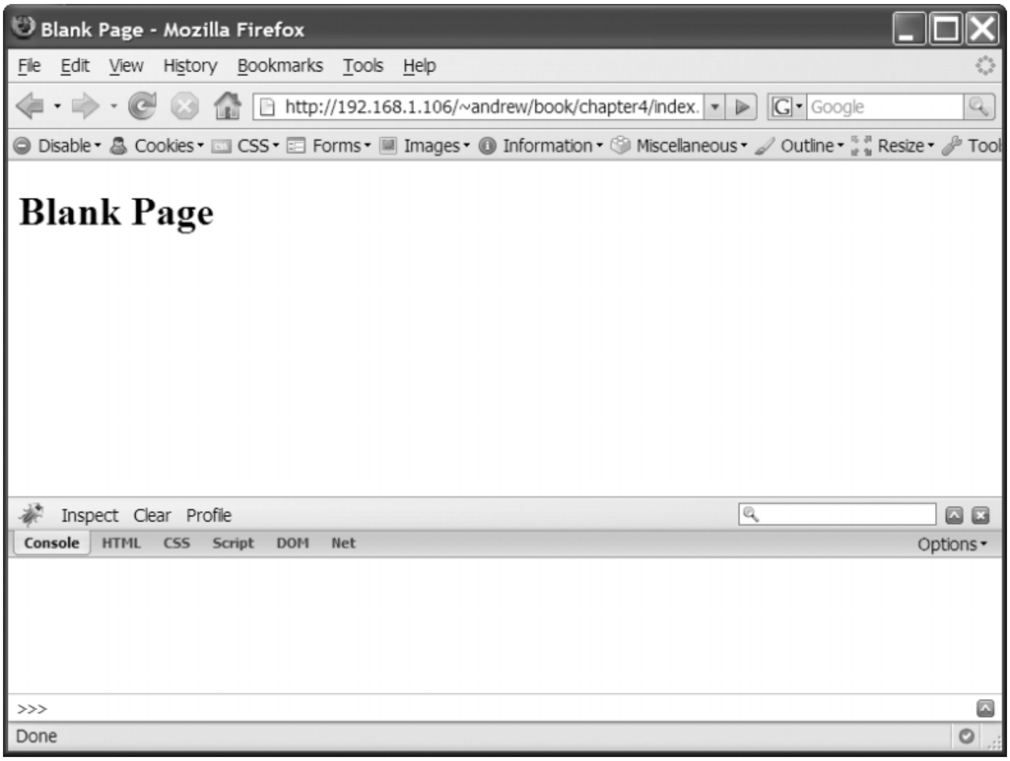

Blank Page - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/book/chapter4/index.
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tool
Blank Page
Inspect Clear Profile
Console HTML CSS Script DOM Net
Options
>>>
Done

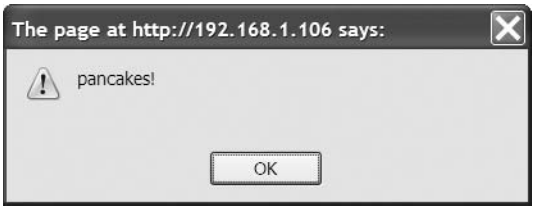
Figure 4-1. *Firebug is open to the console tab at the bottom of the screen.*

Ajax.Request

Now type the following into the shell:

```
new Ajax.Request('ajax.js', { method: 'get' });
```

You should see the dialog shown in Figure 4-2.



The page at http://192.168.1.106 says:

pancakes!

Figure 4-2. *This dialog came from our external JavaScript file.*

The `Ajax.Request` call fetched our external file and evaluated the JavaScript we placed inside. This simple example teaches you several things about `Ajax.Request`:

- It's called as a constructor (using the `new` keyword).
- Its first argument is the name of the URL you want to load. Here it's a relative URL because the file we want to load is in the same directory; but you can also use an absolute URL (one that begins with a forward slash).

■ **Caution** Keep in mind that this URL can't begin with `http` because of the same-domain policy of Ajax—even if the URL points internally.

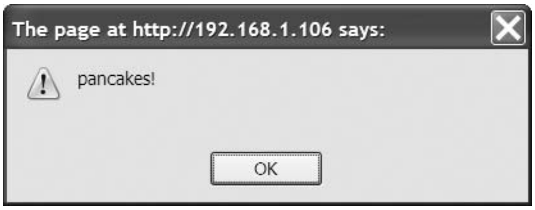
- Its second argument is an object that can contain any number of property/value pairs. (We'll call this the `options` argument.) Prototype uses this convention in a number of places as a way of approximating named arguments. In this example, we're specifying that the browser should make an HTTP GET request for this file. We only need to specify this because it's overriding a default—if you omit the `method` option, `Ajax.Request` defaults to a POST.
- The JavaScript we placed in `ajax.js` was evaluated *automatically*, so we know that `Ajax.Request` will evaluate the response if it's served up as JavaScript. Web servers typically give JS files a MIME type of `text/javascript` or `application/x-javascript`; Prototype knows to treat those types (and a handful of others) as JavaScript.

Now let's add to this example. Type the same line as before, but with an extra property in the `options` argument:

```
new Ajax.Request('ajax.js', { method: 'get',
  onComplete: function() { alert('complete'); }
});
```

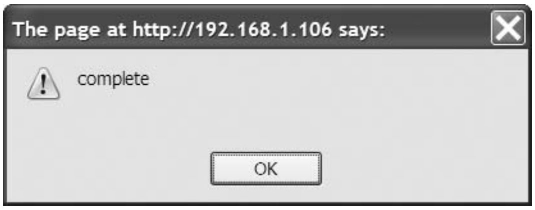
■ **Tip** You can switch the Firebug console to multiline input by clicking the button at the far right of the command line.

Figure 4-3 shows the results. This time, you'll see two dialogs: the original, "pancakes!" and the one inside the line highlighted in the previous code block, "complete."



The page at http://192.168.1.106 says:

pancakes!



The page at http://192.168.1.106 says:

complete

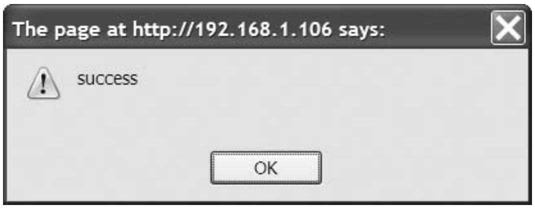
Figure 4-3. *These two dialogs appear in sequence.*

So, by adding just a little code to this example, you've learned two more things:

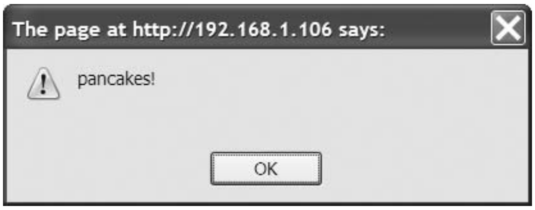
- The `onComplete` option is a new property in our `options` object. It sets up a *callback*—a function that will run at a certain point in the future. An Ajax request keeps the browser updated on its progress, triggering several different "ready states" along the way. In this case, our `onComplete` function will be called when the request is complete.
- The "pancakes!" dialog appears before the "complete" dialog, so you can deduce that the `onComplete` function is called *after* the response is evaluated.

Let's use another callback. Replace `onComplete` with `onSuccess` (see Figure 4-4):

```
new Ajax.Request('ajax.js', { method: 'get',
  onSuccess: function() { alert('success'); }
});
```


The page at http://192.168.1.106 says:

success


The page at http://192.168.1.106 says:

pancakes!

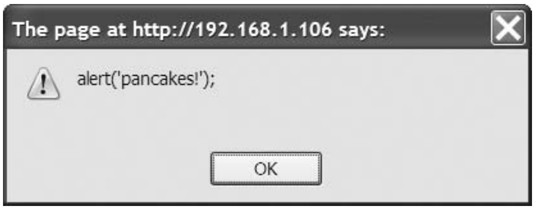
Figure 4-4. *These two dialogs appear in sequence.*

Figure 4-4 is subtly different than Figure 4-3. Like before, you'll see two dialog boxes—but this time the "pancakes!" dialog comes last. So, you can assume the following:

- The `onSuccess` option is a callback that will be run if the request is a success. If there's a failure of some kind (a 404 error, a communication error, an internal server error, etc.), its companion, `onFailure`, will get called instead.
- Since we saw the callback's `alert` dialog first, we know that `onSuccess` and `onFailure` are called before the remote JavaScript file is evaluated, and also before `onComplete`. True to its name, `onComplete` is called as the *very last thing* `Ajax.Request` does before it punches its timecard. But it decides between calling `onSuccess` or `onFailure` as soon as it knows the outcome of the request.

We can request any kind of file with Ajax—not just JavaScript files. To prove it, rename `ajax.js` to `ajax.txt` and try this (see Figure 4-5):

```
new Ajax.Request('ajax.txt', { method: 'get',
  onSuccess: function(request) { alert(request.responseText); }
});
```


The page at http://192.168.1.106 says:

alert('pancakes!');

Figure 4-5. *Our new dialog contains some familiar text.*

You just learned two more things from Figure 4-5:

- Because our “pancakes!” dialog didn’t appear, we know that the response was not evaluated as JavaScript—because it was served as an ordinary text file.
- Callbacks like `onSuccess` are passed the browser’s native `XmlHttpRequest` object as the first argument. This object contains several things of interest: the `readyState` of the request (represented as an integer between 0 and 4), the `responseText` (plain-text contents of the requested URL), and perhaps the `responseXML` as well (a DOM representation of the content, if it’s served as HTML or XML). That’s how we were able to display the contents of `ajax.txt` in our dialog.

Here’s where it all comes together—since we can fetch a fragment of HTML from a remote file, we can update the main page incrementally by dropping that fragment into a specific portion of the page. This is such a common task that Prototype has a subclass for it.

Ajax.Updater

Prototype’s `Ajax.Updater` does exactly what you think it does: it “updates” a portion of your page with external content from an Ajax request.

To demonstrate this, let’s add an empty container to our `index.html` file.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <title>Blank Page</title>

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

  <body>
    <h1>Blank Page</h1>
    <div id="bucket"></div>
  </body>
</html>
```

Now we can request an external HTML file and direct the browser to place its contents into the `div` we just created. So let's create a file called `ajax.html`, as shown in Listing 4-3.

Listing 4-3. *The ajax.html File*

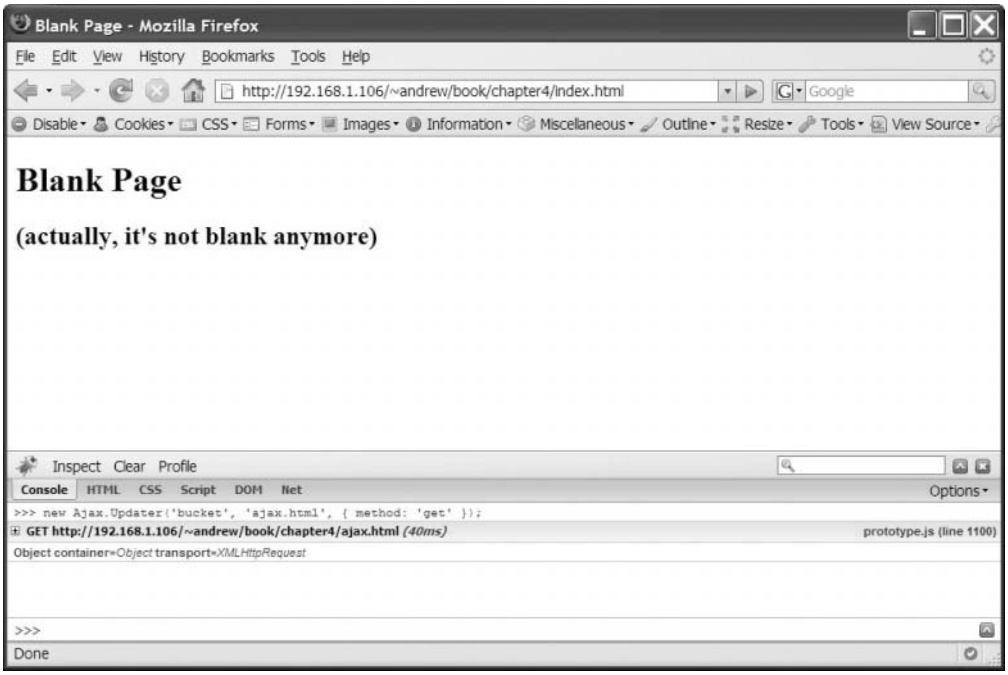
```
<h2>(actually, it's not blank anymore)</h2>
```

This isn't a full HTML file, you'll notice—since we'll be inserting this content into a fully formed page, it should just be an HTML fragment.

Now reload `index.html` in Firefox. You won't see the `div` we created, of course, because there's nothing in it yet. Type this into the Firebug console:

```
new Ajax.Updater('bucket', 'ajax.html', { method: 'get' });
```

In Figure 4-6, you'll see our once-empty `div` chock-full of content!


Blank Page - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/book/chapter4/index.html
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tools View Source
Inspect Clear Profile
Console HTML CSS Script DOM Net
Options
Done

Blank Page

(actually, it's not blank anymore)

```
>>> new Ajax.Updater('bucket', 'ajax.html', { method: 'get' });
GET http://192.168.1.106/~andrew/book/chapter4/ajax.html (40ms)        prototype.js (line 1100)
Object container=Object transport=XMLHttpRequest
>>>
```

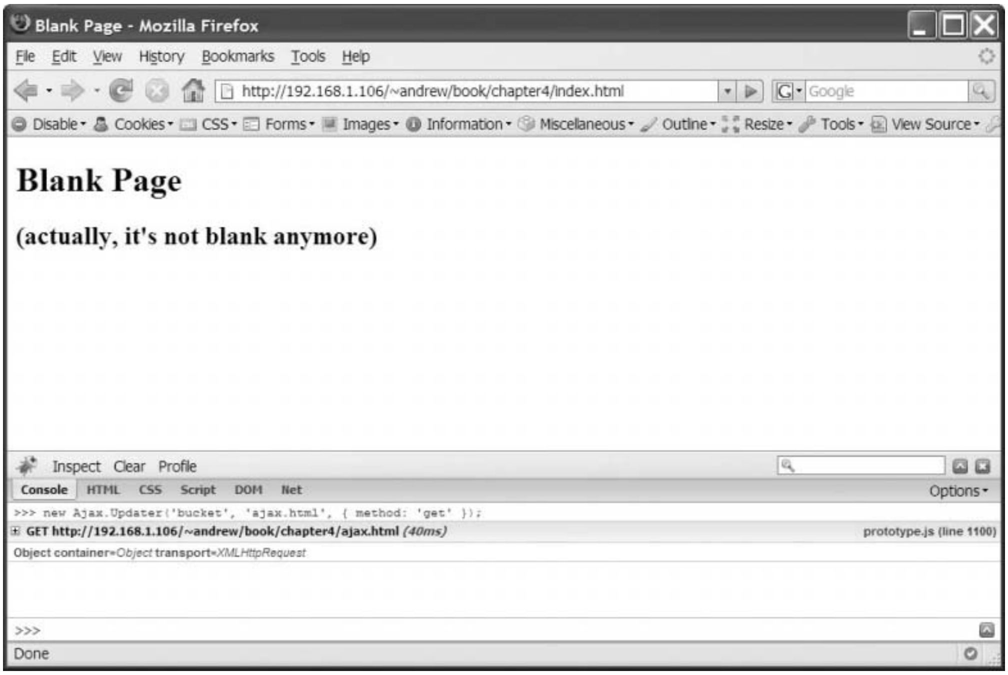
Figure 4-6. *Our h1 is no longer alone on the page.*

This line of code reads almost like a sentence: *Using Ajax, update the* `bucket` *element with the contents of* `ajax.html`. It introduces you to more new things:

- `Ajax.Updater` works a lot like `Ajax.Request`. But it's got an extra argument at the beginning: the element to be updated. Remember what you learned in Chapter 2: any function that takes a DOM node can also take a string reference to that node's ID. We could just as easily have used `$('bucket')` (or a native DOM call like `document.getElementsByTagName('div')[0]`) as the argument instead of `'bucket'`.

- Just like `Ajax.Request`, `Ajax.Updater` takes an `options` hash as its final argument. It supports all the options we've covered already, plus a few new ones, which we're about to look at.

Now press the up arrow key at the Firebug command line to bring up the statement you just typed. Run it again (see Figure 4-7).


Blank Page - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/book/chapter4/index.html
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tools View Source
Inspect Clear Profile
Console HTML CSS Script DOM Net
Options
Done

Blank Page

(actually, it's not blank anymore)

```
>>> new Ajax.Updater('bucket', 'ajax.html', { method: 'get' });
GET http://192.168.1.106/~andrew/book/chapter4/ajax.html (40ms)          prototype.js (line 1100)
Object container=Object transport=XMLHttpRequest
>>>
```

Figure 4-7. *Isn't this the same as the last?*

Nothing changed between Figures 4-6 and 4-7. Well, that's not true—something changed, but you didn't notice because the old content was identical to the new content. Every time you call `Ajax.Updater` on an element, it will *replace* the contents of that element.

You can change this behavior with one of `Ajax.Updater`'s options: `insertion`. If present, the updater object will add the response to the page without overwriting any existing content in the container.

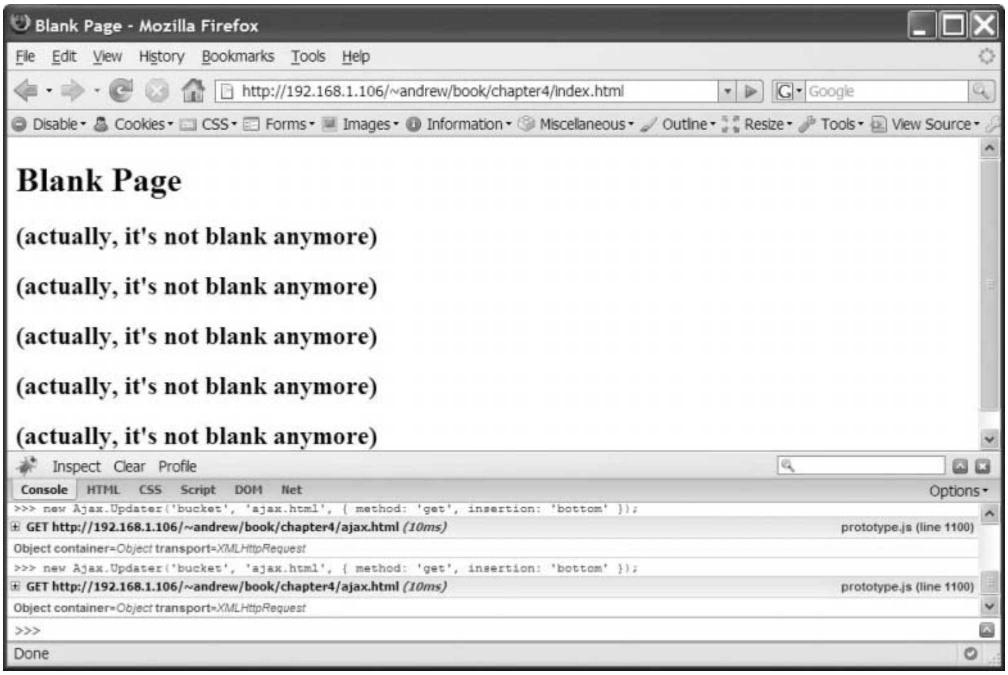
The `insertion` property takes one of four possible values: `top`, `bottom`, `before`, or `after`. Each one inserts the content in the described location, relative to the container element: `top` and `bottom` will insert *inside* the element, but `before` and `after` will insert *outside* the element.

So let's try appending the response instead. Type this into your console:

```
new Ajax.Updater('bucket', 'ajax.html', { method: 'get', insertion: 'bottom' });
```

Although it's a bit longer, this line of code also reads like a sentence: *Using Ajax, get the contents of* `ajax.html` *and insert them at the bottom of the* `bucket` *element.*

Run this code. Then run it again, and again, and again. Each time you'll see an extra `h2` tag on the page, as shown in Figure 4-8.


Blank Page - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/book/chapter4/index.html
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tools View Source
Inspect Clear Profile
Console HTML CSS Script DOM Net
Options
Done

Blank Page

(actually, it's not blank anymore)

(actually, it's not blank anymore)

(actually, it's not blank anymore)

(actually, it's not blank anymore)

(actually, it's not blank anymore)

```
>>> new Ajax.Updater('bucket', 'ajax.html', { method: 'get', insertion: 'bottom' });
⊞ GET http://192.168.1.106/~andrew/book/chapter4/ajax.html (10ms)      prototype.js (line 1100)
Object container=Object transport=XMLHttpRequest
>>> new Ajax.Updater('bucket', 'ajax.html', { method: 'get', insertion: 'bottom' });
⊞ GET http://192.168.1.106/~andrew/book/chapter4/ajax.html (10ms)      prototype.js (line 1100)
Object container=Object transport=XMLHttpRequest
>>>
```

Figure 4-8. *The h2s are starting to reproduce like mad.*

This is pretty cool stuff. It's a shame you have to run this code every single time, though. You could pay someone to sit at your desk and enter this line into the Firebug console over and over—but it's probably easier to use `Ajax.PeriodicalUpdater`.

Ajax.PeriodicalUpdater

Just as `Updater` builds on `Request`, `PeriodicalUpdater` builds on `Updater`. It works like it sounds: give it a URL, an element to update, and a time interval, and it will run an `Ajax.Updater` at that interval for the life of the page—or until you tell it to stop.

There are tons of applications for a repeating Ajax request: imagine the client side of a chat application asking the server if anyone's spoken in the last 10 seconds. Imagine a feed reader that checks every 20 minutes for new content. Imagine a server doing a resource-intensive task, and a client polling every 15 seconds to ask how close the task is to completion.

Rather than dispatch an Ajax request as the result of a user action—a click of a button or a drag-and-drop—these examples set up a request to run automatically, thereby saving the user the tedious task of manually reloading the page every time.

Let's try it. Reload `index.html` and run this command in the console:

```
new Ajax.PeriodicalUpdater('bucket', 'ajax.html', {
  method: 'get', insertion: 'bottom', frequency: 5
});
```

Right away we see our first `h2` element. Then, 5 seconds later, we see another. Then another. Now they're reproducing with no help from us.

So here's what you've probably figured out about `PeriodicalUpdater`:

- It takes the same basic arguments as `Ajax.Updater`, but it also accepts a `frequency` parameter in the `options` object, allowing you to set the number of seconds between requests. We could have omitted this parameter and left the frequency at the default 2 seconds—but then we wouldn't have had the occasion to talk about it.

- At the specified interval, `Ajax.PeriodicalUpdater` will create its own instance of `Ajax.Updater`—passing it the element to update, the URL to request, and any relevant options. For example, the `insertion` parameter is being passed to `Ajax.Updater`, which is why new content is being added to the bottom of `div#bucket` instead of replacing the old stuff.

OK, these `h2`s are getting on my nerves. Reload the page.

Controlling the Polling

When you set up a `PeriodicalUpdater`, you don't necessarily want it to keep running until the end of time. Prototype gives us a couple of tactics to regulate this constant flow of requests.

To demonstrate the first, we'll have to interact with the `PeriodicalUpdater` instance. So let's run our most recent line of code again, making sure to assign it to a variable:

```
var poller = new Ajax.PeriodicalUpdater('bucket', 'ajax.html', {
  method: 'get', insertion: 'bottom', frequency: 5
});
```

We could have been doing this all along, but only now do we need to refer to the Ajax object in subsequent lines of code.

The familiar steady stream of `h2`s is back, inexorably marching down the page like textual lemmings. But this time we can make them stop:

```
poller.stop();
```

Spend a few seconds staring at the screen, nervously wondering if you truly shut it all down. You'll eventually realize that the `h2`s have stopped, and no more Ajax requests are being logged to the Firebug console.

It's really that simple: `PeriodicalUpdater` has an instance method named `stop` that will put all that periodical updating on hold. There's a predictable complement to `stop`, and we'll use it right now to turn the `h2`s back on:

```
poller.start();
```

Their respite was short-lived—the `h2`s are back and growing in number every 5 seconds. Calling `start` on a stopped `PeriodicalUpdater` works a lot like creating a new one: the request is run immediately, and then scheduled to run again once the specified interval of time passes.

There's one more flow control strategy we can employ. It's called `decay`, but it's nowhere near as gross as it sounds—think atoms, not carcasses.

Let's reload `index.html` one more time and add the `decay` parameter to our `options` object:

```
var poller = new Ajax.PeriodicalUpdater('bucket', 'ajax.html', {
  method: 'get', insertion: 'bottom', frequency: 5, decay: 2
});
```

It will take a little longer to realize what's going on this time. Just like before, the first Ajax request is dispatched immediately. Then there's another request 5 seconds later. Then . . . wait. That time it felt more like 10 seconds. And now it's even longer. Is this thing *slowing down*?

After a few more cycles, you'll be able to figure out what's going on.

It is, in fact, slowing down. Our `decay` parameter is causing the interval between requests to double each time. (5 seconds, then 10, 20, 40, 80, etc.) If we changed `decay` to `3`, the interval would be tripled each time. In mathematics, this is called *exponential decay*. It has many applications across all fields of science, but here we're using it to make web pages *awesome*. The default value for `decay` is `1`—that is, by default there *is no* decay.

But *why* is it slowing down? Because it's getting the same response every time. `PeriodicalUpdater` keeps track of this, comparing the latest response to the previous response each time the updater runs. If the contents are different, all proceeds as normal; if the contents are identical, the interval gets multiplied by the `decay` parameter and the result is used to schedule the next updater. (In this example, of course, we're requesting a static HTML file, so each request is identical to the previous.) If, after the interval is lengthened, a fresh response comes back, it snaps back to the `frequency` that was originally set.

So `PeriodicalUpdater`s can be started, stopped, and decayed, abiding by your exacting rules of flow control. You'll need these rules someday. You probably didn't feel any dread at the prospect of HTML elements that reproduce infinitely, but you will feel dread when the server hosting your web app starts getting hit every 15 seconds by every single client using it. Responsiveness is good, and periodic client-server communication is good—but these benefits will eventually clash with the practical constraints of bandwidth and processing power. Knowing when to poll, when not to poll, and when to poll *less often* can be the antidote to the typical "chattiness" of Ajax-driven applications.

Advanced Examples: Working with Dynamic Content

We've already looked at a handful of simple examples of what Prototype's Ajax objects can do. But simple examples are boring. Let's get our feet wet.

Increasing the complexity means we'll have to introduce server-side scripting to the mix. These examples will use PHP, but the concepts are applicable no matter what your architecture.

Example 1: The Breakfast Log

Most of your Ajax calls will involve dynamic content, rather than the HTML and text files we've been using—the response will vary based on the data you send. You're probably already familiar with GET and POST—the two HTTP methods for sending data to the server—from working with HTML forms. Ajax can use either method to submit data.

For this set of examples, we'll be creating a "blog." If you don't know what a blog is, you're behind the times, my friend; it's short for "breakfast log," and it's a minute-by-minute account of which breakfast foods you've consumed on which dates and times. The trend is spreading like wildfire: at least half a dozen people on earth have breakfast logs.

The Server Side

We'll start with the server side, so that our page will have something to talk to. Create a file called `breakfast.php` (as shown in Listing 4-4) and put it in the same directory as `index.html`.

Listing 4-4. *The breakfast.php File*

```
<?php
// make a human-readable date for the response
$time      = date("g:i a \o\\n F j, Y", time());

$food_type = strip_tags($_REQUEST['food_type']);
$taste     = strip_tags($_REQUEST['taste']);
?>

<li>At <strong><?= $time ?></strong>, I ate <strong><?= $taste ?>
<?= $food_type ?></strong>.</li>
```

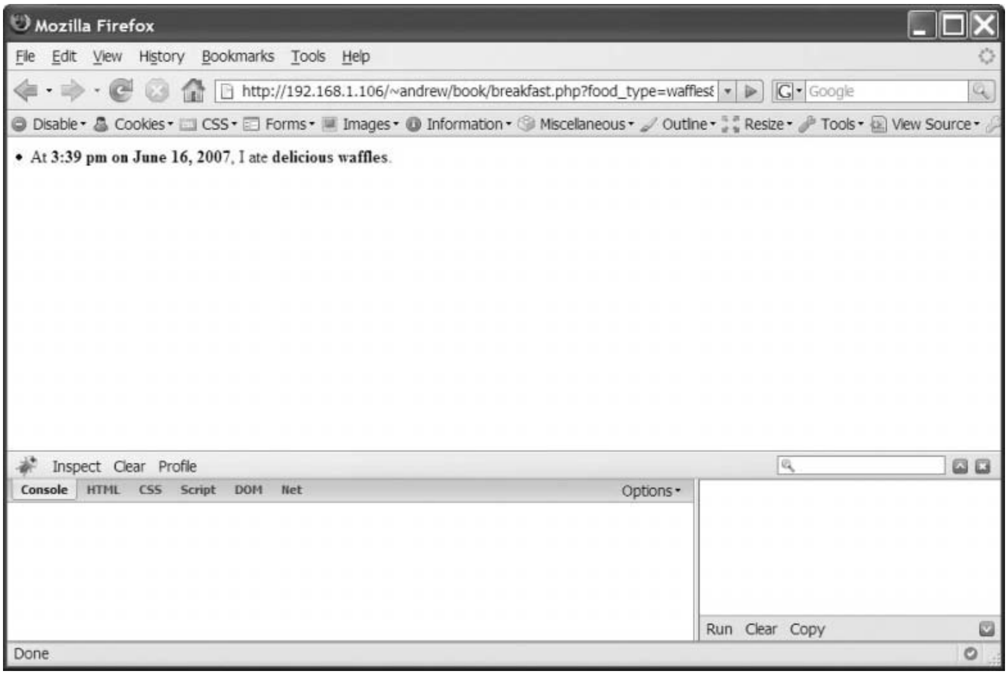
I've highlighted the lines that reference `$_REQUEST`, the global variable for looking up any query parameters given to a script. This script expects to receive two crucial pieces of data: the kind of food I ate and its taste quality. For now, we won't send the time of the meal—we'll simply assume that the breakfast logger ("blogger" for short) is posting from his or her kitchen table, mouth full of French toast. This lets us take the shortcut of using the server's time rather than relying on the client to provide it.

Our script takes the provided values for food and taste and strips them of HTML using PHP's `strip_tags` function. (Ordinarily, it would also save the values to a database, but that's not important for what we're doing here.) Then it prints a small fragment of HTML to describe to the browser what has just happened.

Let's make sure our script works. Open a browser and navigate to wherever you put `breakfast.php` on your server. But remember, we've got to tell it what we ate and how delicious it was. So we need to add a query string to the end of the URL. Yours should look something like this:

```
http://your-server.dev/breakfast.php?food_type=waffles&taste=delicious
```

Press return, and you ought to see your HTML fragment in the browser window, as shown in Figure 4-9.


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/book/breakfast.php?food_type=waffles
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tools View Source
Inspect Clear Profile
Console HTML CSS Script DOM Net Options
Run Clear Copy
Done

- At **3:39 pm** on **June 16, 2007**, I ate **delicious waffles**.

Figure 4-9. *The HTML fragment that represents our delicious meal*

No errors! Emboldened by this programming victory, let's go back to `index.html` to make it look more like a breakfast log.

The Client Side

Our HTML page is no longer generic—it's got a purpose! Let's make it friendlier. Make these changes to `index.html`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <title>Andrew's Breakfast Log</title>

    <script src="prototype.js" type="text/javascript"></script>
  </head>
  <body>
    <h1>Andrew's Breakfast Log</h1>
    <ul id="breakfast_history"></div>
  </body>
</html>
```

It's still ugly and sparse, but at least it's got a human touch now.

We're going to keep a list on this page, so let's treat it as such. We've changed our container `div` to a `ul`, a proper container for `li`s, and given it a more descriptive ID.

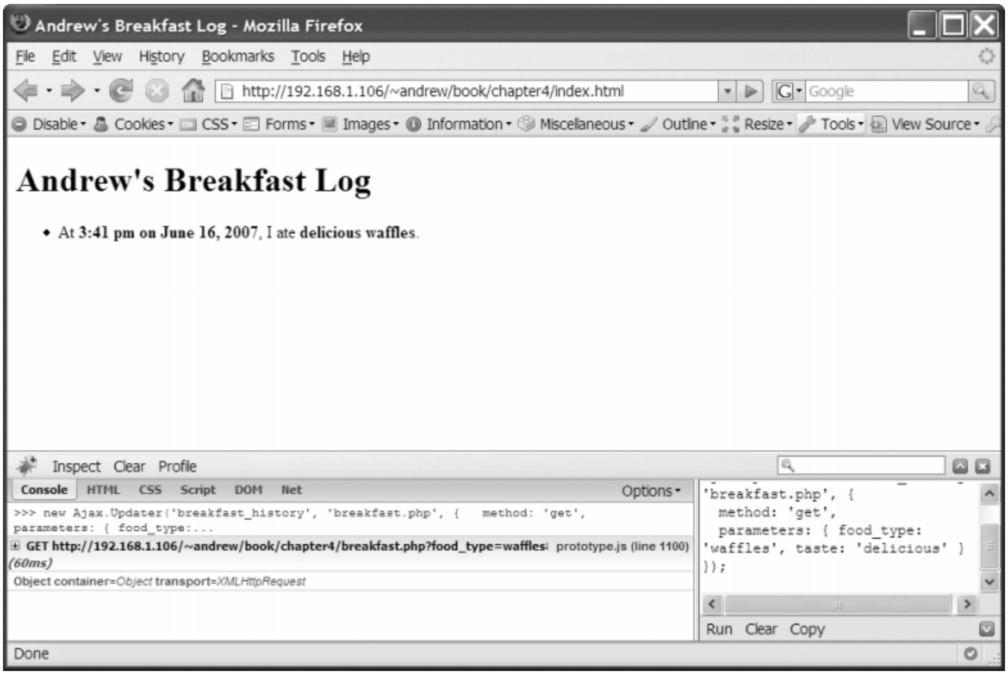
Now we're ready to record our meals for posterity! Reload `index.html` in Firefox, and then type this into the Firebug console:

```
new Ajax.Updater('breakfast_history', 'breakfast.php', { method:'get',
  parameters: { food_type: 'waffles', taste: 'delicious' }
});
```

You should recognize the highlighted line—we're sending these name/value pairs along with our request. Our script gets the message, saves it to a database (presumably), and then gives us some HTML to put on the page.

Also, notice how we've removed the `method` parameter from the options. We could explicitly set it to `"post"`, but since that's the default we're better off omitting it altogether.

Run this code. The result should look like Figure 4-10.


Andrew's Breakfast Log - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/book/chapter4/index.html
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tools View Source
Inspect Clear Profile
Console HTML CSS Script DOM Net
Options
Run Clear Copy
Done

Andrew's Breakfast Log

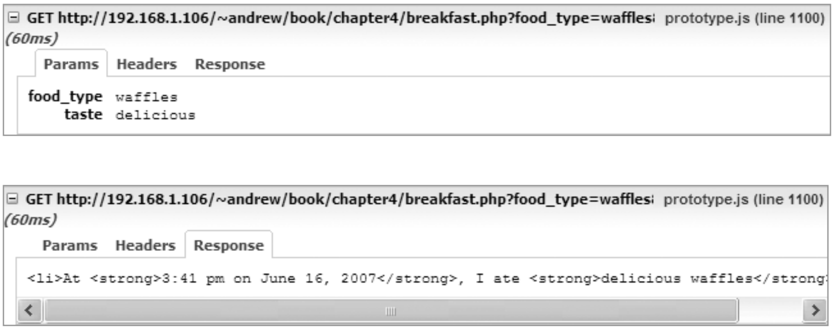
- At 3:41 pm on June 16, 2007, I ate delicious waffles.

```
>>> new Ajax.Updater('breakfast_history', 'breakfast.php', {   method: 'get',
parameters: { food_type:...
⊞ GET http://192.168.1.106/~andrew/book/chapter4/breakfast.php?food_type=waffles   prototype.js (line 1100)
(60ms)
Object container=Object transport=XMLHttpRequest
```

```
'breakfast.php', {
  method: 'get',
  parameters: { food_type:
'waffles', taste: 'delicious' }
});
```

Figure 4-10. *The fragment from the previous figure has been placed on the page.*

Since Firebug logs all Ajax requests, you can see for yourself. Near the bottom of your console should be a gray box containing the URL of the request; expand this box to view all the request's details, as depicted in Figure 4-11.

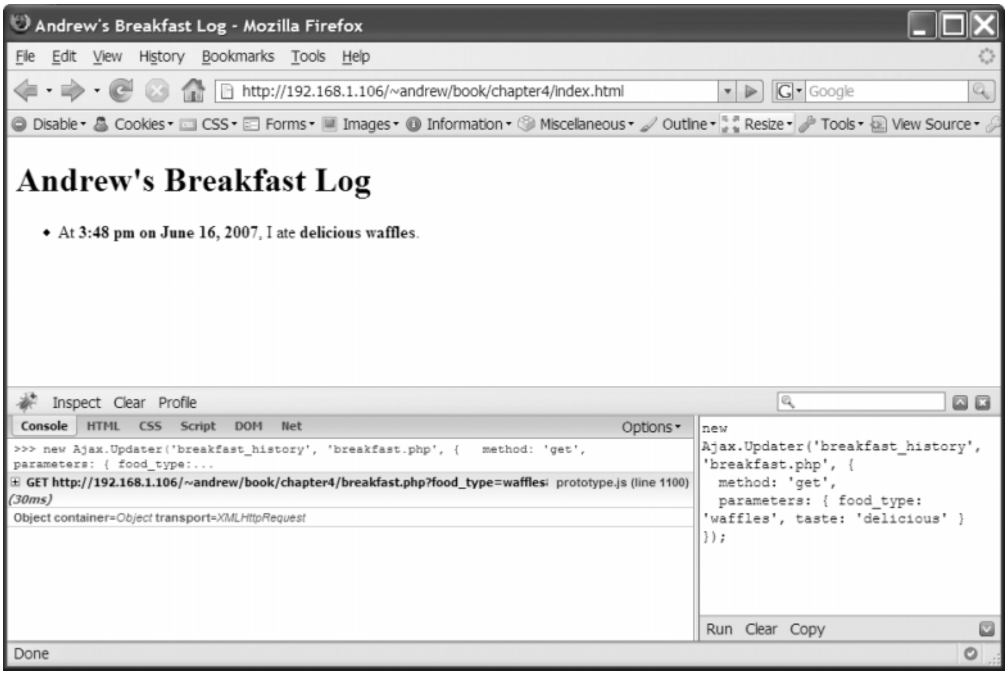


```
⊟ GET http://192.168.1.106/~andrew/book/chapter4/breakfast.php?food_type=waffles  prototype.js (line 1100)
(60ms)
   Params   Headers   Response

   food_type  waffles
       taste  delicious
```

```
⊟ GET http://192.168.1.106/~andrew/book/chapter4/breakfast.php?food_type=waffles  prototype.js (line 1100)
(60ms)
   Params   Headers   Response

   <li>At <strong>3:41 pm on June 16, 2007</strong>, I ate <strong>delicious waffles</strong
```

Figure 4-11. *The details of our Ajax request*

That was fun. Let's try it again—run the exact same command in the console (see Figure 4-12).


Andrew's Breakfast Log - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/book/chapter4/index.html
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tools View Source
Inspect Clear Profile
Console HTML CSS Script DOM Net Options
Run Clear Copy
Done

Andrew's Breakfast Log

- At **3:48 pm on June 16, 2007**, I ate **delicious waffles**.

```
>>> new Ajax.Updater('breakfast_history', 'breakfast.php', {   method: 'get',
parameters: { food_type:...
⊞ GET http://192.168.1.106/~andrew/book/chapter4/breakfast.php?food_type=waffles  prototype.js (line 1100)
(30ms)
Object container=Object transport=XMLHttpRequest
```

```
new
Ajax.Updater('breakfast_history',
'breakfast.php', {
  method: 'get',
  parameters: { food_type:
'waffles', taste: 'delicious' }
});
```

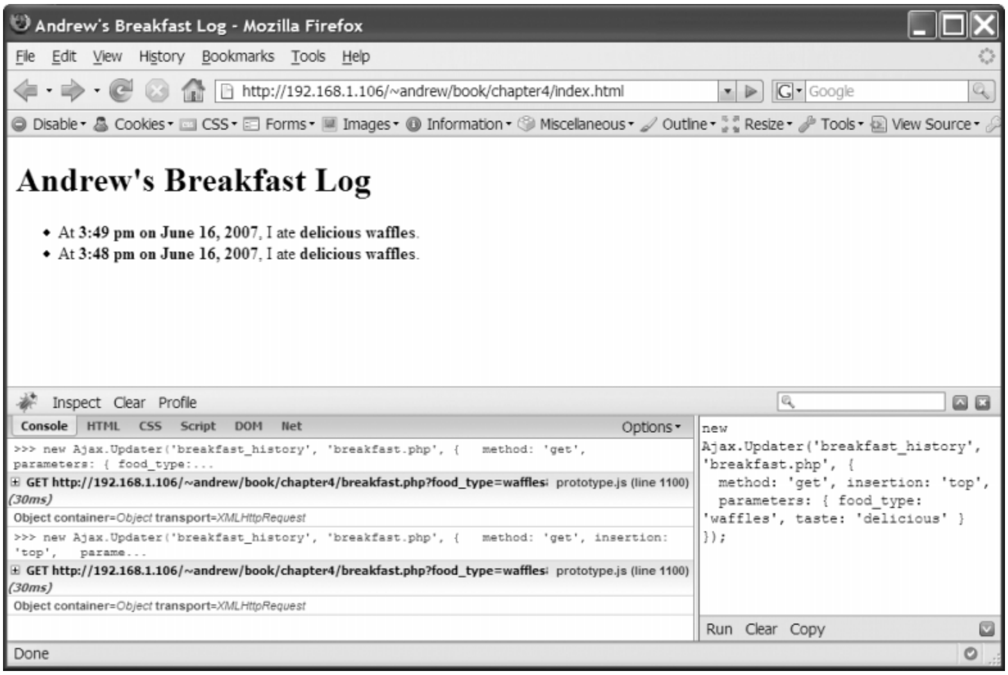
Figure 4-12. *What happened to the first one?*

Figure 4-12 is not quite what we expected. The time is different, so the content got replaced properly. But we don't want to *replace* the contents of `ul#breakfast_history`; we want to *add* to what's already there.

Typically, breakfast log entries are arranged so that the most recent is first. So let's change our Ajax call so that new entries are appended to the top of the container:

```
new Ajax.Updater('breakfast_history', 'breakfast.php', {
  insertion: 'top', method: 'get',
  parameters: { food_type: 'waffles', taste: 'delicious' }
});
```

Run this code and you'll see your new entry added to the top of the list, as in Figure 4-13. Each time you run this code, in fact, a new `li` will be added to the top of your `ul` container.


Andrew's Breakfast Log - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/book/chapter4/index.html
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tools View Source
Inspect Clear Profile
Console HTML CSS Script DOM Net
Options
Run Clear Copy
Done

Andrew's Breakfast Log

- At **3:49 pm on June 16, 2007**, I ate **delicious waffles**.
- At **3:48 pm on June 16, 2007**, I ate **delicious waffles**.

```
>>> new Ajax.Updater('breakfast_history', 'breakfast.php', {   method: 'get',
parameters: { food_type:...
⊞ GET http://192.168.1.106/~andrew/book/chapter4/breakfast.php?food_type=waffles  prototype.js (line 1100)
(30ms)
Object container=Object transport=XMLHttpRequest
>>> new Ajax.Updater('breakfast_history', 'breakfast.php', {   method: 'get', insertion:
'top',   parame...
⊞ GET http://192.168.1.106/~andrew/book/chapter4/breakfast.php?food_type=waffles  prototype.js (line 1100)
(30ms)
Object container=Object transport=XMLHttpRequest
```

```
new
Ajax.Updater('breakfast_history',
'breakfast.php', {
  method: 'get', insertion: 'top',
  parameters: { food_type:
'waffles', taste: 'delicious' }
});
```

Figure 4-13. *New entries appear at the top.*

Handling Errors

Our `breakfast.php` script works, but it's not exactly battle-tested. It naively assumes that each request will have the two pieces of information it wants. What if something goes wrong? What if our scrambled eggs fail to be tabulated? We need to work out some sort of code between client and server to handle situations like these.

Actually, it's been worked out for us. Each HTTP request has a status code that indicates whether everything went well or not. The standard success response is `200`, which means "OK," although most web surfers are more familiar with `404` (File Not Found), since one usually isn't shown the status code until something goes wrong.

The first digit of an HTTP status code tells you what kind of message this is going to be. Codes starting with 2 are all various forms of success codes, 3 signifies a redirect, 4 means the request was faulty somehow, and 5 means that the server encountered an error.

This is just what we need. Our script can check for the presence of `food_type` and `taste` as query parameters. If it doesn't find them, it can return an error status code instead of the typical `200`. And it can use the content of the response to present a friendlier error message to the user.

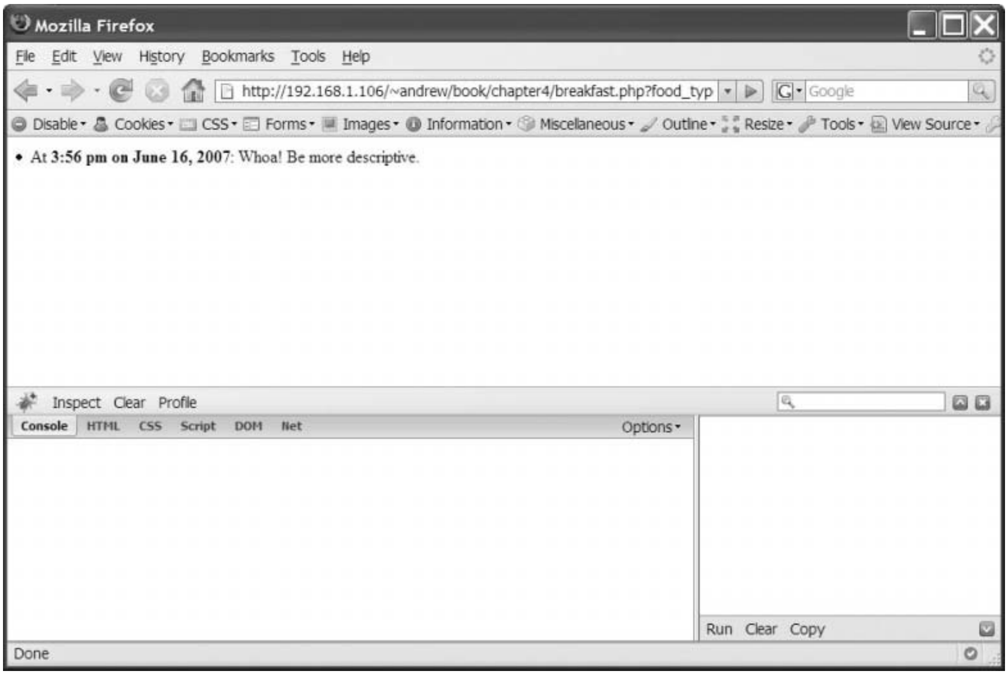
PHP lets us do this rather easily.

```
<?php
// make a human-readable date for the response
$time      = date("g:i a \o\\n F j, Y", time());

if (!isset($_REQUEST['food_type']) || !isset($_REQUEST['taste'])) {
  header('HTTP/1.0 419 Invalid Submission');
  die("<li>At <strong>${time}</strong>: Whoa! Be more descriptive.</li>");
}

$food_type = strip_tags($_REQUEST['food_type']);
$taste     = strip_tags($_REQUEST['taste']);
?>
<li>At <strong><?= $time ?></strong>, I ate <strong><?= $taste ?>
<?= $food_type ?></strong>.</li>
```

The `419` error code isn't canonical—we just made it up. But Apache delivers this code just fine, and Prototype properly recognizes it as an error code.

Test this in your browser. Open up `breakfast.php` directly, just like you did before—but this time leave one of the parameters out of the URL (see Figure 4-14):

```
http://your-server.dev/breakfast.php?food_type=waffles
```

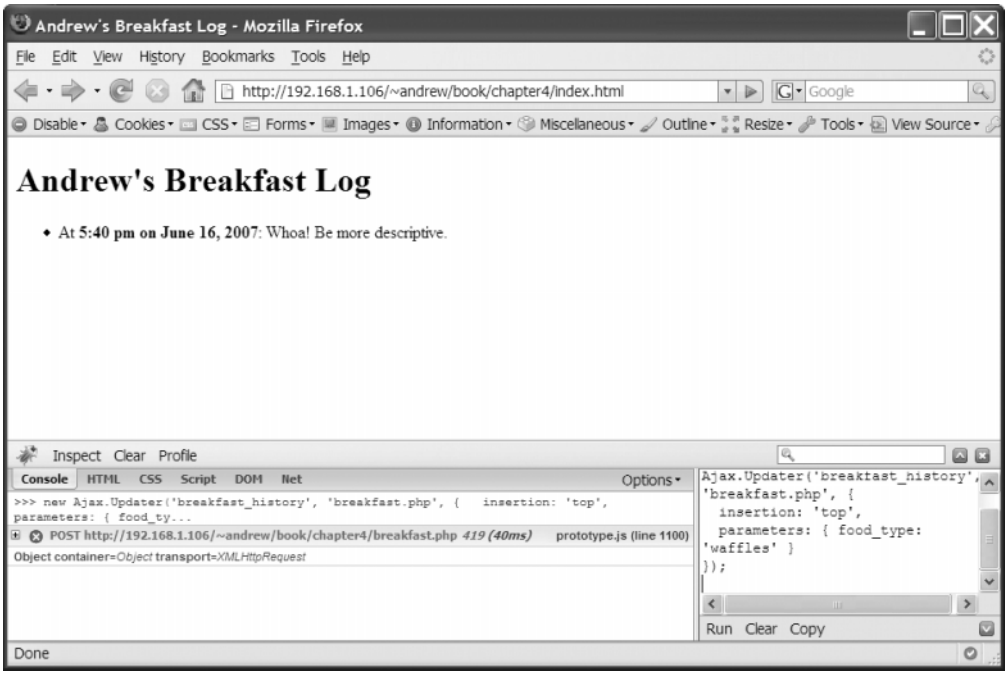

Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/book/chapter4/breakfast.php?food_typ
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tools View Source
Inspect Clear Profile
Console HTML CSS Script DOM Net Options
Run Clear Copy
Done

- **At 3:56 pm on June 16, 2007**: Whoa! Be more descriptive.

Figure 4-14. *Our error is a success!*

As expected, the response (shown in Figure 4-14) tells us that we weren't forthcoming enough about the waffles we just ate. We can't actually see the status code this way, but we can if we request the URL through Ajax instead. So go back to `index.html` and run the `Ajax.Updater` call once more—but this time remove one of the `parameters` from the `options` hash. (Figure 4-15 shows the result.)

```
new Ajax.Updater('breakfast_history', 'breakfast.php', {
  insertion: 'top',
  parameters: { food_type: 'waffles' }
});
```


Andrew's Breakfast Log - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/book/chapter4/index.html
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tools View Source
Inspect Clear Profile
Console HTML CSS Script DOM Net
Options
Run Clear Copy
Done

Andrew's Breakfast Log

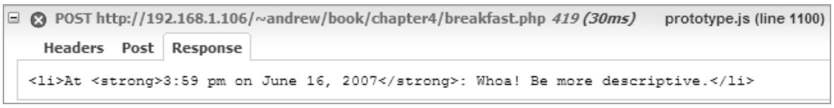
- **At 5:40 pm on June 16, 2007**: Whoa! Be more descriptive.

```
>>> new Ajax.Updater('breakfast_history', 'breakfast.php', {   insertion: 'top',
parameters: { food_ty...
POST http://192.168.1.106/~andrew/book/chapter4/breakfast.php 419 (40ms)    prototype.js (line 1100)
Object container=Object transport=XMLHttpRequest
```

```
Ajax.Updater('breakfast_history',
'breakfast.php', {
  insertion: 'top',
  parameters: { food_type:
'waffles' }
});
```

Figure 4-15. *Proper error reporting*

Firebug, ever helpful, shows you that something went wrong with the request—it makes the URL red and adds the status code to the end of the line, as shown in Figure 4-16.



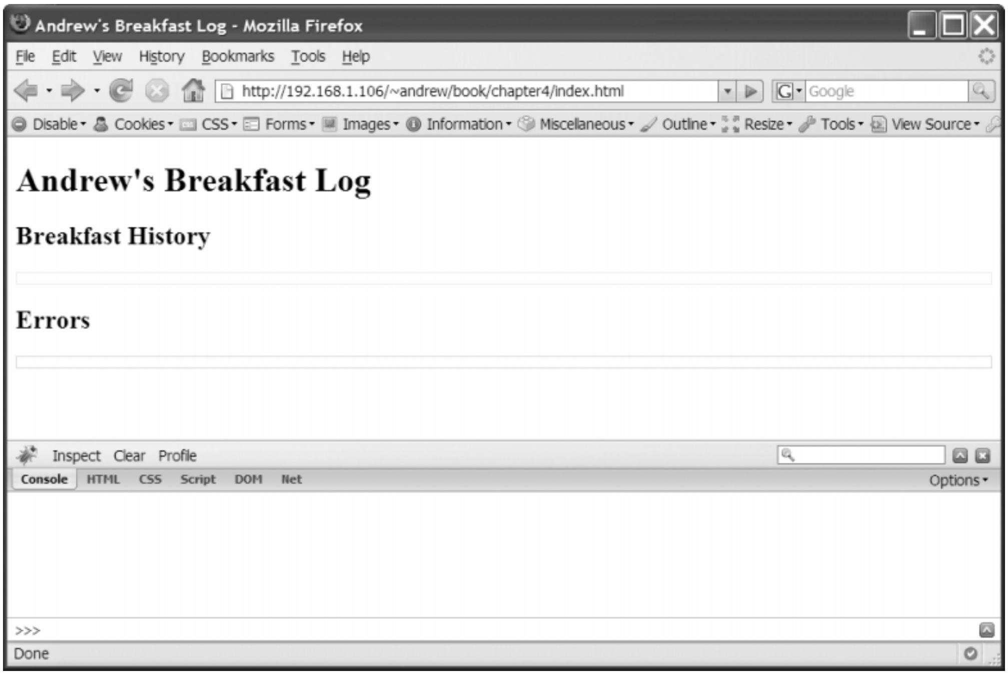
```
POST http://192.168.1.106/~andrew/book/chapter4/breakfast.php 419 (30ms)    prototype.js (line 1100)
<li>At <strong>3:59 pm on June 16, 2007</strong>: Whoa! Be more descriptive.</li>
```

Figure 4-16. *Firebug shows the status code when an error occurs.*

So our omission of taste information is being reported as an error, just like we want. But our JavaScript code doesn't yet treat errors differently from successful responses. We need to separate the two if we want errors to stand out to the user.

So let's create a new `ul`, this one for errors. We can style the two containers differently and give them headings so that the user knows they're two different groups of things. Make these changes to `index.html` and view the results in Figure 4-17:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <title>Andrew's Breakfast Log</title>
    <style type="text/css" media="screen">
      #breakfast_history {
        color: green;
        border: 1px solid #cfc;
        padding: 5px 0 5px 40px;
      }
      #error_log {
        color: red;
        border: 1px solid #edd;
        padding: 5px 0 5px 40px;
      }
    </style>

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

  <body>
    <h1>Andrew's Breakfast Log</h1>

    <h2>Breakfast History</h2>
    <ul id="breakfast_history"></ul>

    <h2>Errors</h2>
    <ul id="error_log"></div>
  </body>
</html>
```


Andrew's Breakfast Log - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/book/chapter4/index.html
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tools View Source
Inspect Clear Profile
Console HTML CSS Script DOM Net
Options
Done

Andrew's Breakfast Log

Breakfast History

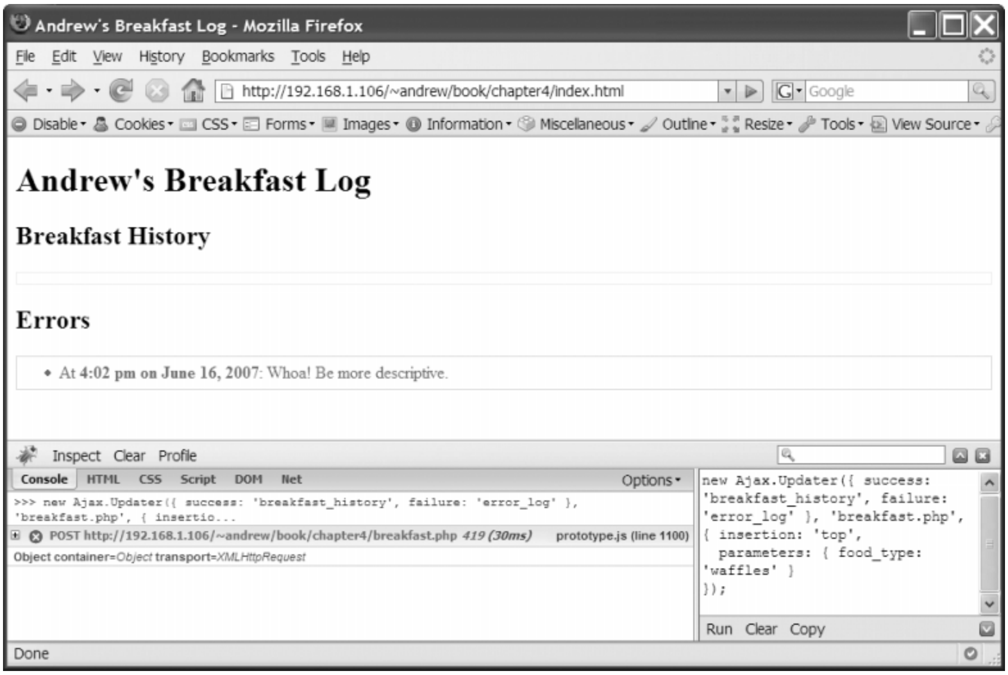
Errors

Figure 4-17. *Slightly less ugly*

We're almost there. The last part is the easiest because it's built into `Ajax.Updater`. Instead of designating one container to update, you can designate two: one for successful requests and one for unsuccessful requests. The conventions followed by HTTP status codes make it easy to figure out what's an error and what's not.

```
new Ajax.Updater({ success: 'breakfast_history', failure: 'error_log' },
'breakfast.php', { insertion: 'top',
  parameters: { food_type: 'waffles' }
});
```

Victory! As Figure 4-18 shows, bad requests show up bright red in the error log.


Andrew's Breakfast Log - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/book/chapter4/index.html
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tools View Source
Inspect Clear Profile
Console HTML CSS Script DOM Net Options
Run Clear Copy
Done

Andrew's Breakfast Log

Breakfast History

Errors

- At 4:02 pm on June 16, 2007: Whoa! Be more descriptive.

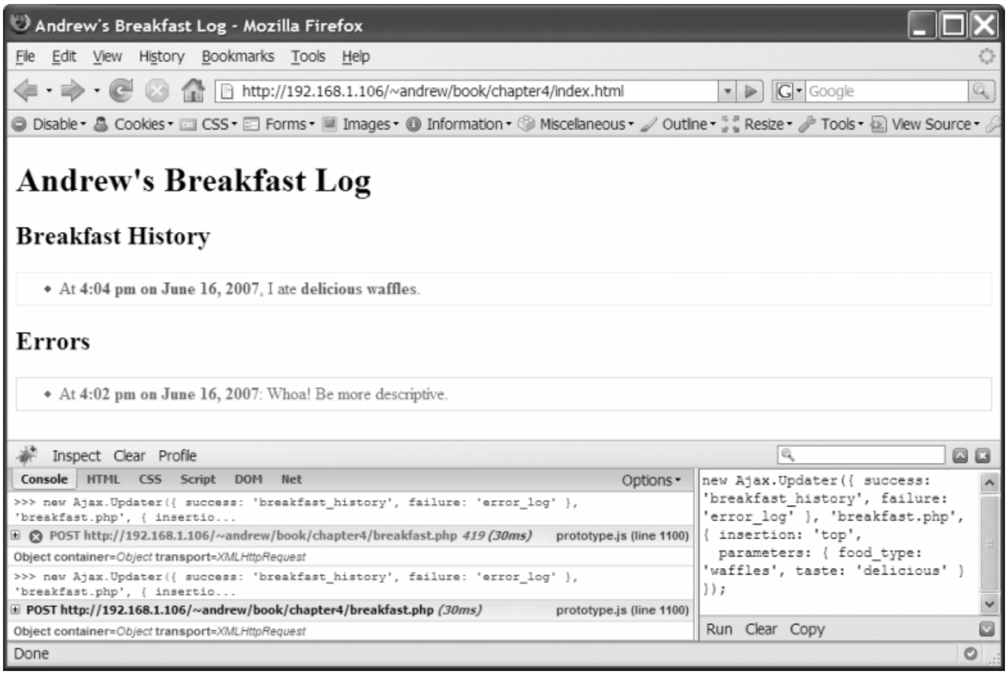
```
>>> new Ajax.Updater({ success: 'breakfast_history', failure: 'error_log' },
'breakfast.php', { insertio...
POST http://192.168.1.106/~andrew/book/chapter4/breakfast.php 419 (30ms)    prototype.js (line 1100)
Object container=Object transport=XMLHttpRequest
```

```
new Ajax.Updater({ success:
'breakfast_history', failure:
'error_log' }, 'breakfast.php',
{ insertion: 'top',
  parameters: { food_type:
'waffles' }
});
```

Figure 4-18. *Errors now look like errors.*

Now put that `taste` parameter back into the statement and watch your valid request appear in a pleasing green color, as shown in Figure 4-19.

```
new Ajax.Updater({ success: 'breakfast_history', failure: 'error_log' },
'breakfast.php', { insertion: 'top',
  parameters: { food_type: 'waffles', taste: 'delicious' }
});
```


Andrew's Breakfast Log - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/book/chapter4/index.html
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tools View Source
Inspect Clear Profile
Console HTML CSS Script DOM Net Options
Run Clear Copy
Done

Andrew's Breakfast Log

Breakfast History

- At 4:04 pm on June 16, 2007, I ate delicious waffles.

Errors

- At 4:02 pm on June 16, 2007: Whoa! Be more descriptive.

```
>>> new Ajax.Updater({ success: 'breakfast_history', failure: 'error_log' },
'breakfast.php', { insertio...
POST http://192.168.1.106/~andrew/book/chapter4/breakfast.php 419 (30ms)    prototype.js (line 1100)
Object container=Object transport=XMLHttpRequest
>>> new Ajax.Updater({ success: 'breakfast_history', failure: 'error_log' },
'breakfast.php', { insertio...
POST http://192.168.1.106/~andrew/book/chapter4/breakfast.php (30ms)    prototype.js (line 1100)
Object container=Object transport=XMLHttpRequest
```

```
new Ajax.Updater({ success:
'breakfast_history', failure:
'error_log' }, 'breakfast.php',
{ insertion: 'top',
  parameters: { food_type:
'waffles', taste: 'delicious' }
});
```

Figure 4-19. *Valid entries now look valid.*

Example 2: Fantasy Football

It's time to talk about the site we'll be building together over the course of this book. JavaScript has a place on many different kinds of sites, but web *applications* are especially ripe for applying the topics we'll cover in the pages ahead. So let's build a site that people need to *use*, rather than just read. We'll build the sort of site that Ajax can have the greatest impact on.

The App

For the next few chapters, we'll be building a *fantasy football* site. For those who don't know (apologies to my non-American readers), fantasy football is a popular activity among followers of professional American football. Here's fantasy football in a nutshell:

- A group of friends will form a league, “draft” certain real-life players, and earn points each week based on the in-game performances of these players. The members of the league, called “owners,” have duties much like the coaches and managers of real-life football teams: they can make trades, sign free agents, and decide who plays and who sits on the bench (“starters” and “reserves”).

- Each week, a fantasy team will compete against another fantasy team to see who can score more points. (Nearly all pro football games are played on Sunday; nearly all pro teams play every single week.) Owners must decide who to start by predicting which of their players will score the most points. This can vary from week to week based on real-life match-ups, injuries, and other factors.

- A fantasy team earns points whenever one of its starters does something notable in his real-life game. For instance, when a player scores a touchdown in a game, any fantasy football owners who started him that week will earn points. Players also earn points for rushing yardage (advancing the ball on the ground), passing yardage (advancing the ball through the air), and field goals (kicking the ball through the uprights). Scoring systems vary from league to league, but these activities are the ones most often rewarded.

- Fantasy football has been around for several decades, but was far more tedious before computers; owners would have to tabulate their scores manually by reading game results in the newspaper. The Web eliminated this chore, thus causing an explosion in the game’s popularity. The National Football League (NFL), America’s top professional league, even offers free fantasy football on its own web site.

If none of this makes sense to you, don’t worry. All you need to know is that we’re building a web application that will need plenty of the bells and whistles we’ll be covering in the chapters ahead.

The League

Most “reliable sources” state that American football originated in the United Kingdom sometime in the 1800s as an offshoot of rugby. This assertion, while “true,” is breathtakingly dull. Wouldn’t it be much more interesting if American football had, in fact, been conceived of by the nation’s *founding fathers*? Wouldn’t you be fascinated to learn that, during the framing of the Constitution, members of the Congress of the Confederation often resolved disputes by advancing an oblong ball down a grassy field on the outskirts of Philadelphia?

There are no surviving documents that tell us the makeup of the teams, nor the outcomes of the games, but a thorough reading of the US Constitution gives us hints. And so every year a small group of New England football lovers holds reenactments of these games, complete with costumes and pseudonyms, to pay tribute to the unrecognized progenitors of American football. (And you thought Civil War reenactments were crazy.)

This folk tale, even though I just made it up, allows us to proceed with the fantasy football concept without using the names of *actual* football players—thus sparing me the wrath of the NFL Players' Association, which regulates the use of NFL players' names. So it's a win-win scenario. I avoid a lawsuit; you receive a fun history lesson.

The Scoring

Scoring varies from league to league, but leagues tend to award points when a player does something good for his team. Thus, the players with the best individual statistics are often the highest-scoring fantasy players. Some leagues also subtract points when a player makes a mistake in a game—throwing an interception, for instance, or fumbling the ball.

For simplicity's sake, our league will feature six starting slots per team: one quarterback, two running backs, two wide receivers, and one tight end. These are all offensive players that accrue stats in a way that's easy to measure.

Table 4-1 shows the tasks that will earn points in our league.

Table 4-1. *League Scoring Table*

Task	Typical Position	Points
Throwing a touchdown pass	QB (quarterback)	4
Catching a touchdown pass	WR (wide receiver), TE (tight end), RB (running back)	6
Rushing for a touchdown	RB, QB	6
Every 25 passing yards	QB	1
Every 10 rushing yards	RB, QB	1
Every 10 receiving yards	WR, TE, RB	1

The Stats

And now, finally, we come to this chapter's task. The most important page on a fantasy football site is the *box score* page—the page that shows the score for a given match-up.

Our target user, the avid fantasy football owner, keeps a close eye on the box score every Sunday during football season. He needs a page that can do the following things:

1. Show him the score of the game in a way that he can tell whether he's winning or losing at a glance.

2. Show him his roster and his opponent's roster. Alongside each player should be the number of points he's scored *and* a summary of what he's done in his pro game to earn those points. When the owner's score changes, he should be able to tell which of his players just earned points for him.

3. Show him the scores of the other games in his league. Clicking a particular score should take him to the match-up for that game, where he can view the results in greater detail.

4. Keep the score current *without needing a page refresh*. This page will be open all day.

The first three requirements need to be addressed in the interface. We'll come back to those in the next few chapters. But the last one—updating the content without refreshing—is, quite literally, what Ajax was made for.

Let's translate this into technical requirements:

1. We'll need to use Ajax to refresh the game stats without reloading the page. This means we'll also need a URL that, when requested, will return game stats.

2. The client will send out an Ajax request every 30 seconds to ask the server for new stats.

3. The whole site runs on stats, so other pages we build will *also* need this information. So the data should be sent in an abstract format. The client-side code we write will determine how to format the data for display.

4. For the preceding reason, the client needs to know how to interpret the data it receives. Let's have the server return the stats as JSON; it will be easy to parse in JavaScript.

5. The client should figure out when and how to alert the user to score changes. In other words, it's the client's job to compare the new stats to the previous stats to figure out what's different.

Knowing what we need is one thing; putting it all together is another. Naturally, we'll be focusing on the client side, but the client needs to talk to *something* so that we can ensure our code works right.

But we don't have any real stats; our fictional league's season hasn't started yet. So we'll have to fake it.

Mocking

Think of the client and server as humans. The JavaScript we write deals only with the browser; to make an Ajax call, our code has to ask the browser to make an external request and hand over the response when it's ready. Imagine Alice walking into a room and telling Bob to make a phone call for her. Bob calls Carol, has a quick conversation, and then hangs up and tells Alice what Carol said.

But Alice has no *direct* contact with Carol. If Bob wanted to, he could turn around, pick up the phone, *pretend* to dial a number, *pretend* to have a conversation, hang up, and tell Alice whatever he feels like. The whole thing's a ruse, but Alice is none the wiser.

In the real world, this would be dishonest and unwise behavior. In the computer world, it helps us build our app faster. Lying to your code shouldn't make you feel bad.

The code we write will make a request to a certain URL every 30 seconds; it will expect a JSON response that follows a certain format. But it doesn't need to know *how* those stats are retrieved; that's the job of the server.

For testing purposes, we need a stream of data that behaves the way a *real* stream would during game day: games start out with no score, but points are amassed over time as real-life players do good things.

So let's write a script that will generate some *mock stats* for us. This script can be told which stats to report at any given time. It can behave the way real stats would.

We'll be using PHP, but the basic idea is the same for any language. Here's how it will work:

1. We define several classes—one or two for each position. These will represent the players. Each will score at a different rate.

2. We'll have stats running on a 10-minute cycle, after which every score will reset. This may seem like a short cycle—and, indeed, it's much shorter than a real football game—but a shorter cycle only helps us develop more quickly.

3. We'll tell each class how to report its own score based on the current time and the pace of scoring we've set.

We'll write as little code as possible in order to get this done. This won't be a part of the final site, so it doesn't have to be pretty; it just has to work.

The Data

Since PHP 5.2, when the JSON module was first included in a default PHP installation, it has been easy to encode and decode JSON in PHP. The `json_encode` function is part of the core language:

```
$data = array(
  "QB" => "Alexander Hamilton",
  "RB" => "John Jay",
  "WR" => "James Madison"
);

json_encode($data);
//-> '{ "QB": "Alexander Hamilton", "RB": "John Jay", "WR": "James Madison" }'
```

This works with nested arrays as well (arrays that contain arrays):

```
$teams = array(
  "team1" => array(
    "QB" => "Alexander Hamilton",
    "RB" => "John Jay",
    "WR" => "James Madison"
  ),
  "team2" => array(
    "QB" => "George Washington",
    "RB" => "John Adams",
    "WR" => "John Hancock"
  )
);

json_encode($teams);

//-> '{
//->   "team1": {
//->     "QB": "Alexander Hamilton",
//->     "RB": "John Jay",
//->     "WR": "James Madison"
//->    },
//->   "team2": {
//->     "QB": "George Washington",
//->     "RB": "John Adams",
//->     "WR": "John Hancock"
//->    }
//->  }'
```

This is great news. It means we can create a data structure using nested associative arrays (PHP's equivalent to JavaScript's `Object` type), waiting until the very last step to convert it to JSON.

So let's decide on a structure for the data we'll receive. Hierarchically, it would look something like this:

- Team 1
    - Score
    - Players
        - Yards
        - Touchdowns
        - Score
        - Summary
- Team 2
    - Score
    - Players
        - Yards
        - Touchdowns
        - Score
        - Summary

In pure JavaScript, we would build this with nested object literals. Since we're in PHP, though, we'll use associative arrays.

If you're using a different server-side language, don't worry—JSON libraries exist for practically every commonly used programming language. The concept is the same.

The Code

Let's figure out how to keep track of the time, since that's the most important part. We don't care what the time is in *absolute* terms; we just care about setting milestones every 10 minutes, and then checking how long it has been since the last milestone. Sounds like a job for the modulus operator.

```
// figure out where we are in the 10-minute interval
$time = time() % 600;
```

PHP's `time` function gives us a UNIX timestamp (the number of seconds since January 1, 1970). There are 600 seconds in 10 minutes, so we divide the timestamp value by 600 and take the remainder. This will give us a value between 0 and 599.

First, we grab the timestamp. All we're looking for is a number from 0 to 600 (telling us where we are in the 600-second cycle), so we'll use the modulus operator on a standard UNIX timestamp.

All player classes will need this value, so we'll write a base `Player` class that will define the `time` instance variable.

```
class Player {
  var $time;
  function Player() {
    global $time;
    $this->time = $time;
  }
}
```

In PHP, we make a constructor by defining a function with the same name as its class. So the `Player` function will get called whenever we declare a new `Player` (or any class that descends from `Player`). All this constructor does is store a local copy of `$time`. (Pulling in `$time` as a global is a little sloppy, but it's quicker.)

Now, by writing position-specific classes that extend `Player`, we can do different things with the time variable in order to report different stats. These classes will have two things in common:

- Each will define a `stats` method that will return the player's stats thus far in the 10-minute cycle.

- The stats will be returned in array form, with fields for yards, touchdowns, fantasy points scored, and a text summary of the player's performance. This structure will be converted to JSON when it's sent over the pipeline.

A quarterback would slowly accrue passing yards over a game—with the occasional touchdown pass in between. Since we're compressing a whole game's statistics into a 10-minute period, we should set a faster pace.

```
// QB throws for 10 yards every 30 seconds
// and a touchdown every 4 minutes.
class QB extends Player {
  function stats() {
    $yards = floor($this->time / 30) * 10;
    $tds   = floor($this->time / 240);
    return array(
      "yards"  => $yards,
      "TD"     => $tds,
      "points" => floor($yards / 25) + (4 * $tds),
      "summary" => $yards . " yards passing, " . $tds . " TD"
    );
  }
}
```

We're extending the `Player` class, so we get its constructor for free. All we have to define, then, is the `stats` method. To get a score from this class, you need only declare a new instance and call this method.

```
$time = 430; // let's say
$qb = new QB();
$qb->score ();

//-> array(
//->  "yards"   => 140
//->  "TD"      => 1,
//->  "points"  => 9,
//->  "summary" => "140 yards passing, 1 TD"
//-> )
```

Now we'll do the same for the running back and wide receiver. But, since a team starts two running backs and two wide receivers (as opposed to starting one quarterback), we should make two different classes for each of these positions. That way, by mixing and matching which combinations start for which team, we can introduce some variation in the scoring.

```
// RB1 runs for 5 yards every 30 seconds and scores at minute #6.
class RB1 extends Player {
  function stats() {
    $yards = floor($this->time / 30) * 5;
    $tds   = floor($this->time / 360);
    return array(
      "yards"   => $yards,
      "TD"      => $tds,
      "points"  => floor($yards / 10) + (6 * $tds),
      "summary" => $yards . " yards rushing, " . $tds . " TD"
    );
  }
}

// RB2 runs for 5 yards every 40 seconds and does not score.
class RB2 extends Player {
  function stats() {
    $yards = floor($this->time / 40) * 5;

    return array(
      "yards"   => $yards,
      "TD"      => 0,
      "points"  => floor($yards / 10),
      "summary" => $yards . " yards rushing, 0 TD"
    );
  }
}

// WR makes one catch every minute for 15 yds and scores at minute #4.
class WR1 extends Player {
  function stats() {
    $yards = floor($this->time / 60) * 15;
    $tds   = $this->time > 240 ? 1 : 0;

    return array(
      "yards"   => $yards,
      "TD"      => $tds,
      "points"  => floor($yards / 10) + (6 * $tds),
      "summary" => $yards . " yards receiving, " . $tds . " TD"
    );
  }
}
```

```
// WR makes one catch every 2 minutes for 25 yds and does not score.
class WR2 extends Player {
  function stats() {
    $yards = floor($this->time / 120) * 25;

    return array(
      "yards"   => $yards,
      "TD"      => 0,
      "points"  => floor($yards / 10),
      "summary" => $yards . " yards receiving, 0 TD"
    );
  }
}
```

These classes all return data in the same format. They only differ in the "script" they follow—the way they turn that original `$time` value into a point total.

Each team will start only one tight end, so we needn't bother with more than one "version" of tight end.

```
// TE makes one catch at minute #8 for a 20-yard TD.
class TE extends Player {
  function stats() {
    $yards = $this->time > 480 ? 20 : 0;
    $tds   = $this->time > 480 ? 1 : 0;

    return array(
      "yards"   => $yards,
      "TD"      => $tds,
      "points"  => floor($yards / 10) + (6 * $tds),
      "summary" => $yards . " yards receiving, " . $tds . " TD"
    );
  }
}
```

There's only one thing left to do: organize these players into teams. At the bottom of `scores.php`, we'll add the code to do this and output to JSON.

```
// Adds a player's score to a running total; used to
// compute a team's total score
function score_sum($a, $b) {
  $a += $b["points"];
  return $a;
}
```

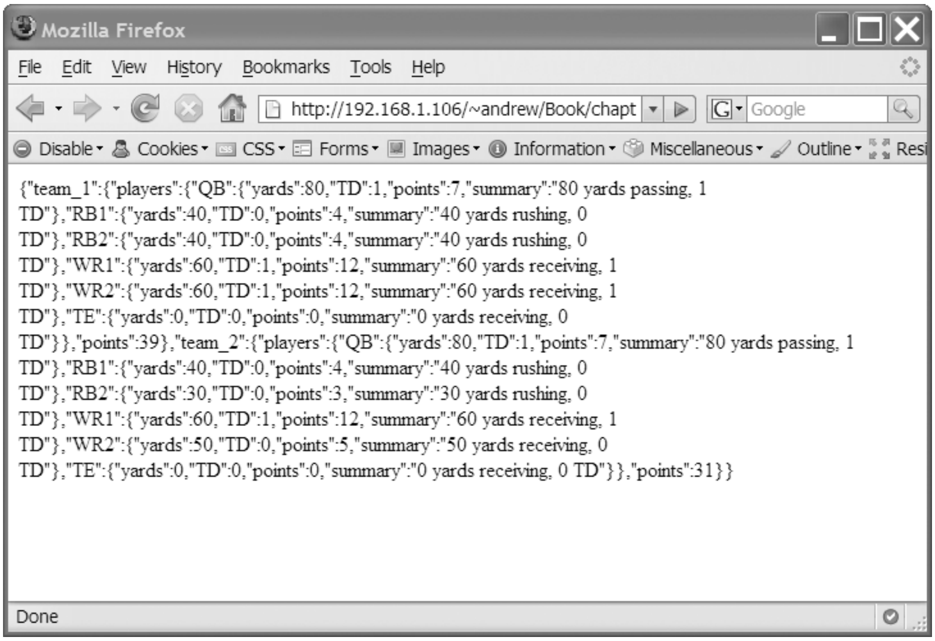
```
$qb  = new QB();
$rb1 = new RB1();
$rb2 = new RB2();
$wr1 = new WR1();
$wr2 = new WR2();
$te  = new TE();

$team1 = array();

// team 1 will score more points, so we give it
// the better "versions" of RB and WR
$team1["players"] = array(
  "QB"  => $qb->stats(),
  "RB1" => $rb1->stats(),
  "RB2" => $rb1->stats(),
  "WR1" => $wr1->stats(),
  "WR2" => $wr1->stats(),
  "TE"  => $te->stats()
);

// take the sum of all the players' scores
$team1["points"] = array_reduce($team1["players"], "score_sum");

$team2 = array();

// team 2 will score fewer points, so we give it
// both "versions" of RB and WR
$team2["players"] = array(
  "QB"  => $qb->stats(),
  "RB1" => $rb1->stats(),
  "RB2" => $rb2->stats(),
  "WR1" => $wr1->stats(),
  "WR2" => $wr2->stats(),
  "TE"  => $te->stats()
);

// take the sum of all the players' scores
$team2["score"] = array_reduce($team2["players"], "score_sum");

// deliver it in one large JSON chunk
echo json_encode(array("team_1" => $team1, "team_2" => $team2));
```

To paraphrase Blaise Pascal: I apologize for writing a long script, but I lack the time to write a short one. We could have taken the time to write more elegant code, but why? This script doesn't need to be maintainable; it just needs to work. And football season is fast approaching. Better to take extra care with the code that the *general public* will see.

Testing It Out

It will be easy to see whether our script works—we need only open it in a browser. Fire up Firefox and type the URL to your `scores.php` file.

If all goes well, you should see some JSON on your screen (see Figure 4-20).


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/Book/chapt
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resi
Done

```
{"team_1":{"players":{"QB":{"yards":80,"TD":1,"points":7,"summary":"80 yards passing, 1
TD"},"RB1":{"yards":40,"TD":0,"points":4,"summary":"40 yards rushing, 0
TD"},"RB2":{"yards":40,"TD":0,"points":4,"summary":"40 yards rushing, 0
TD"},"WR1":{"yards":60,"TD":1,"points":12,"summary":"60 yards receiving, 1
TD"},"WR2":{"yards":60,"TD":1,"points":12,"summary":"60 yards receiving, 1
TD"},"TE":{"yards":0,"TD":0,"points":0,"summary":"0 yards receiving, 0
TD"}},"points":39},"team_2":{"players":{"QB":{"yards":80,"TD":1,"points":7,"summary":"80 yards passing, 1
TD"},"RB1":{"yards":40,"TD":0,"points":4,"summary":"40 yards rushing, 0
TD"},"RB2":{"yards":30,"TD":0,"points":3,"summary":"30 yards rushing, 0
TD"},"WR1":{"yards":60,"TD":1,"points":12,"summary":"60 yards receiving, 1
TD"},"WR2":{"yards":50,"TD":0,"points":5,"summary":"50 yards receiving, 0
TD"},"TE":{"yards":0,"TD":0,"points":0,"summary":"0 yards receiving, 0 TD"}},"points":31}}
```

Figure 4-20. *The raw data generated by our script*

The numbers on your screen will vary from those in Figure 4-20. Because they run off a 10-minute cycle, the last digit of your system time (in minutes) is the factor—the closer it is to 0, the closer the scores will be to 0. Reload your page in 30 seconds and some of the scores will increment—and will continue to increment until that minute hand hits another multiple of 10, at which time the scores will all go back to 0.

We have spent a lot of time on `scores.php`, but it will save us much more time later on. We've just written a simulation of nearly all the data our site needs from the outside world.

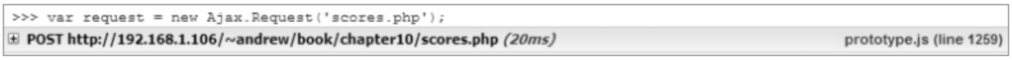
Making an Ajax Call

Finally, we come to the Ajax aspect of this example. Create a blank `index.html` file in the same directory as your `scores.php` file. It shouldn't be completely empty—make sure it loads `prototype.js`—but it doesn't need any content. From here we can use the Firebug shell to call our PHP script and look at the response.

Open `index.html` in a browser, and then open the Firebug console and type the following:

```
var request = new Ajax.Request("scores.php");
```

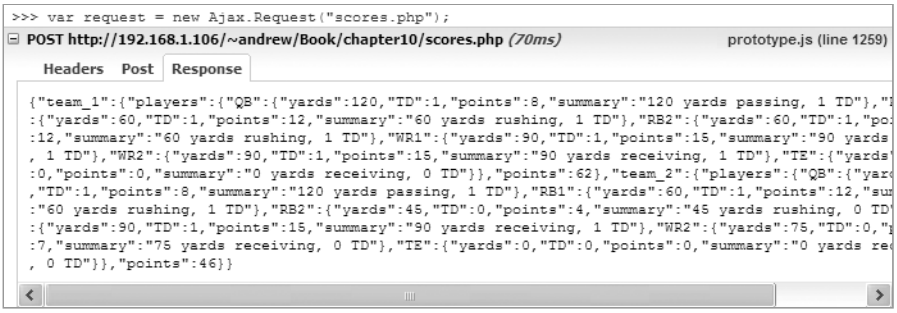
Firebug logs all the details about the Ajax request, as shown in Figure 4-21.



```
>>> var request = new Ajax.Request('scores.php');
⊞ POST http://192.168.1.106/~andrew/book/chapter10/scores.php (20ms)          prototype.js (line 1259)
```

Figure 4-21. *Our Ajax request in the Firebug console*

Expand this line, and then click the Response tab (see Figure 4-22).



```
>>> var request = new Ajax.Request("scores.php");
⊟ POST http://192.168.1.106/~andrew/Book/chapter10/scores.php (70ms)     prototype.js (line 1259)
```

```
{"team_1":{"players":{"QB":{"yards":120,"TD":1,"points":8,"summary":"120 yards passing, 1 TD"},"
:{"yards":60,"TD":1,"points":12,"summary":"60 yards rushing, 1 TD"},"RB2":{"yards":60,"TD":1,"po
:12,"summary":"60 yards rushing, 1 TD"},"WR1":{"yards":90,"TD":1,"points":15,"summary":"90 yards
, 1 TD"},"WR2":{"yards":90,"TD":1,"points":15,"summary":"90 yards receiving, 1 TD"},"TE":{"yards"
:0,"points":0,"summary":"0 yards receiving, 0 TD"}},"points":62},"team_2":{"players":{"QB":{"yar
,"TD":1,"points":8,"summary":"120 yards passing, 1 TD"},"RB1":{"yards":60,"TD":1,"points":12,"su
:"60 yards rushing, 1 TD"},"RB2":{"yards":45,"TD":0,"points":4,"summary":"45 yards rushing, 0 TD
:{"yards":90,"TD":1,"points":15,"summary":"90 yards receiving, 1 TD"},"WR2":{"yards":75,"TD":0,"
:7,"summary":"75 yards receiving, 0 TD"},"TE":{"yards":0,"TD":0,"points":0,"summary":"0 yards re
, 0 TD"}},"points":46}}
```

Figure 4-22. *The same data we saw in Figure 4-20*

There's our JSON, brackets and everything. Typing `request.responseText` into the Firebug console will give you the response in string form.

We can do better than that, though. Go back to the request details, and then switch to the Headers tab. There are two sets of headers—*request* headers and *response* headers—corresponding to the headers we sent out and the headers we got back, respectively. The response headers should tell you that our JSON data was served with a `Content-type` of `text/html`.

It's *not* HTML, though; PHP just serves up everything as HTML by default. We can tell our script to override this default. The de facto `Content-type` for JSON is `application/json`, so let's use that.

Go back to `scores.php` (last time, I promise) and insert the following bold line near the bottom:

```
// deliver it in one large JSON chunk
header("Content-type: application/json");
echo json_encode(array("team_1" => $team1, "team_2" => $team2));
```

This call to the `header` function will set the proper `Content-type` header for the response.

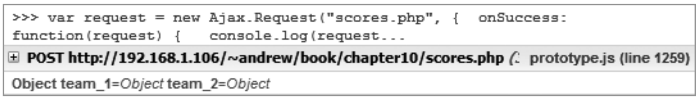
■ **Caution** You must call the `header` function before any output has been placed in the response. This includes anything printed or echoed, plus anything that occurs in your script before the PHP start tag (`<?php`). Even *line breaks* count as output.

Save your changes, and then go to the Firebug console. Press the up arrow key to recall the last statement you typed, and then press Enter. Inspect the details of this request and you'll notice that the `Content-type` has changed to `application/json`.

Why did we bother with this? It's not just a compulsion of mine; I promise. When Prototype's `Ajax.Request` sees the `application/json` content type, it knows what sort of response to expect. It unserializes the JSON string *automatically*, creating a new property on the response. To prove it, we'll try one more statement in the Firebug console. (You may want to switch to multiline mode for this one.)

```
var request = new Ajax.Request("scores.php", {
 onSuccess: function(request) {
  console.log(request.responseJSON);
 }
});
```

Run this statement; then watch a miracle happen in your console (see Figure 4-23).



```
>>> var request = new Ajax.Request("scores.php", {  onSuccess:
function(request) {   console.log(request...
⊞ POST http://192.168.1.106/~andrew/book/chapter10/scores.php (:  prototype.js (line 1259)
Object team_1=Object team_2=Object
```

Figure 4-23. *Our data. But it's no longer raw.*

Egad! That looks like our data. Click the bottom line to inspect the object—the JSON response has been converted to `Object` form automatically.

Let's recap what we've learned about the different Ajax response formats:

- All requests, no matter what the `Content-type` of the response, bear a `responseText` property that holds a string representation of the response.
- Requests that carry an XML `Content-type` bear a `responseXML` property that holds a DOM representation of the response.
- Prototype extends this pattern to JSON responses. Requests that carry a JSON `Content-type` bear a `responseJSON` property that holds an `Object` representation of the response.

The `responseJSON` property, while nonstandard, is the natural extension of an existing convention. It simplifies the very common pattern of transporting a data structure from server to client, converting the payload into the data type it's meant to be.

Summary

The code you've written in this chapter demonstrates the flexible design of Prototype's Ajax classes—simple on the surface, but robust on the inside. As the examples went from simple to complex, the amount of code you wrote increased in modest proportion.

You typed all your code into Firebug because you're just starting out—as you learn about other aspects of Prototype, we'll mix them in with what you already know, thus pushing the examples closer and closer to real-world situations. The next chapter, all about events, gives us a big push in that direction.

CHAPTER 5

Events

If you're a fan of the absurd, bizarre, and overcomplicated—and you must be if you write JavaScript—you're probably familiar with Rube Goldberg machines. Named for their cartoonist creator, these ad hoc, convoluted contraptions humorously obfuscate simple processes like flipping a light switch or opening a door. They paint a picture of a parallel universe where simple tasks are complicated.

I'm drawn to Goldberg machines because they're the opposite of how things work in everyday life. Usually we dream of an ideal world where things that *seem* simple actually *are* simple, without reality stubbornly standing in the way.

Browser-based, event-driven interfaces *should* be simple. *Highlight an element when the user clicks on it. Disable a submit button after a form has been submitted.* But reality is harsh in the world of web development. Writing event handlers can feel like building an elaborate set of pulleys to open a window.

In an unyielding quest to help client-side developers manage this complexity, Prototype sports a robust event system—one that makes simple things simple and complex things possible.

State of the Browser (Or, How We Got Here)

I feel like a broken record—browser wars, lack of specifications, Netscape did one thing, Internet Explorer did another. It bears repeating because we're still feeling the effects 10 years later.

Rather than subject you to a bland history lesson, though, I'll recap through code.

Pre-DOM, Part 1

Let's travel back in time to 1996—the heyday of Netscape 2.0, the first version to implement JavaScript. Back in the day, this was how you assigned events:

```
<input type="submit" value="Post" onclick="postBreakfastLogEntry();">
```

The event assignment was just another attribute in the HTML. On its face, this is a simple and straightforward way to assign events: it makes simple things simple. Unfortunately, it also makes complex things damn near impossible.

First: note that we're inside a pair of quotation marks. What if we need to use quotation marks in our code?

```
<input type="submit" value="Post"
 onclick="postBreakfastLogEntryWithStatus(\"draft\");">
```

We could use single quotes instead, but that's just a band-aid. With sufficiently complex code, we'd be forced to escape quotation marks. By itself it's not a big deal—just an illustration of how HTML and JavaScript don't mix well.

Second, what if we need to assign the event to a bunch of different things?

```
  <input type="submit" value="Save as Draft"
    onclick="postBreakfastLogEntryWithStatus('draft');">
  <input type="submit" value="Save and Publish"
    onclick="postBreakfastLogEntryWithStatus('published');">
  <input type="submit" value="Discard"
    onclick="postBreakfastLogEntryWithStatus('discarded');">
```

That's a lot of typing—and a lot of code duplication for something that should be much easier. DOM scripting makes it trivial to work with arbitrary collections of elements. Why, then, are we writing copy-and-paste HTML to solve this problem?

Pre-DOM, Part 2

Such scary times. Let's jump ahead one year and assign events the way Netscape 3.0 lets us: in pure JavaScript.

```
// HTML:
<input type="submit" value="Save and Publish" id="save_and_publish">

// JavaScript:
var submitButton = document.getElementById('save_and_publish');
submitButton.onclick = postBreakfastLogEntry;
```

Here we're doing what the previous example only hinted at. Event handlers (`onclick`, `onmouseover`, `onfocus`, etc.) are treated as properties of the node itself. We can assign a function to this property—passing it a reference to a named function *or* declaring an anonymous function on the spot.

Now it's much more elegant to assign the same handler to a bunch of elements:

```
$('save_and_publish', 'save_as_draft', 'discard').each( function(button) {
  button.onclick = postBreakfastLogEntry;
});
```

But if we assign one function to a bunch of different elements, how do we figure out which one received the event?

In this example, our `postBreakfastLogEntry` function should receive an *event object* as its first argument—one that will report useful information about that event's context. Just as you can inspect a letter and know its post office of origin, an event handler is able to inspect an event and know what type it is, where it came from, and what should be done with it.

```
function postBreakfastLogEntry(event) {
  var element = event.target;
  if (element.id === 'save_and_publish')
    saveAndPublish();
  /* ...et cetera */
}
```

Unfortunately, events have never been quite this simple. This is a portrayal of an ideal simplicity—not on Earth, but in a parallel universe where the adversarial and fast-moving browser market didn't make simplicity impossible. The real-world example would look like this:

```
function postBreakfastLogEntry(event) {
  event = event || window.event;
  var element = event.target || event.srcElement;
  if (element.id === 'save_and_publish')
    saveAndPublish();
  /* ...et cetera */
}
```

The browser wars were in full swing by 1997. Hasty to add differentiating features to their own products, and working in an area where a standards body had not yet claimed authority, Internet Explorer and Netscape developed event models that were alike enough to *seem* compatible, but still different enough to cause maximum confusion and headaches.

Ten years later, the landscape is not all that different. But instead of Internet Explorer versus Netscape, it's Internet Explorer versus the standards. The other three major browsers have all adopted DOM Level 2 Events, the 2000 specification that finally brought some authority to the discussion, but as of version 7, Internet Explorer still uses its proprietary event model.

The strides web applications have made in the last decade only call more attention to this problem: writing cross-browser event code is just as hard now as it was in 1997. Having to reconcile these differences—in property names, in event assignment, and in the event types themselves—is the mental mildew that makes simple things hard.

Events: The Crash Course

Let's go back to our site from the previous chapter. So far, we've neglected the UI (i.e., there is none). Unless we want our breakfast loggers to type all their commands into the Firebug console, we'll have to build a simple form for adding entries to the log.

In between chapters, I took the liberty of writing some CSS to make our page a *little* less ugly (see Figure 5-1).

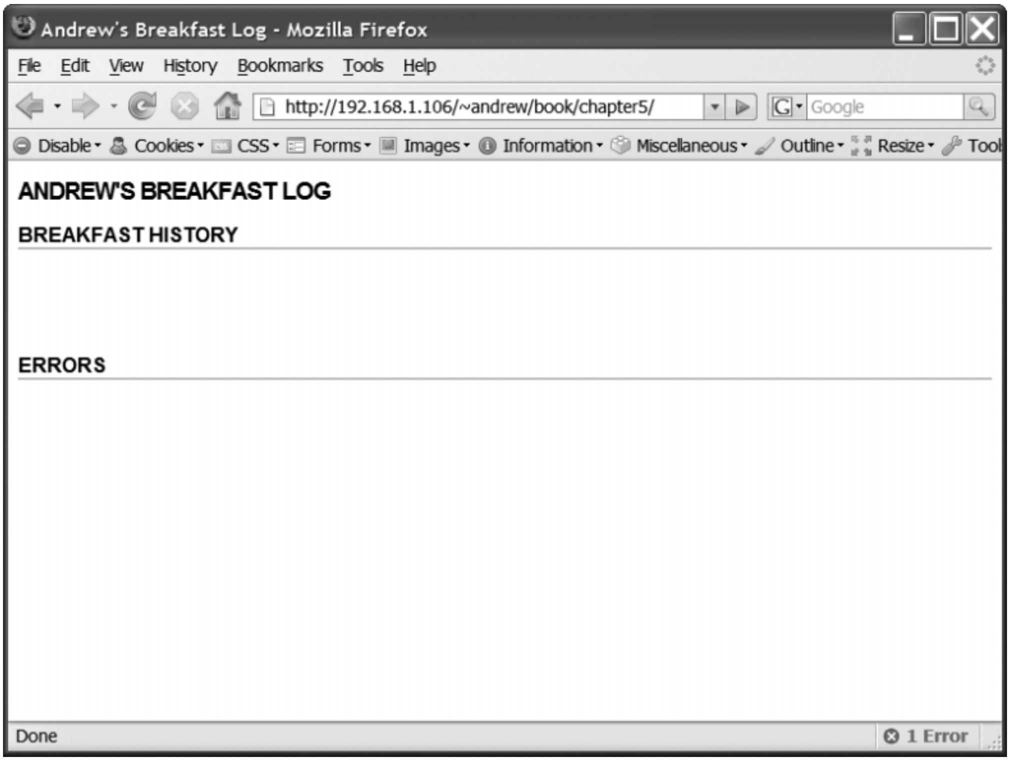

Andrew's Breakfast Log - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/book/chapter5/
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tool
ANDREW'S BREAKFAST LOG
BREAKFAST HISTORY
ERRORS
Done
1 Error

Figure 5-1. *Lipstick on the pig*

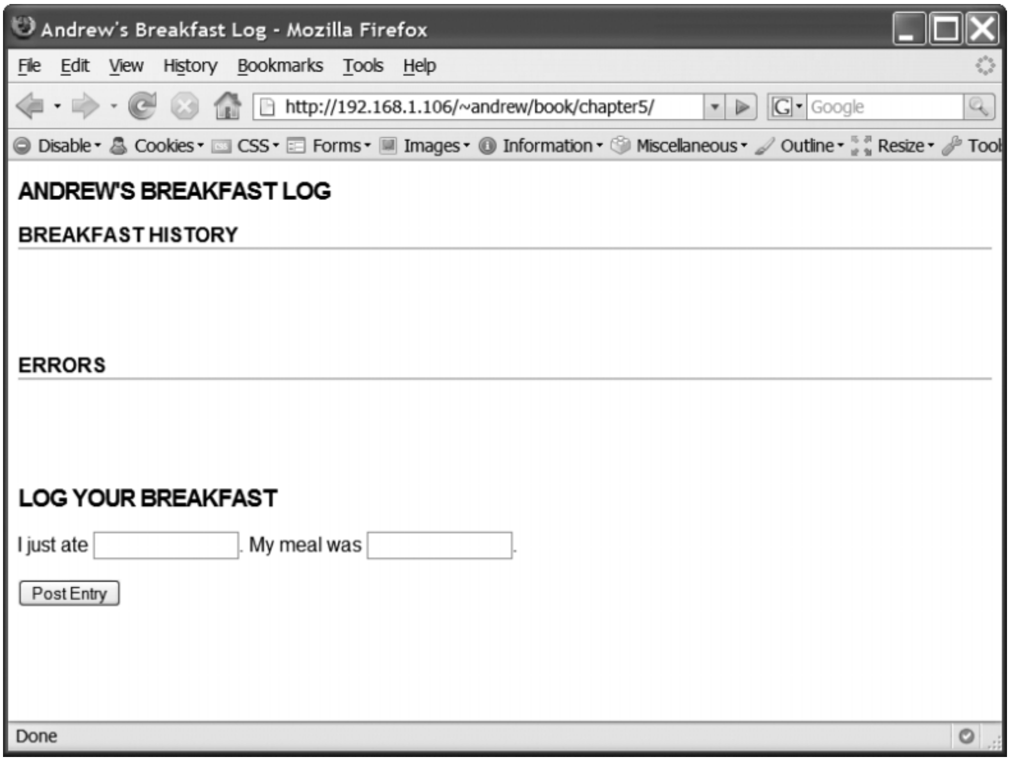
Open up `index.html` and follow along. (Feel free to style your version in whatever manner you choose.)

We're collecting two pieces of information: what the user ate *and* how good it was. The form practically writes itself.

```
    <h1>Log Your Breakfast</h1>
    <form id="entry" method="post" action="breakfast.php">
      <p>
        I just ate <input type="text" id="food_type" name="food_type"
         size="15" />.
 My meal was <input type="text" id="taste" name="taste" size="15" />.</p>
      <input type="submit" name="submit" value="Post Entry" />
    </form>
```

The hardest part has already been done. Remember that we already have a `breakfast.php` script, one that expects a form submission containing these two fields.

Insert this markup at the bottom of `index.html`, and your page should look something like Figure 5-2.


Andrew's Breakfast Log - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/book/chapter5/
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tool
Post Entry
Done

ANDREW'S BREAKFAST LOG

BREAKFAST HISTORY

ERRORS

LOG YOUR BREAKFAST

I just ate . My meal was .

Figure 5-2. *Our form doesn't do anything yet, but it looks nice.*

Now let's start writing some JavaScript! First we're going to create a new file called `breakfast.js` and include it from `index.html`. Separating HTML from JavaScript, putting each in its own file, will stave off the urge to write spaghetti code.

File: index.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <title>Andrew's Breakfast Log</title>
    <link rel="stylesheet" href="breakfast.css" type="text/css" />

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

Because we'll be writing code that uses parts of Prototype, we must include our new script at the *end*. (Remember, Prototype should be the first script you include on your page.)

There's nothing in `breakfast.js` yet, so let's fix that. We need to write the function that will get called when the form is submitted. Then we'll write the glue to connect it to the actual event.

```
function submitEntryForm() {
  var updater = new Ajax.Updater({
    success: 'breakfast_history', failure: 'error_log'
  }, 'breakfast.php',
  { parameters: { food_type: $('food_type').value, taste: $('taste').value } });
}
```

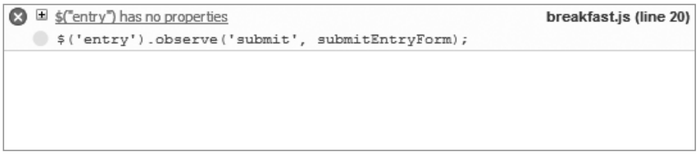
This code is almost identical to the code we wrote in the last chapter. Only one thing has changed: instead of specifying the values directly, we look up the values of the two text boxes. There are many ways to hook into these values, but an ID lookup is the quickest.

Now the glue. It won't take much code to connect the function and the event—we can use Prototype's `observe` method:

```
$('entry').observe('submit', submitEntryForm);
```

The first argument indicates what we're listening for—we want to run this code when our form is submitted. The second argument is our responder—the name of the function that will get called when the form's `submit` event fires.

Add the `submitEntryForm` function and the `observe` call to `breakfast.js`. Save, go back to your browser, reload the page, and . . . what? Error? (See Figure 5-3.)



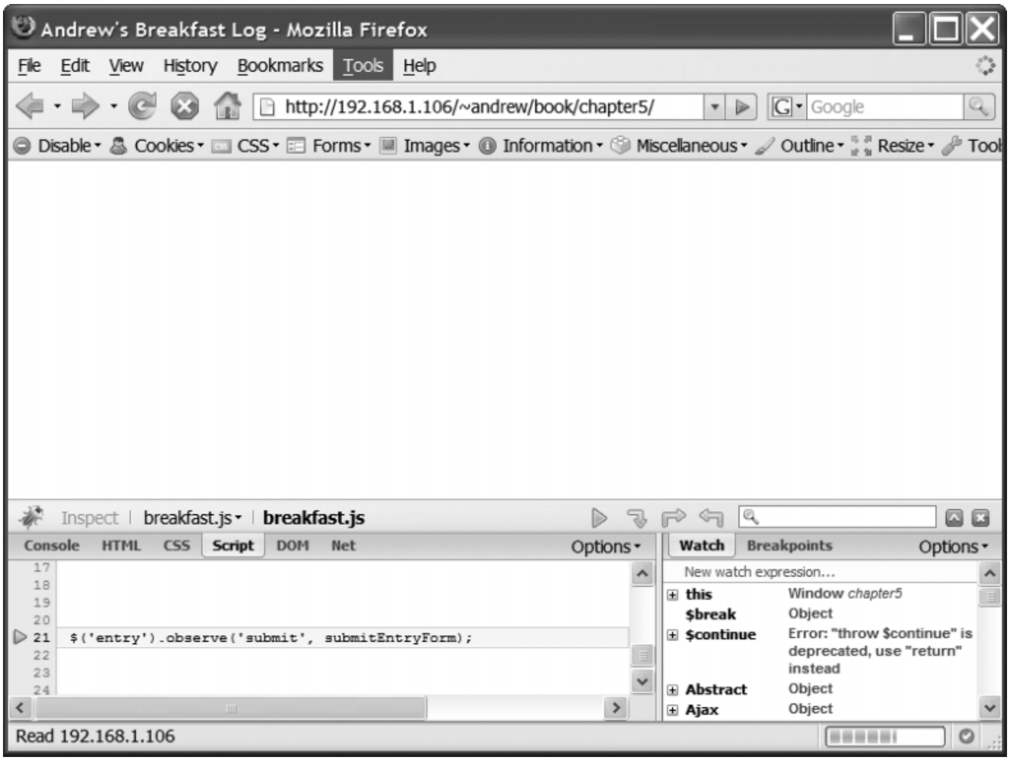
```
$("entry") has no properties                    breakfast.js (line 20)
$('entry').observe('submit', submitEntryForm);
```

Figure 5-3. *Guh?*

Of course it's defined! It's right there on the page! I'm staring straight at it!

Firebug can tell us what went wrong. Select the Script tab, click Options, and then click Break on All Errors. This way you'll know exactly when that error happens.

Reload the page. Almost immediately the page load will halt, as Firebug points you to the offending line (see Figure 5-4).


Andrew's Breakfast Log - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/book/chapter5/
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tool
Inspect breakfast.js breakfast.js
Console HTML CSS Script DOM Net Options
Watch Breakpoints Options
New watch expression...
Read 192.168.1.106

```
$('entry').observe('submit', submitEntryForm);
```

```
this        Window chapter5
$break      Object
$continue   Error: "throw $continue" is
            deprecated, use "return"
            instead
Abstract    Object
Ajax        Object
```

Figure 5-4. *When Break on All Errors is on, any error in your code is treated as a debugger breakpoint.*

The Script tab is Firebug's JavaScript debugger. We've just set a *breakpoint*, pausing the evaluation of scripts (and rendering in general) at a certain spot. From here, we can resume the page load, step through functions one by one, and even use Firebug's console.

But right now we don't need to do any of that—the screen tells us all we need to know. Notice how the viewport is empty. None of our content is there. At the time we tried to set the event, the element we were referencing *hadn't yet been created*.

This is an easy trap to fall into. Script tags are typically placed in the head of an HTML document. In the common case where a script needs to modify stuff in the *body* of a document, it's got to wait.

OK, new plan—we'll add our listeners *when the document is fully loaded*, so that we can be sure that the entire DOM tree is at our disposal. Take the offending line of code and place it inside a function:

```
function addObservers() {
  $('entry').observe('submit', submitEntryForm);
}
```

Now we can set this function to run when the page loads using the `load` event:

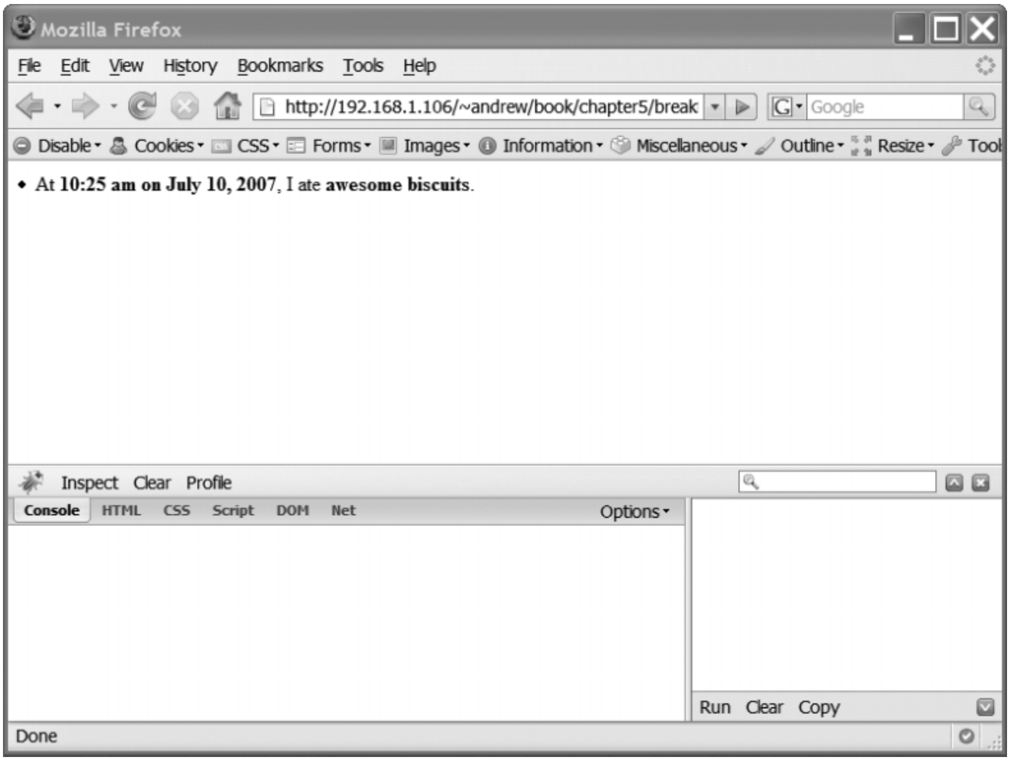
```
Event.observe(window, 'load', addObservers);
```

Make these changes to `breakfast.js`, and then reload the page. Our error is gone—and, more to the point, the Ajax form submission works this time! Wait, no. Never mind. Something else is wrong (see Figure 5-5).

What could be causing this? The only thing on the page is the HTML fragment that should've been injected into our *other* page.

Look at the address bar. When we submitted the form, the browser went to `breakfast.php`, the URL in the form's `action` attribute. Following that URL is the `submit` event's *default action*.

That means we're at fault again. When we submitted the form, `submitEntryForm` was called as we intended. But we didn't *hijack* the submit event; we just listened for it. If we want to suppress this default action, we must explicitly say so.


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/book/chapter5/break
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tool
Inspect Clear Profile
Console HTML CSS Script DOM Net Options
Run Clear Copy
Done

- **At 10:25 am on July 10, 2007**, I ate **awesome biscuits**.

Figure 5-5. *This is the HTML fragment we wanted, but it's on its own page.*

Using Event#stopPropagation, Event#preventDefault, and Event#stop

To pull this off, we're borrowing a couple of methods from the DOM2 Events spec. Internet Explorer doesn't support these events natively, but we can fake it on the fly—augmenting Internet Explorer's event object with instance methods the same way we augment DOM nodes with instance methods.

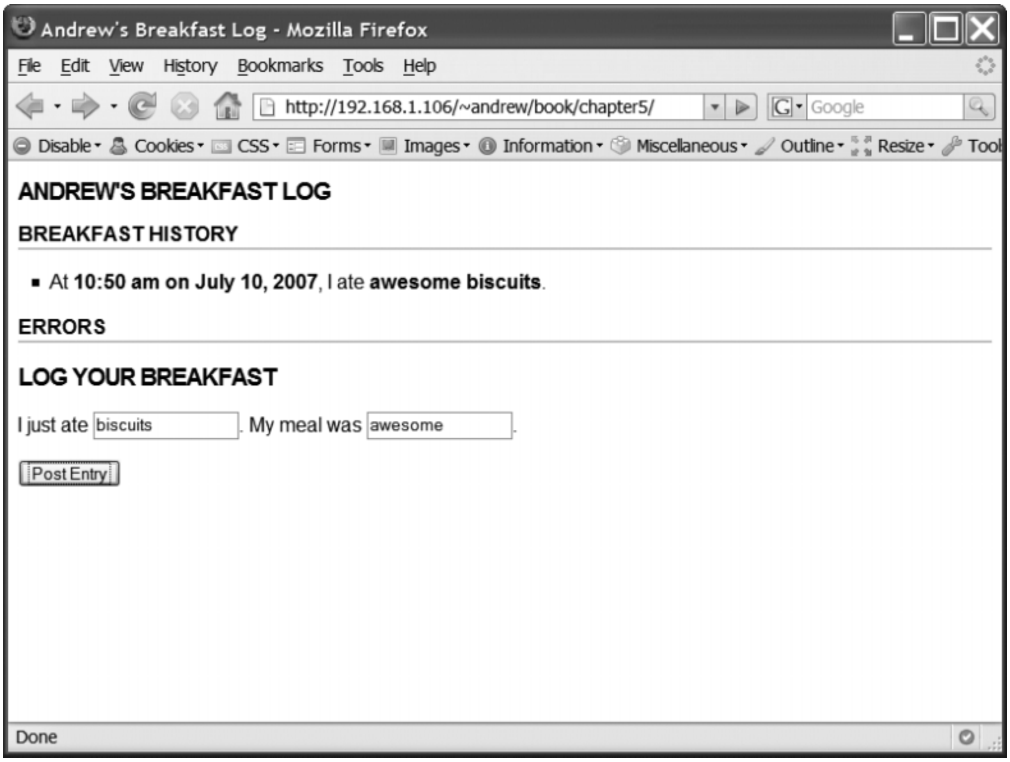
First, we add an `event` argument to our handler so that we can use the event object. (We could have done this from the start, but we didn't have a use for it until just now.) Then, at the end of the handler, we tell the event not to do what it had originally planned.

```
function submitEntryForm(event) {
  var updater = new Ajax.Updater({
    success: 'breakfast_history', failure: 'error_log'
  }, 'breakfast.php',
  { parameters: { food_type: $('food_type').value, taste: $('taste').value } });
  event.preventDefault();
}
```

Prototype gives you two other methods to control the flow of events:

- Normally, events start deep in the DOM tree and "bubble" up to the top (e.g., clicking a table cell will also fire an event in that cell's table row, in the table body, in the table, in the table's parent node, and so on all the way up to `window`). But you can halt the bubbling phase using the `stopPropagation` method.
- When you need to stop the event from bubbling *and* prevent the default action, use Prototype's custom `stop` method. It's a shortcut for calling both `stopPropagation` and `preventDefault`.

OK, let's try one more time. Reload `index.html` and try to submit a breakfast log (see Figure 5-6).


Andrew's Breakfast Log - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/book/chapter5/
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tool
Done

ANDREW'S BREAKFAST LOG

BREAKFAST HISTORY

- At **10:50 am on July 10, 2007**, I ate **awesome biscuits**.

ERRORS

LOG YOUR BREAKFAST

I just ate biscuits. My meal was awesome.

Post Entry

Figure 5-6. *Finally, we can submit meal information without having to reload the page! Eureka!*

That was easy, right? Right?

Be aware: The behavior layer of web development (JavaScript) is far more complex than the structural layer (HTML) or the presentational layer (CSS). Ordinary web pages are snapshots—the server sends it, the browser renders it, and it's done. Pages that make use of JavaScript, however, have some aspect of mutability to them. The page may be loaded, but it's never *done*.

You *will* run into problems like the ones we encountered in this example. You *will* make mistakes simply because all this may be new and unfamiliar. Don't get discouraged! Rely on your tools—Firebug, Microsoft Script Debugger, and even the trusty `alert` dialog—to get you out of the quagmire.

A Further Example

We'll keep coming back to events in subsequent chapters, since they're a part of everything you do in DOM scripting. But let's add just one more thing to our page.

Being able to post entries without leaving the page is quite handy, but what if you're just there to read your old entries? In the interest of removing clutter, let's hide the form by default, showing it only if the user asks for it.

Let's assign an `id` to the Log Your Breakfast heading so that we can grab it easily. Let's also write some CSS to make it feel more button-like and invite clicking.

```
// HTML:
<h2 id="toggler">Log Your Breakfast &darr;</h2>

// CSS:
#toggler {
  cursor: pointer;
  border: 2px solid #222;
  background-color: #eee;
}
```

We also want the form to be hidden when the page first appears, so let's add a handler that will hide the form when the page loads:

```
Event.observe(window, "load", function() { $('entry').hide(); });
```

And the last ingredient is a handler for the new link's `click` event:

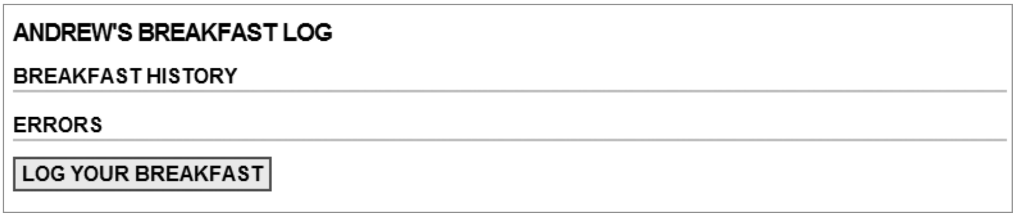
```
function toggleEntryForm(event) {
  $('entry').toggle();
  event.preventDefault();
}
```

The `toggle` method conveniently alternates an element between hidden and shown. (In other words, it will show hidden elements and hide shown elements.) Note the use of `preventDefault`—since we don't want the browser to *follow* the link, we've got to suppress the default action.

We can assign this event just like we assigned the other one—with our `addObservers` function:

```
function addObservers() {
  $('entry').observe('submit', submitEntryForm);
  $('toggler').observe('click', toggleEntryForm);
}
```

Now *two* events will be assigned on page load. Save `breakfast.js`, reload `index.html`, and marvel that this exercise was much easier than the last (see Figure 5-7).



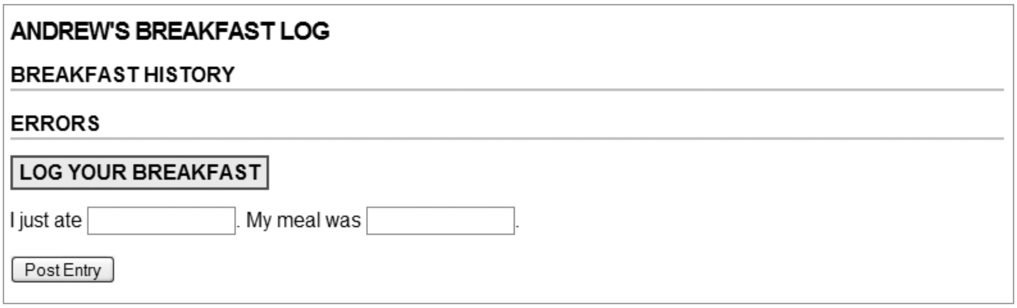


Figure 5-7. *Each click of the link will toggle the display state of the form.*

Events and Forms

A whole group of events is devoted to the user's interaction with form elements. These can be tricky to manage, but they also stand to gain the most from UI enhancements.

Client-Side Validation

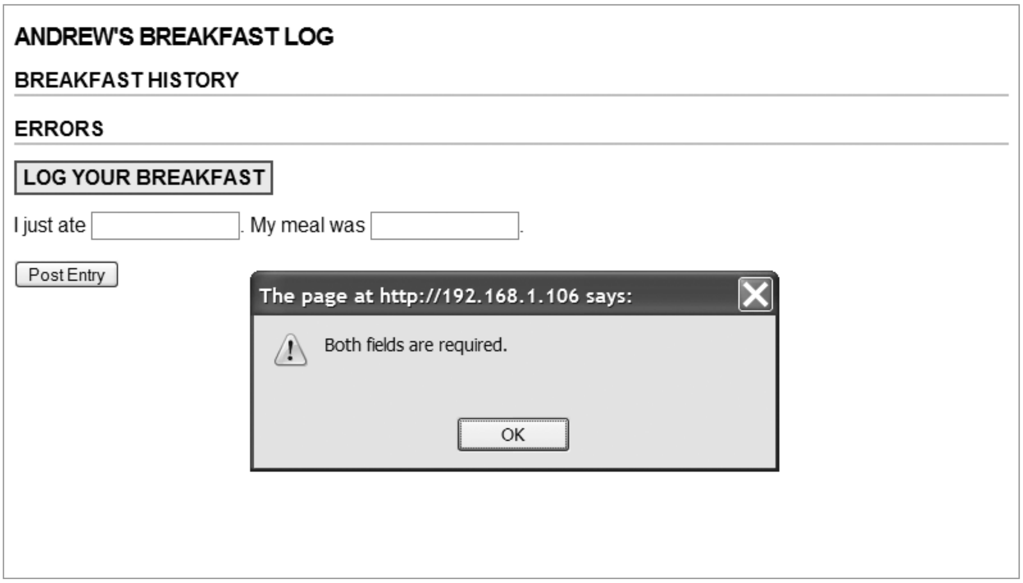
In Chapter 4, we wrote some PHP to check the submitted values on the server side. If the user had left either field blank, the submission would have been invalid, and the server would have sent back an error HTTP status code.

We don’t need a server round-trip here, though. We can just as easily check whether a field is blank on the client side. We want to code defensively, catching possible user mistakes as early as possible.

We can perform this check when the form is submitted; we’ve already got a function handling that event.

```
function submitEntryForm(event) {
  event.preventDefault();
  if ($('food_type').value === '' || $('taste').value === '') {
    alert('Both fields are required.');
    return;
  }
  var updater = new Ajax.Updater(
    { success: 'breakfast_history', failure: 'error_log' },
    'breakfast.php',
    { parameters: { food_type: $('food_type').value, taste: $('taste').value } }
  );
}
```

Our handler now branches. If the two fields we’re looking at are empty, we show a message and stop; if not, we submit the form via Ajax. Either way, we want to stop the default action of the form, so we move the `event.preventDefault` call to the top of the function (see Figure 5-8).



Both fields are required.

Figure 5-8. *The validation message we expected*

This works just like we expected. But let's try something a bit subtler. Imagine that each of the text boxes has a state that's either *valid* or *invalid*. At any point in time, it's either one or the other. Both text boxes need to be valid before the form is submitted.

Let's write a CSS rule for an invalid text box:

```
input#food_type.invalid,
input#taste.invalid {
  border: 2px solid #900;
}
```

So that we aren't *too* subtle, an invalid text box will be shown with a thick red border.

When the page loads, both text boxes are empty, but neither one can be called *invalid* yet, because the user hasn't had a chance to enter text. But we know it's *definitely* invalid if the text box receives focus, then loses focus, and is *still* empty. That's when we should alert the user.

There's an event for losing focus—it's called `blur`. So let's use it.

```
function onTextBoxBlur(event) {
  var textBox = event.element();
  if (textBox.value.length === 0) textBox.addClassName('invalid');
  else textBox.removeClassName('invalid');
}
```

We use Prototype's `Event#element` method to figure out which element received the event. The method ensures that we get an element node as the target, not a text node.

When a text field is blurred, we make sure it has a nonempty value. If so, we add the class name to mark it as invalid. If not, the field is valid, so we remove the class name.

We're going to leave our `submit` handler the way it is, since the `blur` handler won't catch everything. But let's change it to use the new approach.

```
function submitEntryForm(event) {
  event.preventDefault();
  var valid = true;
  $('food_type', 'taste').each(function(box) {
    if (box.value.length === 0) {
      box.addClassName('invalid');
      valid = false;
    } else box.removeClassName('invalid');
  });
  if (!valid) {
    alert('Both fields are required.');
    return;
  }
  var updater = new Ajax.Updater( { success: 'breakfast_history',
   failure: 'error_log' }, 'breakfast.php',
   { parameters: { food_type: $('food_type').value, taste: $('taste').value } });
}
```

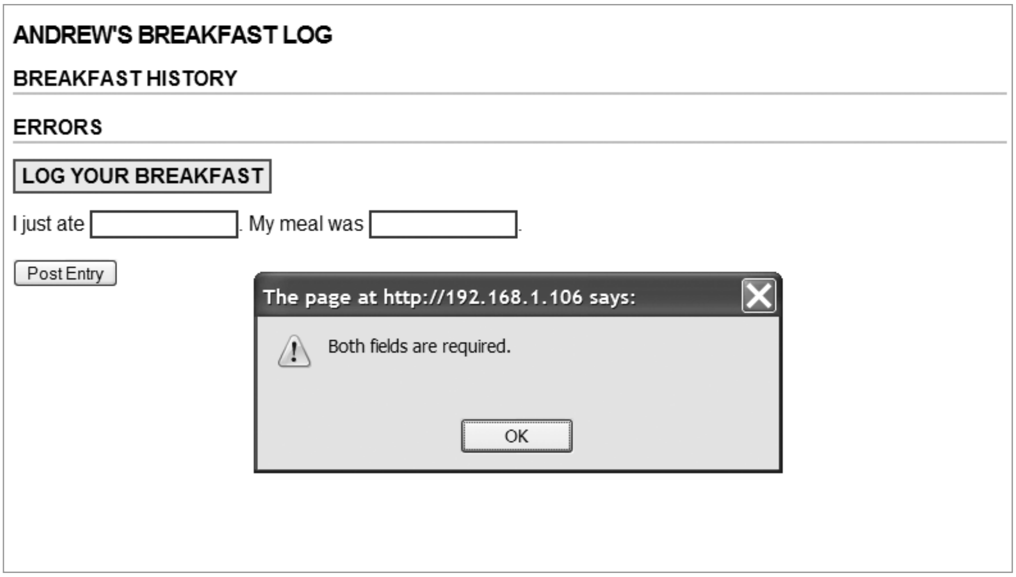
Don't forget to attach the new handler. Add it to the `addObservers` function.

```
function addObservers() {
  $('entry').observe('submit', submitEntryForm);
  $('toggler').observe('click', toggleEntryForm);
  $('food_type', 'taste').invoke('observe', 'blur', onTextBoxBlur);
}
```

Remember `Enumerable#invoke`? Here we're using it to call the `observe` method on a collection of two elements, passing the last two arguments in the process.

Now let's do some functional testing to make sure this works right. Reload the page in your browser.

First, try clicking the submit button immediately, before focus has been given to either text box. Figure 5-9 shows the result.


ANDREW'S BREAKFAST LOG

BREAKFAST HISTORY

ERRORS

LOG YOUR BREAKFAST

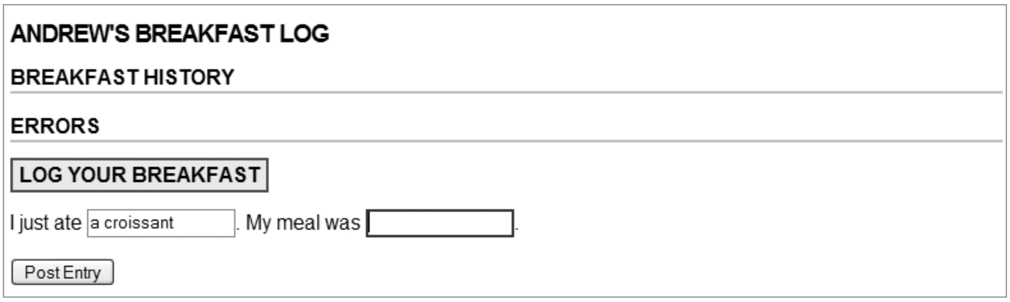
I just ate ______. My meal was ______.

The page at http://192.168.1.106 says:

Both fields are required.

Figure 5-9. *Both text boxes are invalid.*

Works as expected! Now click inside the `food_type` text box and type some text. Press Tab to change focus, and notice how the red border disappears (see Figure 5-10).



ANDREW'S BREAKFAST LOG

BREAKFAST HISTORY

ERRORS

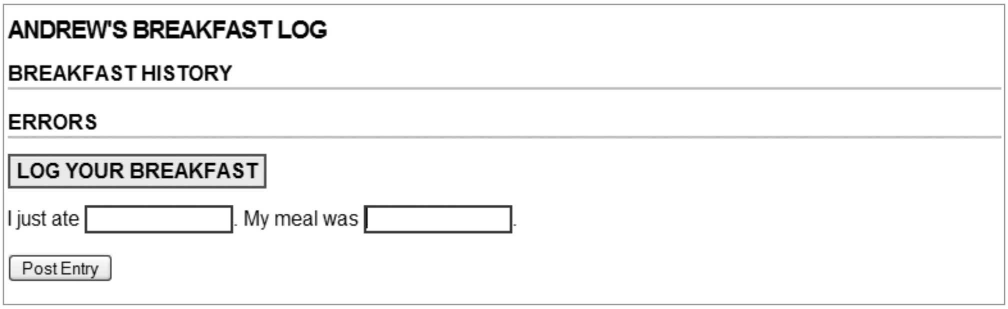
LOG YOUR BREAKFAST

I just ate a croissant. My meal was .

Post Entry

Figure 5-10. *Only the second text box is invalid.*

Now click inside the `food_type` text box once again to give it focus. Delete the box's value, and then press Tab. The text box should once again have a red outline, as shown in Figure 5-11.



ANDREW'S BREAKFAST LOG

BREAKFAST HISTORY

ERRORS

LOG YOUR BREAKFAST

I just ate . My meal was .

Post Entry

Figure 5-11. *Both text boxes are once again invalid.*

You might consider all this redundant, since the server validates the data itself, but keep in mind that the client-side and server-side validations serve different purposes. The server is validating the data so that it can let the client know if the request was successful; this is useful no matter *how* the breakfast log entry gets posted. The client is validating the data so that it can present a helpful and humane UI. The two are not at odds with one another.

Also, remember that *client-side validation is not a replacement for server-side validation*. Ideally, you'd do both, but validating data on the server is *essential*. You're in control of the server; you're not in control of the client.

Cleaning It Up

We could leave it like this, but if you're code-compulsive like I am, you've probably noticed several redundant lines of code. The check we're doing inside `submitEntryForm` is nearly identical to the one we're doing inside `onTextBoxBlur`. By changing the way we observe the form, we can easily combine these checks into one.

First, we'll write a function that checks *one* text box for validity:

```
function validateTextBox(textBox) {
  if (textBox.value.length === 0) {
    textBox.addClassName('invalid');
    return false;
  } else {
    textBox.removeClassName('invalid');
    return true;
  }
}
```

We'll use the return value of this function to indicate whether the text box in question is valid or invalid—`true` is valid; `false` is invalid.

Now we'll write a function that checks the *entire form* for empty text boxes:

```
function validateForm(form) {
  var textBoxes = Form.getInputs(form, 'text');
  return textBoxes.all(validateTextBox);
}
```

The highlighted part introduces a new method: `Form.getInputs`. It accepts a `form` element as the first parameter and returns all of the `input` elements contained within. The second argument is optional; if present, it will return only `inputs` of the given type. Here we want the form's text boxes, so the second argument is `"text"`.

`Form.getInputs` is also available as an instance method on `form` elements (e.g., `form.getInputs('text');`).

The second line gets a little tricky. Instead of using `Array#each` to iterate over the text boxes, we're using `Array#all`. Since `validateTextBox` returns a Boolean, we can look at the return values to figure out whether all text boxes are valid. If so, the statement (and thus the `validateForm` function) returns `true`; if not, `false`.

So, this function doesn't just mark our text boxes; it also returns a "good data/bad data" Boolean value. If the function returns `false`, we'll know not to submit the form.

Now we can simplify the code in `submitEntryForm`:

```
function submitEntryForm(event) {
  event.preventDefault();
  if (!validateForm('entry')) return;
  var updater = new Ajax.Updater(
   { success: 'breakfast_history', failure: 'error_log' },
    'breakfast.php',
    { parameters: { food_type: $('food_type').value, taste: $('taste').value } }
  );
}
```

Our new code does the same task more succinctly. If `validateForm` returns `false`, we bail on the form submission so that the user can correct the errors (which have been helpfully outlined in red). Otherwise we proceed as planned.

As a last step, we can rewrite the `onTextBoxBlur` function and save a couple lines of code:

```
function onTextBoxBlur(event) {
  return validateTextBox(event.target);
}
```

We've done more than clean up our code in this section; we've also eliminated redundancy. Furthermore, the new code will continue to work even if we add extra text boxes to the form later on. Make your code as flexible as possible—it will save time in the long run.

Custom Events

I've saved the best part for last. Native browser events are purely reactive; they're triggered by user action. Wouldn't it be great if we could harness the event model and use it to make our own events?

Let's go back to our fantasy football example. Imagine being able to trigger an "event" in your code whenever the user changes his lineup, or whenever the lead changes in a game. You'd also be able to write code that *listens* for those kinds of events and calls handlers accordingly.

If I were to get academic on you, I'd call this an *event-driven architecture*. I could also call it a publish/subscribe model (or pub/sub for short). No matter what I call it, the key idea is the *decoupling* of publisher and subscriber. The code that *responds* to these kinds of events doesn't need to know *how* the event was triggered—the object sent along with the event will contain all the necessary information.

I wouldn't be telling you any of this, naturally, if it were a pipe dream. Prototype introduced support for custom events in version 1.6. Using Prototype, you can fire custom events from anywhere in your code; you can also listen for custom events with the *same API* that you'd use to listen for native browser events.

The First Custom Event

Prototype itself fires a custom event called `dom:loaded`. It fires at a specific time in the page's life cycle: *after* the page's DOM tree is fully accessible to scripts, but *before* the window's `load` event, which doesn't fire until all external assets (e.g., images) have been fully downloaded.

Use `dom:loaded` when you want to work with the DOM in that narrow window of time before the page appears on the screen fully rendered. In nearly all cases, it's better to assign to `dom:loaded` than `load`—unless your handler depends upon having everything downloaded and rendered.

This is also a good time to talk about the naming scheme for custom events. You've probably noticed that `dom:loaded`, unlike native events, contains a colon. This is by design—*all custom events must contain a colon in their names.* Since custom events are handled differently under the hood, Prototype needs a way to distinguish them from native browser events (which number in the *hundreds* if all major browsers are considered). Embrace the convention.

Broadcasting Scores

The data stream we built in Chapter 4 will power a large portion of our fantasy football site. It would be wasteful and silly for each JavaScript component to make its own Ajax requests, so let's write some general code with the specific task of "asking" for scores from the server, and then "announcing" these scores through some sort of public address system.

Create a new file called `score_broadcaster.js` and place this code inside:

```
var ScoreBroadcaster = {
  setup: function() {
    this.executer = new PeriodicalExecuter(this.update.bind(this), 30);
    this.update();
  },

  update: function() {
    this.request = new Ajax.Request("scores.php", {
      onSuccess: this.success.bind(this)
    });
  },
  success: function(request) {
    document.fire("score:updated", request.responseJSON);
  }
};

document.observe("dom:loaded", function() {
  ScoreBroadcaster.setup();
});
```

First, notice the design pattern—we're creating a `ScoreBroadcaster` object to act as our namespace. Next, jump to the bottom—we've hooked up `ScoreBroadcaster.setup` to run as soon as the DOM is ready. This function schedules a new Ajax request every 30 seconds; successful requests will call another function that will fire a custom event with our data.

Now look in the middle—we call `document.fire` with two arguments. This method fires custom events, naturally, and exists on all elements (`Element#fire`) and on the `document` object, too. You've just learned two things about this method:

- The first argument is the name of the event to be fired. As we discussed, the name needs to have a colon in it, so let's call it `score:updated`. The *noun:verbed* naming scheme is just a convention, but it's a useful one.

- The second argument is an object that contains any custom properties for attaching to the event object. Just like native browser events, custom events pass an event object as the first argument to any handler. Alongside familiar properties like `target`, custom events have a `memo` property on their event objects. The second argument of `Element#fire` gets assigned to this property. In short, we're attaching the score information so that handlers can read it.

As we covered in Chapter 4, we're using Prototype's special `responseJSON` property on the Ajax response object—useful because it automatically unserializes the JSON payload. Using the `application/json` MIME type gets us this property for free.

That's one fewer link in the chain we have to worry about. When we write components, we won't have to deal with the boring details of getting the data. Score updates will be dropped upon them as though they were manna from heaven.

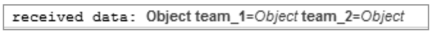
Listening for Scores

To illustrate this point, let's write some quick test code to make sure the custom event is working right. Add this to the bottom of `score_broadcaster.js`:

```
document.observe("dom:loaded", function() {
  document.observe("score:updated", function(event) {
    console.log("received data: ", event.memo);
  });
});
```

We listen for a custom event the same way we listen to a native event: using `Event.observe`. Custom events behave much the same way as native events: they bubble up the DOM tree, and they have their own event objects that implement all the properties and methods we've already covered.

Here we listen for our custom `score:updated` event and log to the Firebug console whenever it fires. Include this script on an HTML page and observe the result. Every 30 seconds, one of the lines shown in Figure 5-12 should appear in your Firebug console.



```
received data:  Object team_1=Object team_2=Object
```

Figure 5-12. *This line should appear in your Firebug console twice a minute.*

In subsequent chapters, we'll write code that hooks these stats up to the interface.

Summary

To revisit the theme of the chapter, events should make simple things simple and complex things possible. Prototype's event system doesn't make everything simple, but it does manage the *unnecessary* complexities of modern browser events.

The watchword for this chapter has been *normalization*: making different things behave uniformly. Prototype makes two different event systems (Internet Explorer's and the W3C's) behave uniformly; it also makes native events and custom events behave uniformly. Keep this concept in mind while we look at DOM traversal in the next chapter.

CHAPTER 6

Working with the DOM

Now that you've got the foundation you need to explore advanced concepts, it's time to learn about Prototype's powerful helpers for working with the DOM.

About the DOM API

As we discussed in the last chapter, the DOM is specified in a series of documents released by the W3C.

DOM Level 1 outlines the basics you're probably used to. Levels 2 and 3 specify a series of enhancements and expansions to the DOM API, such as events, node traversal, and style sheets. Level 1 enjoys support in all modern browsers, but the other levels cannot be counted on.

Despite its obvious power and utility, at times the DOM API just doesn't feel very JavaScript-y. Methods have annoyingly long names. Some methods take a *lot* of arguments, and some methods expect their arguments in an unintuitive order.

This is an unfortunate but necessary result of the DOM's language independence. Though the most famous implementation of the DOM is JavaScript's, the APIs are designed to be implemented in nearly any modern programming language. This approach has its drawbacks (the DOM can't leverage any of JavaScript's dynamism, since it has to work in more static languages like Java), but it also has the advantage that the DOM works the same way in any language: learn once, write anywhere.

Still, we're writing our code in JavaScript, so let's make the most of it. Prototype contains a large number of extensions to the browser's DOM environment, so developers can have their DOM and eat it too.

Node Genealogy

The strange world of the DOM is filled with jargon and terms of art. In order to minimize confusion, let's look at a few of them up front.

Think of the DOM as an unseralizer. It takes a linear stream of HTML, parses it into different types of nodes, and arranges those nodes in a tree according to their relationships.

That may not have been too helpful, so I'll let the code talk (see Figure 6-1):

```
<p><u>Colorless</u> <i>green <u>ideas</u></i> sleep <b>furiously</b>.</p>
```

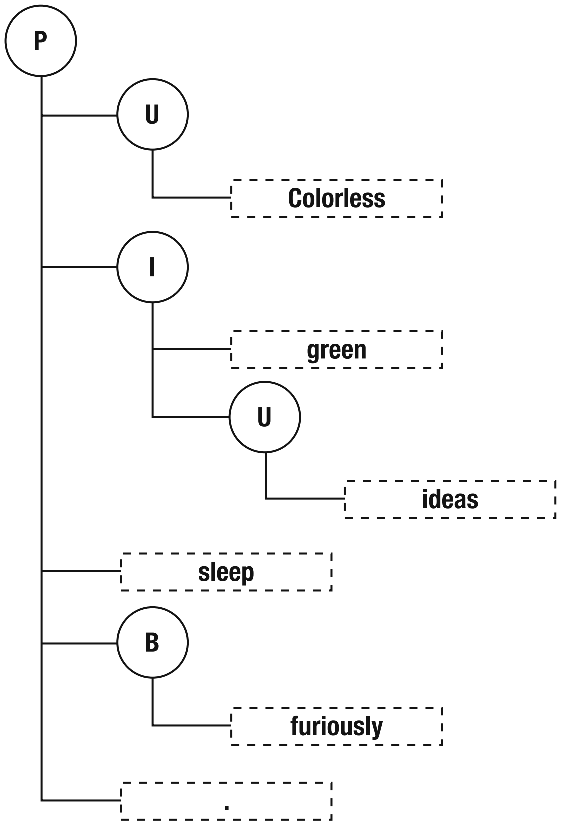


Figure 6-1. *The DOM translates a stream of HTML into a tree of nodes. This paragraph has both element nodes and text nodes as descendants.*

The resemblance of a tree like this to a *family* tree is convenient—it lets us borrow the jargon of genealogy.

Take the `p` tag at the top. It has five *children*: `u`, `i`, `"sleep"`, `b`, and `"."`. The two in quotation marks are text nodes. The other three are element nodes, and those elements have children of their own. And `"ideas"` is `p`'s great-grandchild, so to speak; it's in the third level of descendants from `p`.

The distinction is useful, then, between *children* and *descendants*, and between *parent* and *ancestor*. *Children* are all elements that are exactly one level descended from a certain node. *Descendants* are all elements that a node contains—the sum of all the node's children, its children's children, and so on.

Likewise, a node can have only one parent, but can have many ancestors.

Prototype's DOM Extensions

The DOM is broad, sterile, and built by committee. In the interest of creating a "safe" API that can be used by many different languages, it maintains a cordial distance from the features of JavaScript. Its API chooses verbose method names like `getElementById` and `getAttributeNode`—as with natural language, the cost of eliminating all ambiguity is to double the number of words.

Prototype establishes a bridge between the DOM and the commonest use cases of the typical developer. As a result, the code you write will be shorter and far more readable.

Prototype's DOM API is broad, so let's divide it into rough categories based on task: modifying, traversing, collecting, and creating.

Modifying

These methods modify properties of a DOM node or report information about a node for later modification.

The hide, show, visible, and toggle Methods

These are the most commonly used element methods. They control whether the element is visible or hidden (whether its CSS `display` property is set to `none`, thereby hiding it from view).

Controlling element display is a staple of web applications. Think of a message that should disappear after the user dismisses it. Think of a listing of items, each with a summary that should only be shown when the user mouses over that item. Think of a view stack—a group of elements that should occupy the same area of the page, with only one visible at a time.

`Element#hide` and `Element#show` control element display:

```
var foo = $('foo');
foo.hide();
foo.style.display; //-> 'none';
foo.show();
foo.style.display; //-> 'block';
```

Both methods are *idempotent*: calling them more than once in a row has the same effect as calling them just once. So you don't need to check whether an element is visible before you hide it.

Nonetheless, an easy way to figure out the display state of an element is to use `Element#visible`:

```
foo.hide();
foo.visible(); //-> false
foo.show();
foo.visible(); //-> true
```

`Element#visible` simply reports on the element's CSS `display`, returning `true` if it's set to `none`. Now imagine using all three of these methods to write code that will switch an element between visible and hidden:

```
if (element.visible()) element.hide();
else element.show();
```

You don't have to imagine it, actually, because Prototype does it for you with a method called `Element#toggle`:

```
foo.visible(); //-> true
foo.toggle();
foo.visible(); //-> false
foo.toggle();
foo.visible(); //-> true
```

`Element#toggle` is a handy abstraction for elements that should have their display state toggled each time an event (e.g., a click) is triggered.

```
// HTML:
<div class="news-item" id="item_1">
  <h3>Declaration of Independence <span id="toggle_1">Hide/Show</span></h3>
    <p id="summary_1">When, in the course of human events, it becomes necessary
for one people to dissolve the political bonds which have connected them with
another, and to assume among the powers of the earth, the separate and equal
station to which the laws of nature and of nature's God entitle them, a decent
respect to the opinions of mankind requires that they should declare the
causes which impel them to the separation.</p>
</div>

// JavaScript:
$('toggle_1').observe('click', function() { $('summary_1').toggle(); });
```

With the preceding markup, this one line of code toggles the visibility of the paragraph of text every time the `span` is clicked. In other words, items with *expanded* and *collapsed* states are perfect for `Element#toggle`.

The addClassName, removeClassName, hasClassName, and toggleClassName Methods

Showing and hiding elements is handy. But what if we want to do more with CSS? It's remarkably easy to style elements directly through JavaScript using the `style` property that exists on every node.

It's so easy, in fact, that it enables sloppy coding. Consider the following:

```
function highlight(element) {
  element.style.backgroundColor = 'yellow';
}

function unhighlight(element) {
  element.style.backgroundColor = 'white';
}
```

Imagine these two functions as mouseover/mouseout handlers for a table. When the user moves the pointer over a row, its background color is changed to yellow; when the pointer leaves the row, it goes back to the default white.

This code works as intended, but it's not very maintainable. It's a bad idea for the same reason that inline CSS in HTML is a bad idea: it mixes different layers and makes future changes more painful.

For example, what if you introduce a third background color for table rows (red, perhaps, to denote important items)? Such rows would start out red, turn yellow on mouseover, and turn *white* on mouseout. You could rewrite the handlers to take this into account, storing the initial color value somewhere so that it can be restored on mouseout, but you'd just be going further down the wrong path.

Instead, use CSS class names—they're well-suited to the task.

```
// CSS:
#items tr {
  background-color: white;
}

#items tr.important {
  background-color: red;
}
```

```
#items tr.highlighted {
  background-color: yellow;
}

// JavaScript:
function highlight(element) {
  element.addClassName('highlighted');
}

function unhighlight(element) {
  element.removeClassName('highlighted');
}
```

This is a far safer and more elegant way to apply arbitrary styling. It uses Prototype's `Element#addClassName` and `Element#removeClassName` (which do exactly what they say they do) to add and remove the styling. And, unlike the first approach, the style information is in CSS, where it should be. It won't be forgotten about three years from now when you re-skin your application.

This approach is particularly flexible because any element can have any number of class names. The HTML `class` attribute accepts any number of names separated by spaces (`<tr class="important highlighted">`).

Since it's an attribute, we can use the DOM to modify its value. In JavaScript, `class` is a reserved word, so the relevant property is called `className` instead.

```
foo.className; //-> "";
foo.className = 'important'; //-> "important"
```

This is readable but foolish. Assigning a new `class` name directly will step on any others that may have been there already. Likewise, you can remove a `class` name by giving it an empty value (`""`), but that will remove *all* of an element's `class` names, not just a specific one.

In other words, this isn't a value you change; it's a value you add to or subtract from. You use `addClassName` and `removeClassName` to ensure that these operations have no side effects.

If you've noticed how `addClassName` and `removeClassName` resemble `hide` and `show`, then you'll have predicted our next two methods: `toggleClassName` and `hasClassName`.

```
foo.className; //-> "important"

foo.toggleClassName('highlighted');
foo.hasClassName('highlighted'); //-> true
foo.hasClassName('important');   //-> true

foo.toggleClassName('highlighted');
foo.className; //-> "important"
```

All four methods expect the `class` name as an argument; otherwise, they work the same way as `hide`, `show`, `visible`, and `toggle`.

The setStyle and getStyle Methods

Despite the warnings I just gave you, sometimes you'll have no choice but to set style properties directly. Size and position (`width`, `height`, `top`, and `left`) can be declared in a style sheet, but complex use cases will require manipulating these properties dynamically as a result of user input.

The native DOM style API forces you to set these properties one at a time:

```
foo.style.top    = '12px';
foo.style.left   = '150px';
foo.style.width  = '100px';
foo.style.height = '60px';
```

But Prototype's `Element#setStyle` can apply CSS styles in bulk. It accepts an object literal as an argument, directly correlating JavaScript object properties and values with CSS properties and values:

```
foo.setStyle({
  top:    '12px',
  left:   '150px',
  width:  '100px',
  height: '60px';
});
```

`Element#setStyle`'s counterpart is named `Element#getStyle`, appropriately—though it's not *quite* a counterpart. It retrieves the given style property on the node, regardless of that style's origin.

```
// CSS:
#foo {
  font-family: "'Helvetica Neue', Helvetica, Arial, sans-serif"
  padding: 10px;
  border: 15px;
}

// JavaScript:
foo.getStyle('font-family');
//-> "'Helvetica Neue', Helvetica, Arial, sans-serif"
foo.getStyle('padding-left');
//-> "10px"
foo.getStyle('border-top');
//-> "15px"
```

Upon first look, this seems unnecessary—why not just read from `foo.style`? But the `style` property doesn't read from the entire style cascade—just inline styles.

```
// HTML:
<div id="foo" style="display: inline;"></div>

// JavaScript:
foo.style.fontFamily;  //-> undefined
foo.style.paddingLeft; //-> undefined
foo.style.borderTop;   //-> undefined

0foo.style.display;     //-> 'inline'
```

So the element's `style` object *may* contain what you're looking for—if a property reports a value, then it's sure to be the correct one, since inline styles are the most specific. But most of the time it will return `undefined`, since most style declarations come from a CSS block or an external CSS file.

Internet Explorer and the DOM both provide APIs for determining the *true* style of a certain element. `Element#getStyle` acts as a wrapper around both.

The update, replace, insert, and remove Methods

Changing the content of an element is a common task, but it's not an easy one. Suppose we want to change

```
<div id="foo"><p>bar</p></div>
```

to

```
<div id="foo"><span>thud</span></div>
```

There are two ways to do this in modern browsers. One uses DOM Level 1 methods, relying on the *hierarchy-of-nodes* model:

```
var foo = $('foo');
var span = document.createElement('span');
var text = document.createTextNode('thud');
span.appendChild(text);
foo.replaceChild(span, foo.firstChild);
```

The other uses the element's `innerHTML` property (first introduced in Internet Explorer, and then implemented by other browsers). It treats an element's contents as a string:

```
foo.innerHTML; //-> "<p>bar</p>";
foo.innerHTML = "<span>thud</span>";
```

Which should be used? There isn't an easy answer. The `innerHTML` technique has the advantage of simplicity (in most cases) *and* speed (it can be up to an order of magnitude faster), but the disadvantage of being only a de facto standard, prone to irregular behavior.

Meanwhile, the DOM technique is considered "purer" by standardistas—but, just like most other parts of the DOM, it can feel clunky and verbose, with many lines of code needed to do simple tasks. Purity comes at a cost.

Those that champion one approach and slander the other are presenting a false dilemma. Sometimes it makes sense to treat markup as a node tree; sometimes it makes sense to treat markup as a string. Both techniques work well in all modern browsers; you don't have to choose one and stick with it.

If embrace of the nonstandard `innerHTML` property gives you pause, consider that the scope of what we can accomplish *with today's browsers* would be severely impeded if we restricted ourselves to that which is defined by a W3C specification. We'd be building sites of interest to academics and power users, but which ignore the real world and the market-leading browser. Writing JavaScript for today's Web requires balancing the ideal and the practical.

As a way forward, we can push for standardization of all the nonstandard APIs we use—"paving the cowpaths," as it's called. For instance, the HTML 5 specification aims to prescribe a standard behavior for `innerHTML` so that it can be used without guilt or caveat.

I find that the DOM approach works best when creating HTML programmatically—when the structure or content of the inserted markup isn't the same every time—because building long strings in JavaScript isn't my idea of a fun afternoon. For simpler cases, `innerHTML` is easier and faster. Figure out the balance you're comfortable with.

The first three methods we'll deal with—`update`, `replace`, and `insert`—don't force you to pick one approach or the other. All three can accept *either* a markup string *or* a node.

Using update

`Element#update` *changes* the contents of an element. Think of it as a thin wrapper around `innerHTML`—just as assigning to the `innerHTML` property will erase whatever was there before, `update` will discard the original contents of the element, replacing it with what you've given.

```
// HTML (before):
<p id="foo"><b>narf</b></p>
// JavaScript:
$('foo').update('<span>thud</span>');

// HTML (after):
<p id="foo"><span>thud</span></p>
```

It boasts several advantages over `innerHTML`:

- As explained, it can take a DOM node or a string.
- The convenient "automatic script evaluation" you were introduced to in Chapter 4 also applies to `Element#update`. Any `script` elements in the inserted markup will be removed; the code inside them will be extracted and evaluated after the element has been updated.
- It gracefully handles some special cases where Internet Explorer tends to choke. For instance, most table elements have read-only `innerHTML` properties, as do oddballs like `col` and `select`.

Let's try a DOM node instead of an HTML string:

```
var span = document.createElement('span');
span.appendChild(document.createTextNode('thud'));
$('foo').update(span);
$('foo').innerHTML; //-> "<span>thud</span>"
```

CHAINING

Prototype's augmentation of DOM node instance methods opens the door to *method chaining*: a syntactic shortcut that makes lines of code read like sentences.

Many of the methods in this chapter—specifically those that do not need to return other values—will return the elements *themselves*. Consider this code:

```
$('foo').addClassName('inactive');
$('foo').hide();
```

Because both `addClassName` and `update` return the element itself, this code can be simplified:

```
$('foo').addClassName('active').hide();
```

Chaining method calls like this—joining them in a line, each acting upon the return value of the last—can increase code clarity when used judiciously. In this example, we've also optimized the code, removing a redundant call to `$` to re-fetch the element.

Look out for methods that *do not* return the original element. Consider `Element#wrap`, which returns the new created parent node:

```
$('foo').wrap('div');
$('foo').addClassName('moved');

// wrong:
$('foo').wrap('div').addClassName('moved');
```

We've changed the meaning of the code by accident: instead of adding a class name to the element with an ID of `foo`, we're now adding it to the `div` that was created and returned by `wrap`. Reversing the order of the method calls preserves our intent:

```
// right:
$('foo').addClassName('moved').wrap('div');
```

Similarly, note that `Element#replace` will return the original element, but that element has been replaced and is no longer a part of the DOM tree. If you want to work with the content that has replaced it, you'll need to obtain that reference some other way.

Using replace

`Element#replace` is nearly identical to its brother `update`, but can be used to *replace* an element (and all its descendants) instead of just changing its contents.

```
// HTML (before):
<div id="foo"><span>thud</span></div>

// JavaScript:
$('foo').replace('<p id='foo'><b>narf</b></p>');

// HTML (after):
<p id="foo"><b>narf</b></p>
```

The new content occupies the same position in the document as its predecessor. So `replace` is a way to remove an element, keep a finger on its spot in the DOM tree, and then insert something else at that spot.

Using insert

`Element#insert` appends content to an element without removing what was there before. We flirted with `insert` in Chapter 4, so the syntax will be familiar to you:

```
// HTML (before):
<div id="foo"><span>thud</span></div>

// JavaScript:
$('foo').insert("<span>honk</span>", 'top');

// HTML (after):
<div id="foo"><span>honk</span><span>thud</span></div>
```

The second argument is the position of insertion—either `before`, `after`, `top`, or `bottom`. This argument will default to `bottom` if omitted.

```
// equivalent in meaning:
$('foo').insert("<span>honk</span>", 'bottom');
$('foo').insert("<span>honk</span>");
```

A more robust syntax can be used to insert several things at once. Instead of a string, pass an object as the first argument—the keys are insertion positions and the values are HTML strings.

```
// HTML (before):
<div id="foo"><span>thud</span></div>

// JavaScript:
$('foo').insert({ top: "<span>honk</span>", bottom: "<span>narf</span>" });

// HTML (after):
<div id="foo"><span>honk</span><span>thud</span><span>narf</span></div>
```

The positions `before` and `after` are similar to `top` and `bottom`, respectively, but you insert the new elements outside the boundaries of the given element—as siblings, rather than children.

```
// HTML (before):
<div id="foo"><span>thud</span></div>

// JavaScript:
$('foo').insert({ before: "<span>honk</span>", after: "<span>narf</span>" });

// HTML (after):
<span>honk</span><div id="foo"><span>thud</span></div><span>narf</span>
```

Using remove

In the DOM API, you must remove an element by calling the `removeChild` method on its *parent*:

```
// to remove "foo"
$('foo').parentNode.removeChild($('foo'));
```

Prototype adds `Element#remove` in order to circumvent this annoyance:

```
$('foo').remove();
```

Note that removing an element from the document doesn't make it vanish; it can be reappended somewhere else, or even modified while detached. But a detached node won't respond to calls to `$` or `document.getElementById` (since the node is no longer in the document), so make sure you preserve a reference to the node by assigning it to a variable.

```
// to remove "foo" and append it somewhere else
var foo = $('foo');
foo.remove();
$('new_container').appendChild(foo);
```

The readAttribute and writeAttribute Methods

Prototype's `readAttribute` and `writeAttribute` methods are used to get and set attributes on elements.

"Aren't these superfluous?" you ask. "Doesn't the DOM give us `getAttribute` and `setAttribute`?" Yes, it does, and browsers also expose attributes as properties of their object representations—a holdover from the pre-DOM days. So, for instance, the `href` attribute of a link can be fetched with `$('foo').getAttribute('href')` or even `$('foo').href`.

But these approaches have compatibility problems. Internet Explorer, in particular, exhibits a host of bugs in this area, thwarting our attempts to get identical behavior from all major browsers. `Element#readAttribute` and `Element#writeAttribute` are wrappers that ensure identical behavior.

Using readAttribute

Let's look at some examples of surprising `getAttribute` behavior in Internet Explorer:

```
// HTML:
<label id="username_label" class="required" for="username">
  <input type="text" id="username" name="username"
   disabled="disabled" />
</label>
<a id="guidelines" href="/guidelines.html">Username guidelines</a>

// JavaScript:
var label = $('username_label');
label.getAttribute('class');     //-> null
label.getAttribute('for');       //-> null

var input = $('username');
input.getAttribute('disabled');  //-> true

var link = $('guidelines');
link.getAttribute('href');  //-> "http://www.example.com/guidelines.html"
```

The `label` element has `class` and `for` attributes set, but Internet Explorer returns `null` for both. The `input` tag has a `disabled` attribute with a value of “disabled” (in accordance with the XHTML spec), but Internet Explorer doesn’t return the literal value—it returns a Boolean. And the `a` element points to an absolute URL on the same server, but Internet Explorer gives us the “resolved” version when we ask for its `href`.

In all three cases, we’re expecting the *literal value* that was set in the markup—that’s how `getAttribute` is supposed to work, and that’s how the other browsers do it. When it was released, Internet Explorer 6 had the best DOM support of any browser, incomplete as it was; now, six years later, bugs like these make Internet Explorer the slowpoke of the bunch.

In nearly all cases, the value we want is hidden somewhere—we’ve just got to find it. Prototype does the heavy lifting for you.

```
var label = $('username_label');
label.readAttribute('class'); //-> "required"
label.readAttribute('for');   //-> "username"

var input = $('username');
input.readAttribute('disabled'); //-> "disabled"

var link = $('guidelines');
link.readAttribute('href'); //-> "/guidelines.html";
```

Use `readAttribute` anywhere you’d use `getAttribute`. It’s safer.

Using writeAttribute

As you may expect, `writeAttribute` lets you set attribute values safely in all browsers, succeeding where Internet Explorer’s `setAttribute` fails:

```
label.setAttribute('class', 'optional'); // fails
label.writeAttribute('class', 'optional'); // succeeds
```

But that’s not all. It adds a major syntactic time-saver for writing multiple attributes at once—simply pass an object literal full of attribute names and values.

```
input.writeAttribute('id', 'user_name');
label.writeAttribute({
  title: 'Please choose a username.',
  'class': 'optional',
  'for':   'user_name'
});
```

You might recognize this pattern from `Element#setStyle`, discussed earlier in the chapter.

There's only one gotcha: `class` and `for` are reserved words in JavaScript (they have special meaning), so be sure to wrap them in quotes, as in the preceding example.

Later in the chapter, you'll use this pattern as part of a powerful element-creation API.

Traversing and Collecting

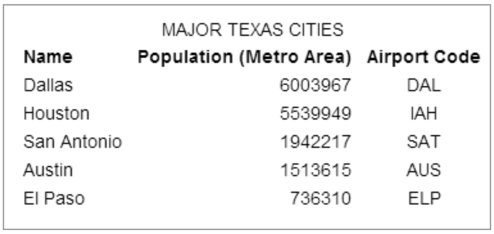
The tasks we're about to cover could broadly be defined as "getting nodes from other nodes." *Traversal* is getting from one node to another; *collection* is asking for a group of nodes that relates to the node you're on in some way.

Navigating Nodes

Since traversal is all about swiftly navigating complex node structures, let's create a complex node structure:

```
<table id="cities">
  <caption>Major Texas Cities</caption>
  <thead>
    <tr>
      <th scope="col">Name</th>
      <th scope="col" class="number">Population (Metro Area)</th>
      <th scope="col">Airport Code</th>
    </tr>
  </thead>
  <tbody>
    <tr id="dallas">
      <td>Dallas</td>
      <td class="number">6003967</td>
      <td class="code">DAL</td>
    </tr>
    <tr id="houston">
      <td>Houston</td>
      <td class="number">5539949</td>
      <td class="code">IAH</td>
    </tr>
    <tr id="san_antonio">
      <td>San Antonio</td>
      <td class="number">1942217</td>
      <td class="code">SAT</td>
    </tr>
    <tr id="austin">
      <td>Austin</td>
      <td class="number">1513615</td>
      <td class="code">AUS</td>
    </tr>
    <tr id="el_paso">
      <td>El Paso</td>
      <td class="number">736310</td>
      <td class="code">ELP</td>
    </tr>
  </tbody>
</table>
```

Nothing too fancy here—just an ordinary data table (see Figure 6-2). But I've taken a couple liberties with the styling. Anything with a `class` of `number` gets right-aligned (so that all the digits line up); anything with a `class` of code gets center-aligned.



MAJOR TEXAS CITIES

Name	Population (Metro Area)	Airport Code
Dallas	6003967	DAL
Houston	5539949	IAH
San Antonio	1942217	SAT
Austin	1513615	AUS
El Paso	736310	ELP

Figure 6-2. *A simple, contrived data table*

Drop this code into your `index.html` file. We're done with the breakfast log, sadly, so you can clear that stuff out, but leave the `script` tag that loads Prototype.

The up, down, next, and previous Methods

Visualize a DOM tree. The root element, `document`, is at the top, roughly corresponding to the `html` tag in your markup. And from there the tree branches to reflect the complex relationships of nodes in the document. A `table` node branches into `thead` and `tbody`; `tbody` branches into several `trs`; each of those `trs` branches into several `tds`.

Now imagine you're a lowly `td` tag in the body of our table. You know that your parent node is a `tr`, and that you've got a reference to that table row in your `parentNode` property.

```
td.parentNode; //-> <tr>
```

You don't know who your grandparent node is, but you can ask the `tr`:

```
td.parentNode.parentNode; //-> <tbody>
```

And so on, up the line:

```
td.parentNode.parentNode.parentNode; //-> <table>
```

Tedious, isn't it? Makes me want to be an orphan. But it's even worse in the opposite direction: imagine you're a `table` element and you want to find one of your `td` grandchildren.

The DOM foresees these needs, but addresses them only partially. We need better ways to jump from one node to another, no matter which direction we're going.

With `Element#up`, `Element#down`, `Element#next`, and `Element#previous`, we have the fine control that the DOM lacks. Each method returns one element in the specified direction.

Imagine we've got a reference to the `td` with the content "Houston." From there, we can traverse with ease:

```
td.up();   //-> <tr>
td.next(); //-> <td class="number">
td.previous(); //-> null
td.down();     //-> null
```

Calls to `up` and `next` return the parent node and the next sibling, respectively. Calls to `down` and `previous` return `null` because no element is found in that direction.

■ **Note** These four methods ignore text nodes entirely. When you call `next`, you're asking for the next *element* sibling, which may not be the same as the node's `nextSibling` property.

Now let's jump up one level:

```
var tr = td.up();

tr.up();       //-> tbody
tr.next();     //-> tr#san_antonio
tr.previous(); //-> tr#dallas
tr.down();     //-> td
```

This time, we get results in all four directions: `next` and `previous` point to table rows 1 and 3, while `up` and `down` point to the parent node and the first child node.

To repeat, each of these methods returns *one* result. If there is more than one element to choose from, it will pick the *first* one that satisfies its search.

These traversal methods become even more useful when given arguments. All four take two types of arguments:

A CSS selector string: Checks potential matches against the selector; accepts the same wide range of selectors as `$$`.

A numeric index: Specifies how many matches should be *skipped*. (Or think of it this way: if *all* the matches were returned in an array, this would be the index of the one you want.)

```
tr.up('table');     //-> table
tr.next(1);         //-> tr#austin
tr.down('td.code'); //-> td.code
tr.down('td', 1);   //-> td.number
```

As you can see, both arguments are optional, and the order is flexible. You can pass in an index or a selector as the first argument; but if you pass *both*, the selector needs to come first.

The select Method

Don't forget about our old friend from Chapter 2. `$$`, the CSS selector function, searches the entire document, but it has an instance-method counterpart (`select`) that can search any subset of a DOM tree.

`Element#selector` works like `Element#down`, except it returns an *array* of elements.

```
tr.select('.code');
// -> [td.code, td.code, td.code, td.code, td.code]
tr.up('table').select('.number');
// -> [th.number, td.number, td.number,
 td.number, td.number, td.number]
tr.up('table').select('td:last-child');
// -> [td.code, td.code,
 td.code, td.code, td.code]
```

The ancestors, descendants, and immediateDescendants Methods

These methods correspond to the groups searched by `Element#up` and `Element#down`:

- `ancestors` returns all ancestors of the node, starting with its parent node and ending with the `html` element.
- `descendants` returns all element descendants of the node in depth-first order. This is equivalent to calling `getElementsByTagName('*')` in the node's scope.
- `immediateDescendants` returns all element children of the node in the order they appear. This is equivalent to the element's `childNodes` property—but with all text nodes filtered out. (`children` would be a better name, but Safari uses that property name for something else.)

The siblings, previousSiblings, and nextSiblings Methods

These methods correspond to the groups searched by `Element#previous` and `Element#next`:

- `previousSiblings` and `nextSiblings` return all the element nodes that come before and after the given node, respectively. The returned collections are ordered based on proximity to the original element, meaning that `previousSiblings` will return a collection in *reverse* DOM order.
- `siblings` returns the union of `previousSiblings` and `nextSiblings` arranged in DOM order. This collection does not include the original node itself. This is equivalent to calling `immediateDescendants` on the node's parent node, and then removing the original node from the returned collection.

Creating Nodes

Individually, all these methods are simply helpers—convenience methods for repetitive tasks. But when they combine, they form a strong platform that allows for a whole new level of coding. The `Element` constructor is the Captain Planet of the Prototype DOM extensions—a force greater than the sum of its parts.

Think of the example we've been using in a specific context. Suppose we were building a site where a user could select any number of cities and compare some of their qualities. To make the UI snappy, we'd load city data via Ajax and stuff it into our comparison table dynamically. The data itself has to come from the server, but we can offload some of the easier stuff to the client side.

Most of this falls outside the scope of this chapter, but there's one part we can extract into a simple example. Let's say we want to build a new row at the bottom of our table—

one that will add up all the populations of the cities and display a total. In HTML form, the row would look something like this:

```
<tr class="total">
  <td>Total</td>
  <td class="number">15,736,058</td>
  <td class="code"></td>
</tr>
```

We'll give the `tr` its own class so we can style it to look different from the other rows. And we'll leave the last cell empty because it's not applicable to this row, of course.

But now we've got to decide between two equally ugly ways of generating this HTML dynamically. We could use `Element#insert` with a string:

```
var html = "<tr class='total'>";
html += "<td>Total</td>";
html += "<td class='number'>" + totalPopulation + "</td>";
html += "<td class='code'></td>";
html += "</tr>";

$('cities').down('tbody').insert(html, 'bottom');
```

But I hate building long strings like that. It's a syntax error minefield. So, we could stick to the DOM way:

```
// First, create the row.
var tr = document.createElement('tr');
// We need to "extend" the element manually if we want to use
// instance methods.
$(tr).addClassName('total');

// Next, create each cell individually.
var td1 = document.createElement('td');
td1.appendChild(document.createTextNode('Total'));

var td2 = document.createElement('td');
$(td2).writeAttribute('class', 'number');
td2.appendChild(document.createTextNode(totalPopulation));

var td3 = document.createElement('td');
$(td3).writeAttribute('class', 'code');
```

```
// Now append each cell to the row...
tr.appendChild(td1);
tr.appendChild(td2);
tr.appendChild(td3);

// ...and append the row to the table body.
$('cities').down('tbody').insert(tr, 'bottom');
```

Three times as many lines! As is often the case, one approach is easy but sloppy, and the other is harder but more "correct." Can't we split the difference?

```
// First, create the row.
var tr = new Element('tr', { 'class': 'total' });

// Next, create each cell and append it on the fly.
tr.appendChild( new Element('td').update('Total') );
tr.appendChild( new Element('td',
 { 'class': 'number'}).update(totalPopulation) );
tr.appendChild( new Element('td', { 'class': 'code' }) );

// Now append the row to the table body.
$('cities').down('tbody').insert(tr, 'bottom');
```

We haven't talked about this technique yet, but it ought to seem familiar anyway. The glue in the middle, the `new Element` part, takes a tag name as its first argument, creates that element, calls `Element.extend` on it (to give it the Prototype instance methods), and then returns the element.

The rest of it is stuff we've already covered:

- The optional second argument to `Element` is an object with the attribute/value pairs the element should have. `Element#writeAttribute` takes care of that part.
- Instead of the annoying `document.createTextNode`, we can use Prototype's own `Element#update` to set the text content of a new element.
- The last step is the same in all cases. We use `Element#down` to hop from the `table` to its `tbody`, and then we use `Element#insert` to place our new element after all the other rows.

The wrap Method

Let's take a time-out from the example to talk about `Element#wrap`.

Sometimes you'll want to create a new element to act as a container for something that's already on the page. This is one mode of attack for browser bugs—rendering issues, for example, can sometimes be defeated with a "wrapper" element.

Prototype's `Element#wrap` is shorthand for creating an element and specifying its contents all at once. Its arguments are identical to those of the `Element` constructor:

```
// wrap a TABLE in a DIV
var someTable = $('cities');
var wrapper   = someTable.wrap('div', { 'class': 'wrapper' });
```

Here, since the `table` already exists in the DOM, we don't have to explicitly append it somewhere. The `div` is created at the spot occupied by the `table`; the `table` then gets added as a child of the `div`.

Wrapping an element that isn't yet in the document, of course, is a different matter:

```
// add a link to a list
var li = new Element('a', { href: "http://google.com"
 }).update("Google").wrap("li");
$('links').insert(li);
```

Like most methods on `Element`, `wrap` can be chained. But, as these examples show, it returns the *wrapper*, not the original element.

Putting It Together

Back to the example. Let's write a function that, when given a table of the sort we've written, will automatically calculate the total and insert a new row at the bottom. Insert this block in the `head` of the page, right below where Prototype is loaded:

```
<script type="text/javascript">
  function computeTotalForTable(table) {
    // Accept a DOM node or a string.
    table = $(table);

    // Grab all the cells with population numbers in them.
    var populationCells = table.select('td.number');
    // Add the rows together.
    // (Remember the Enumerable methods?)
    var totalPopulation = populationCells.inject(0, function(memo, cell) {
      var total = cell.innerHTML;
      // To add, we'll need to convert the string to a number.
      return memo + Number(total);
    });

    // We've got the total, so let's build a row to put it in.
    var tr = new Element('tr', { 'class': 'total' });

    tr.insert( new Element('td').update('Total') );
    tr.insert( new Element('td',
     { 'class': 'number' } ).update(totalPopulation) );

    // Insert a cell for the airport code, but leave it empty.
    tr.insert( new Element('td', { 'class': 'code' }) );

    table.down('tbody').insert(tr);
  }</script>
```

This code does a lot of stuff, so let's look at it piece by piece.

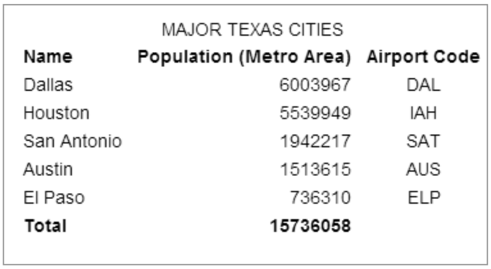
First, we use the `$` function on the table argument so that we can be sure we're working with a DOM node. Then we use `Element#select` to grab all the table cells with a class of `number`—there will be one per table row.

Now that we have all of the table cells that contain population figures, we add them together with `Enumerable#inject`. We start off with `0`, adding the value of each cell to that figure as we loop through the cells. The returned value is the sum of all the numbers contained in the cells.

Now that we have the number, we need to lay the DOM scaffolding for it. We build a `tr` with a class of `total`, and then three `td`s with contents that correspond to that of the existing cells. Our total population figure gets inserted into the third cell via `Element#update`. We insert each cell as a child of the `tr` upon creation; since `Element#insert` adds new nodes at the end, by default, these elements will appear on the page in the order they're inserted.

All that's left to do is test it! Open `index.html` in Firefox, and pass our population table to the `computeTotalForTable` function (see Figure 6-3):

```
computeTotalForTable('cities');
```



MAJOR TEXAS CITIES

Name	Population (Metro Area)	Airport Code
Dallas	6003967	DAL
Houston	5539949	IAH
San Antonio	1942217	SAT
Austin	1513615	AUS
El Paso	736310	ELP
Total	**15736058**	

Figure 6-3. *JavaScript can add better than I can.*

Summary

There's a lot of meat to Prototype's DOM support—primarily because that's where most of your headaches come from. The DOM's verbosity and uneven browser support are the proverbial "rock" and "hard place" of JavaScript development.

An exploration of the DOM, however, isn't complete without attention paid to events. The next chapter will exemplify the power of Prototype's element traversal methods in the context of handling standard (and not-so-standard) browser events.

CHAPTER 7

Advanced JavaScript: Functional Programming and Class-Based OOP

JavaScript is a multi-paradigm language. No matter how well you *think* you know how to use it, you're destined to find some style of writing code that confuses the hell out of you.

This is good news, if you'll believe it. It means that there's often a better, shorter, more secure, or easier-to-understand way of doing something.

Earlier chapters introduced you to object-oriented programming and functional programming. So you're probably familiar with what they are, but may not realize yet why they're *useful*. In this chapter, we'll revisit these techniques, exploring advanced use cases for both.

Object-Oriented JavaScript Programming with Prototype

The term *object-oriented programming* (OOP) has become nearly meaningless with over-use, but I'll try to wring out a few final drops of meaning. JavaScript itself is an object-oriented language, as we've discussed, but most of the common OOP concepts—class definitions, clear inheritance chains, and so on—aren't built into the language.

Prototype builds a class-based facade around the prototypal OOP model of JavaScript. In this chapter, you'll learn how to use classes the Prototype way.

Why OOP?

The jaded developer might wonder whether I'm espousing OOP as the alpha and omega of programming styles. As Steve Yegge once quipped, advocating "object-oriented programming" is like advocating "pants-oriented clothing"; it elevates one architectural model to an overimportant position.

In the DOM scripting world, OOP is often—but not always—the right tool for the job. Its advantages mesh well with the realities of the browser environment.

Cleanliness

JavaScript 1.x has no explicit namespacing and no package mechanism—no comprehensive way to keep different pieces of code from stepping on each other. If your web app needs, say, 50 lines of JavaScript, this isn't a problem; if it needs three external libraries and a bunch of behavior code, you'll run into naming problems. Scripts share the global space.

In other words, if I have a function named `run`, I'll need to make sure that none of the scripts I rely upon defines a `run` function, or else it will be overwritten. I could rename my function to something unique—like `myRun` or `dupont_Run`—but this is just a stall tactic. Ensuring uniqueness of function names becomes exponentially harder as the number of functions increases. (PHP's standard library is an extreme example of the sort of clutter that can accumulate this way.)

OOP helps solve this. It lets me define methods in more appropriate contexts. Instead of a global function called `stringReplace`, JavaScript defines `replace` as an instance method on strings.

Sharing a scripting environment is like sharing an apartment. If you don't help keep the place clean, your roommates will think of you as an inconsiderate bastard. Don't be that guy. If you must be messy, be messy in your own room. Common areas need to remain tidy.

Encapsulation

OOP helps keep things organized through bundling. An object contains all the methods and properties associated with it, minimizing clutter and affording portability.

Information-Hiding

Many clocks rely on an elaborate system of gears to tell time, but humans see very little of what goes on inside a clock. We don't have to figure out the time from the positions of the gears; we can look at the hands on the clock face. In other words, the gears are important to the *clock*, but they're not important to *us*.

Most real-world objects have a "public" interface (the way the outside world uses it) and a "private" interface (the way the object does the tasks it needs to do). OOP works the same way, letting a developer produce code that is easier to understand and more relevant on the line where it's used.

Brevity

Brevity is a by-product of cleanliness. Functions can be named more concisely when they don't have to worry about conflict. Which would you rather type?

```
stringReplace("bar", "r", "z");
"bar".replace("r", "z");
```

The second option is seven characters shorter than the first. Over time, this savings will add up—your code will be clearer and your wrists will thank you.

Remedial OOP: Namespacing

Before we get into "true" OOP, let's look at how objects can be used to isolate our code and make it more resilient in a shared scripting environment.

Because scripts from different authors often coexist on the same page, it's a bad idea to define too many functions in the global scope. The more there are, the more likely one of them will overwrite someone else's function with the same name.

Instead, what if you were to define *one* object in the global scope, and then attach all your functions to that object? You'd balk at defining a method named `run`, but `BreakfastLog.run` is much less risky. The first line of defense in the cleanliness wars is namespacing your code.

Let's look at some of our code from a previous chapter:

```
function submitBreakfastLogEntry(event) {
  if (event.target.id === 'cancel')
    cancelBreakfastLogEntry(event);
  // ... et cetera
}

function cancelBreakfastLogEntry(event) {
  event.stop();
  event.target.up('form').clear();
  // ... et cetera
}
```

Since they're related, these functions ought to be placed into a common namespace.

```
var BreakfastLog = {
  submitEntry: function(event) {
    this.form = event.target.up('form');
    if (event.target.id == "cancel")
      this.cancelEntry(event);
    // et cetera
  },

  cancelEntry: function(event) {
    event.stop();
    this.form.clear();
    // ... et cetera
  }
};
```

Here, we're using the familiar object literal in a new way: as a container for like-minded methods and variables. Remember that `{}` is a shortcut for `new Object`, and that the `new` keyword *triggers a new scope*.

Thus, inside any particular method, the keyword `this` refers to the `BreakfastLog` object. Code running *outside* the `BreakfastLog` object must explicitly call `BreakfastLog.cancelEntry`, but code running *inside* has the privilege of referring to it as `this.cancelEntry`.

Similarly, the object itself can be used to pass information between methods. Notice how `cancelEntry` can read `this.form`, a property set by `submitEntry` (and that would be referred to as `BreakfastLog.form` from another scope).

I've been casually referring to this as "namespacing," even though it's not a true namespacing solution like that of Perl or Ruby. There's still a risk of naming collisions—after all, another library you use could create a `BreakfastLog` object—but it's much less likely when there are fewer items in the global namespace. We can't eliminate the risk altogether, but we can minimize it.

Advanced OOP: Using Classes

In Chapter 1, you learned about JavaScript's *prototypal inheritance* model. Each function has a property called `prototype` that contains the "template" for making new objects of that type. Since every function has this property, every function can be instantiated. (Although this also means that "instantiate" isn't the correct term; prototypal inheritance does not distinguish between classes and instances.)

Prototypal inheritance isn't *better* or *worse* than class-based inheritance, but you may not feel that way if you're not used to it. Most of what you're used to just isn't there: no class definitions, no explicit way to define inheritance, no information-hiding. These things are hard not to miss.

Luckily, nearly anything can be done in JavaScript if you're willing to hack at it long enough. Prototype features a `Class` object that can mimic most of the features of class-based inheritance. It's not meant to change the way JavaScript works; it's just a different approach to OOP—an extra tool to have on your belt.

Example

Here's a glimpse at the project we'll be working on in the second part of the book. It's a site about football (*American football* to the rest of the world). If you know nothing about the game, that's OK—the bits you need to know will be learned along the way.

A football team consists of an offense and a defense, each of which is divided into many positions and roles. But every player has the ability to score points. (For some players, it would only happen by accident, but it's still possible.)

So each player has certain characteristics (things he is) and certain capabilities (things he can do). Those characteristics and capabilities vary based on his position, but there are some that are common to all players. This sounds like the perfect use case for class-based OOP.

Creating the Base Class

First, we'll define a `Player` class—one that describes a generic football player:

```
var Player = Class.create({
  initialize: function(firstName, lastName) {
    this.firstName = firstName;
    this.lastName  = lastName;
    this.points    = 0;
  },

  scorePoints: function(points) {
    this.points += points;
  },

  toString: function() {
    return this.firstName + ' ' + this.lastName;
  }
});
```

Let's consider the bold sections of code in order.

`Class.create` is the Prototype method for building a new class. It accepts one argument: an object literal that contains the properties the class should have. Most of the time, these will be methods, but they can be whatever you like.

`Class.create` returns a function that can be instantiated. Be sure to assign it to a variable (`Player` in this case) so that you can use it later. That will be the name of your class.

Next, look at `initialize`. Prototype treats it as the class's *constructor*. When the `Player` class is instantiated (with `new Player`), this method will be called with the given arguments. Ours takes two arguments: `firstName` and `lastName`.

Finally, look at `toString`. JavaScript gives a `toString` property (with a function value) special treatment: if it's defined, it will be used to convert the object into a string.

Firebug explains it better:

```
// redefine the toString method for all arrays (bad idea)
Array.prototype.toString = function() {
  return "#<Array: " + this.join(', ') + '>';
};
var someArray = [1, 2, 3];

// works on both explicit and implicit string conversions:
someArray.toString(); //-> "#<Array: 1, 2, 3>"
someArray + '';       //-> "#<Array: 1, 2, 3>"
```

So, we're defining a `toString` method for reporting the string value of an instance of `Player`. For now, it's the player's full name.

This class doesn't do much, but it's ready to use. Let's have some fun:

```
var p = new Player('Andrew', 'Dupont');
p.firstName; //-> "Andrew"
p.lastName;  //-> "Dupont"

p + '';      //-> "Andrew Dupont"

p.points;     //-> 0
p.scorePoints(6);
p.points;     //-> 6
```

Notice how Prototype is mixing some of its own conventions with the native constructs of JavaScript. As with prototypal OOP, you'll use the `new` keyword to generate instances of objects. Prototype's `Class.create` method, however, builds a nearly leakproof abstraction over JavaScript's inheritance model. You'll be able to *think* in terms of class-based OOP, using the same techniques you'd use when architecting Java or Ruby code.

The preceding code creates a generic football player with my name. This is exciting for me, as it's as close as I'll get to playing professional football. But it's probably not too exciting for you. All the *interesting* stuff will be contained in `Player`'s subclasses.

Creating the Subclasses

Football players have many things in common, but also many differences. Some are broad and slow; some are tall, skinny, and fast; some are middle-aged and do nothing but kick the ball.

We want to repeat ourselves as little as possible, so let's make sure that all common characteristics are defined in `Player`. But the differences require that we make a class for each position.

A *quarterback* runs the offense. He can throw the ball to a wide receiver, hand it to a running back, or even run with it himself.

```
var Quarterback = Class.create(Player, {
  initialize: function($super, firstName, lastName) {
    // call Player's initialize method
    $super(firstName, lastName);

    // define some properties for quarterbacks
    this.passingYards = 0;
    this.rushingYards = 0;
  },

  throwPass: function(yards) {
    console.log(this + ' throws for ' + yards + 'yds.');
    this.passingYards += yards;
  },

  throwTouchdown: function(yards) {
    this.throwPass(yards);
    console.log('TOUCHDOWN!');
    this.scorePoints(6);
  }
});
```

Notice that the bold parts introduce two new concepts.

Just as before, we define the class using `Class.create`. This time, though, we've placed an extra argument at the beginning: the class to be extended. Because `Quarterback` is based on `Player`, it will have all of `Player`'s properties and methods, plus whatever extra stuff `Quarterback` defines for itself. Observe how, for instance, `Quarterback` has a `scorePoints` method, even though it wasn't defined on `Quarterback` directly. That method was *inherited* from `Player`.

So we've established a relationship between `Player` and `Quarterback`: `Player` is `Quarterback`'s *superclass*, and `Quarterback` is `Player`'s *subclass*.

`Quarterback`'s `initialize` method appears to perform some voodoo. As the comments indicate, we're replacing `Player`'s initialize method, but we're still able to call the original method. To do so, we place an argument named `$super` at the front of `initialize`'s argument list. The name of the argument serves as a signal that we want to override `Player#initialize`. The original method is passed into the function as `$super`, and we call it just like we would call any other method.

Keep in mind that `initialize` *still expects two arguments, not three*. We define `initialize` to take three arguments, but the first one is filled in automatically behind the scenes. The public interface has not changed.

WHY $SUPER?

"Another dollar sign?" you ask. "You've got to be kidding me."

No, I'm serious. There are two reasons why Prototype uses the dollar sign (`$`) as part of its naming scheme:

- As a shortcut for a method that will be used quite often (e.g., `$`, `$$`, `$A`)
- As a sigil in front of a word that would otherwise be reserved in JavaScript (e.g., `$break`, `$super`)

In this case, `super` is one of the "reserved words" that can't be used as an identifier (just like you can't define a function named `if` or `while`). Oddly enough, `super` is not used in JavaScript 1.x, but was preemptively reserved for future use.

By adding a dollar sign to the beginning of the word, Prototype circumvents this edict.

Because we still have access to the original method, `Quarterback#initialize` doesn't have to duplicate code that's already been written for `Player#initialize`. We can tell it to call `Player#initialize`, and give it more instructions afterward. Specifically, `Quarterback#initialize` sets the starting values for `passingYards` and `rushingYards`—statistics that aren't common to all players, and thus need to be defined in subclasses of `Player`.

We can test this code out in a Firebug console:

```
var andrew = new Quarterback('Andrew', 'Dupont');

andrew.passingYards; //-> 0
andrew.points;       //-> 0
andrew.throwPass(23);
>> "Andrew Dupont throws for 23yds."

andrew.passingYards; //-> 23

andrew.throwTouchdown(39);
>> "Andrew Dupont throws for 39yds."
>> "TOUCHDOWN!"

andrew.passingYards; //-> 62
andrew.points;       //-> 6
```

Everything works as expected. So let's try another position. A *wide receiver* plays on offense and catches passes thrown by the quarterback.

```
var WideReceiver = Class.create(Player, {
  initialize: function(firstName, lastName) {
    // call Player's initialize method
    $super(firstName, lastName);

    // define properties for receivers
    this.receivingYards = 0;
  },

  catchPass: function(yards) {
    console.log(this + ' catches a pass for ' + yards + 'yds');
    this.receivingYards += yards;
  },

  catchTouchdown: function(yards) {
    this.catchPass(yards);
    console.log('TOUCHDOWN!');
    this.scorePoints(6);
  }
});
```

Notice again that we're not writing copy-and-paste code. Our `WideReceiver` class defines only those methods and properties that are unique to wide receivers, deferring to the `Player` class for everything else.

Monkeypatching

Most OOP-like languages treat classes as *static*—once they're defined, they're immutable. In JavaScript, however, nothing is immutable, and it would be silly to pretend otherwise. Instead, Prototype borrows from Ruby once again.

In Ruby, all classes are mutable and can be changed at any point. This practice is referred to as "monkeypatching" by those who deride it; I'll refer to it that way simply because I like words that contain *monkey*. But there's no negative connotation for me.

Each class object comes with a method, `addMethods`, that lets us add instance methods to the class later on:

```
Player.addMethods({
  makeActive: function() {
    this.isActive = true;
    console.log(this + " will be a starter for Sunday's game.");
  },

  makeReserve: function() {
    this.isActive = false;
    console.log(this + " will spend Sunday on the bench.");
  }
});
```

So now we've got two new methods on the `Player` class for managing team lineups. But these methods also propagate to `Player`'s two *subclasses*: `Quarterback` and `WideReceiver`. Now we can use `makeActive` and `makeReserve` on all instances of these classes—*even the instances we've already created*. Remember the narcissistic instance of `Quarterback` I created?

```
andrew.makeReserve();
>> "Andrew Dupont will spend Sunday on the bench."
andrew.isActive; //-> false
```

Don't take this freedom as license to code in a style that is stupid and pointless. Most of the time you won't need to monkeypatch *your own* code. But `Class#addMethods` is quite useful when dealing with code that isn't yours—a script.aculo.us class, for example, or another class defined by a Prototype add-on.

Usage: DOM Behavior Pattern

Prototype's advanced OOP model is the perfect tonic for the headache of managing a complex behavior layer. For lack of a better term, I'm going to refer to this as the *behavior* *pattern*—a class describes some abstract behaviors, and instances of that class apply the behavior to individual elements.

I'll rephrase that in plain English. In Chapter 6, we wrote some JavaScript to manage computing totals in a data table: given a table of Texas cities alongside their populations, our code added together all the population numbers, and then inserted the total in a new row at the bottom of the table.

The function we wrote got the job done, but was written with specifics in mind. In the interest of reusing code, let's refactor this function. We'll make the logic more generic and place it into a class that follows the behavior pattern.

```
function computeTotalForTable(table) {
  // Accept a DOM node or a string.
  table = $(table);

  // Grab all the cells with population numbers in them.
  var populationCells = table.select('td.number');

  // Add the rows together.
  // (Remember the Enumerable methods?)
  var totalPopulation = populationCells.inject(0, function(memo, cell) {
    var total = cell.innerHTML;
    // To add, we'll need to convert the string to a number.
    return memo + Number(total);
  });

  // We've got the total, so let's build a row to put it in.
  var tr = new Element('tr', { 'class': 'total' });

  tr.insert( new Element('th').update('Total') );
  tr.insert( new Element('td',
   { 'class': 'number' } ).update(totalPopulation) );

  // Insert a cell for the airport code, but leave it empty.
  tr.insert( new Element('td', { 'class': 'code' }) );

  table.down('tbody').insert(tr);
}
```

What can we improve upon? How can we make this code more generic and versatile?

- We should not make assumptions about the layout of the table, nor the order of its columns, nor the way numeric cells are marked. The function assumes it ought to add all cells with a class of `number`, but what if that selector is too simplistic for what we need?

- To display the total, we build a whole table row in JavaScript, complete with cells, and then insert it at the bottom of the table. This approach isn't very DRY. What happens when we add a column to this table in the HTML? There's a good chance we'll forget to make the same change in the JavaScript.

Above all, we should rewrite this code to be more lightweight. Simple things are reusable; complex things are often context specific. We can add more complexity later if need be, but the best foundation is a simple one.

Refactoring

What are the simplest possible solutions to the preceding issues?

- We should be able to specify a CSS selector that describes which elements (within the context of a certain container) we want totaled. It can have a default (like `.number`) for convenience.

- Instead of building our own DOM structure to place the total into, let's take our cue from the HTML. In other words, the class should accept an *existing* element on the page for displaying the total. This limits the responsibility of the class, simplifying the code.

So let's write a class called `Totaler`, starting off with the bare skeleton:

```
var Totaler = Class.create({
  initialize: function() {

  }
});
```

How many arguments should it take? We *need* at least two things: the context element and the "total container" element. Any other parameters can fall back to intelligent defaults, so we'll place them in an `options` argument at the end.

```
var Totaler = Class.create({
  initialize: function(element, totalElement, options) {
    this.element = $(element);
    this.totalElement = $(totalElement);
  }
});
```

So what are these "options" I speak of? First of all, we should be able to tell Totaler which elements to pull numbers from. So one of them we'll call selector, and by default it will have a value of ".number".

```
var Totaler = Class.create({
  initialize: function(element, totalElement, options) {
    this.element = $(element);
    this.totalElement = $(totalElement);
    this.options = {
      selector: ".number"
    };
  }
});
```

Now we've got a default value for selector, but we also want the user to be able to override this. So let's copy the options argument over this.options, letting all user-specified options trump the defaults:

```
var Totaler = Class.create({
  initialize: function(element, totalElement, options) {
    this.element = $(element);
    this.totalElement = $(totalElement);
    this.options = Object.extend({
      selector: ".number"
    }, options || {});
  }
});
```

We type options || {} because the user is allowed to omit the options argument entirely: it's akin to saying, "Extend this.options with options *or*, failing that, an empty object."

Remember that this refers to the class instance itself. So we've defined three properties on the instance, corresponding to the three arguments in our constructor. These properties will be attached to each instance of the Totaler class as it gets instantiated. But to do the actual adding, we'll write another method:

```
var Totaler = Class.create({
  initialize: function(element, totalElement, options) {
    this.element = $(element);
    this.totalElement = $(totalElement);
    this.options = Object.extend({
      selector: ".number"
    }, options || {});
  },

  updateTotal: function() {

  }
});
```

`Totaler#updateTotal` will select the proper elements, extract a number out of each, and then add them all together, much the same way as before. It needn't take any arguments; all the information it needs is already stored within the class.

First, it selects the elements by calling `Element#select` in the context of the container element.

```
var Totaler = Class.create({
  initialize: function(element, totalElement, options) {
    this.element = $(element);
    this.totalElement = $(totalElement);
    this.options = Object.extend({
      selector: ".number"
    }, options || {});
  },

  updateTotal: function() {
    var numberElements = this.element.select(this.options.selector);
  }
});
```

`Totaler#updateTotal` uses the selector we assigned in the constructor; anything set as a property of `this` can be read both inside and outside the class. It selects all elements *within* `element` (the container) that match the given selector.

As before, we use `Enumerable#inject` to add the numbers together:

```
var Totaler = Class.create({
  initialize: function(element, totalElement, options) {
    this.element = $(element);
    this.totalElement = $(totalElement);
    this.options = Object.extend({
      selector: ".number"
    }, options || {});
  },

  updateTotal: function() {
    var numberElements = this.element.select(this.options.selector);
    var total = numberElements.inject(0, function(memo, el) {
      return memo + Number(el.innerHTML);
    });
  }
});
```

Finally, we fill in `totalElement` with the sum using `Element#update`. And we also modify `initialize` so that it calls `updateTotal`—so that the sum is computed automatically when `Totaler` is instantiated.

```
var Totaler = Class.create({
  initialize: function(element, totalElement, options) {
    this.element = $(element);
    this.totalElement = $(totalElement);
    this.options = Object.extend({
      selector: ".number"
    }, options || {});

    this.updateTotal();
  },

  updateTotal: function() {
    var numberElements = this.element.select(this.options.selector);
    var total = numberElements.inject(0, function(memo, el) {
      return memo + Number(el.innerHTML);
    });
    this.totalElement.update(total);
  }
});
```

We're done writing the `Totaler` class, at least for now. Save it as `totaler.js`.

Testing

Now find the `index.html` file you created for Chapter 6. Copy it over to your Chapter 7 folder. We'll be using the same data for this example, but we'll need to change the table's markup to fit with the changes we've made. Alter the table until it looks like this:

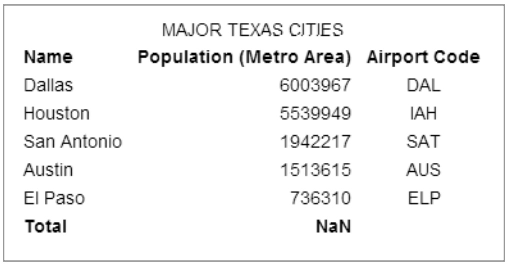
```
<table id="cities">

  <caption>Major Texas Cities</caption>
  <thead>
    <tr>
      <th scope="col">Name</th>
      <th scope="col" class="number">Population (Metro Area)</th>
      <th scope="col">Airport Code</th>
    </tr>
  </thead>
  <tfoot>
    <tr>
      <th scope="row">Total</th>
      <td class="number" id="population_total"></td>
      <td></td>
    </tr>
  </tfoot>
  <tbody>
    <tr id="dallas">
      <th scope="row">Dallas</th>
      <td class="number">6003967</td>
      <td class="code">DAL</td>
    </tr>
    <tr id="houston">
      <th scope="row">Houston</th>
      <td class="number">5539949</td>
      <td class="code">IAH</td>
    </tr>
    <tr id="san_antonio">
      <th scope="row">San Antonio</th>
      <td class="number">1942217</td>
      <td class="code">SAT</td>
    </tr>
    <tr id="austin">
      <th scope="row">Austin</th>
      <td class="number">1513615</td>
      <td class="code">AUS</td>
    </tr>
    <tr id="el_paso">
      <th scope="row">El Paso</th>
      <td class="number">736310</td>
      <td class="code">ELP</td>
    </tr>
  </tbody>
</table>
```

We've inserted a table footer to hold the row that displays the total. Inside that footer, one cell has been marked with an ID and is blank; that's the cell that will hold our total. (Note that the tfoot element appears *above* the tbody in the markup, but is rendered *below* the tbody on the screen. This comes straight from the HTML spec.)

Now we need to include totaler.js on this page and instantiate it on page load. Delete the script element that holds our code from the last chapter. Here's how your scripts should look instead:

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

<script type="text/javascript">
  document.observe("dom:loaded", function() {
    window.totaler = new Totaler('cities', 'population_total');
  });
</script>
```

The first highlighted line is a reference to our Totaler class, of course. The second, run when the DOM is ready, declares a new instance of Totaler on our population table. Figure 7-1 shows what happens.



MAJOR TEXAS CITIES		
Name	**Population (Metro Area)**	**Airport Code**
Dallas	6003967	DAL
Houston	5539949	IAH
San Antonio	1942217	SAT
Austin	1513615	AUS
El Paso	736310	ELP
Total	**NaN**	

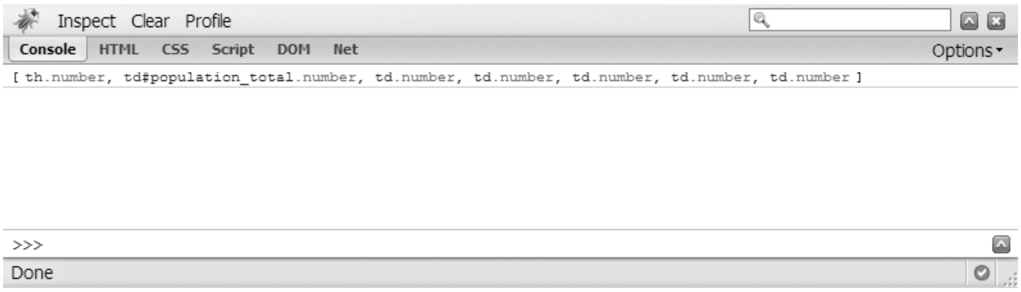
Figure 7-1. *Instead of the total we were expecting, we get NaN.*

Ugh, `NaN` (which stands for "not a number"). That means we tried to make a number out of something that has no numeric value (for instance, `Number("qwerty");`). What did we do wrong?

Let's do some logging to find out. Open up `totaler.js` and add a log statement to `Totaler#updateTotal` so that you can see which elements we're collecting:

```
updateTotal: function() {
  var numberElements = this.element.select(this.options.selector);
  console.log(numberElements);
  var total = numberElements.inject(0, function(memo, el) {
    return memo + Number(el.innerHTML);
  });
  this.totalElement.update(total);
},
```

Save, reload, and look at the Firebug console to see the problem: too many cells are being included in the sum (see Figure 7-2).



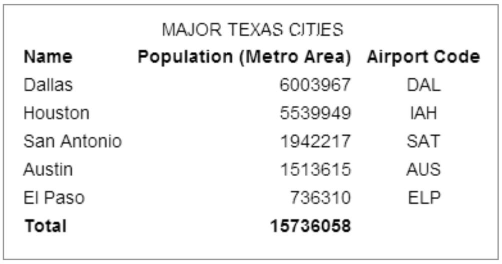
```
[ th.number, td#population_total.number, td.number, td.number, td.number, td.number, td.number ]
>>>
```

Figure 7-2. *Firebug shows us which elements we've logged to the console.*

The header cell and footer cell for that column bear a class of `"number"` so that they get the proper styling (they're aligned to the right of the cell so that all the numbers line up). But our default selector (`.number`) is retrieving them along with the cells in the table body. So let's override the default and give a more specific selector:

```
<script type="text/javascript">
  document.observe("dom:loaded", function() {
    window.totaler = new Totaler('cities', 'population_total', {
      selector: "tbody .number"
    });
  });
</script>
```

Since the total cell is a descendant of the `tfoot`, not the `tbody`, this selector will exclude it. To prove it, save the file and reload the page in your browser (see Figure 7-3).



MAJOR TEXAS CITIES

Name	Population (Metro Area)	Airport Code
Dallas	6003967	DAL
Houston	5539949	IAH
San Antonio	1942217	SAT
Austin	1513615	AUS
El Paso	736310	ELP
Total	**15736058**	

Figure 7-3. *The sum is displayed in the table footer.*

Aha! Because the total cell now shows the sum of all the number cells in the `tbody`, we know the instance of `Totaler` was initialized correctly. Remove the `console.log` statement we added; the debugging is complete.

Since we assigned the `Totaler` instance to `window.totaler` (i.e., a global variable named `totaler`), we can use the Firebug shell to examine the instance:

```
>>> totaler
Object element=table#cities
```

Click that line to inspect the object. You'll see a listing of properties that includes the methods we've defined (`initialize` and `updateTotal`) and the instance properties we've set (`element`, `totalElement`, and `options`).

This means that, for instance, we can call `updateTotal` again if the numbers in our cells change. Try this:

```
>>> totaler.element.down("tbody .number").update(0);
<td class="number">
>>> totaler.updateTotal();
```

To simulate what would happen if every resident of the greater Dallas area simply decided to leave, we drill down to the first number cell in our `tbody` and change the value to `0`. Then, to recompute the total, we call the `updateTotal` method on `totaler`. The total cell changes to reflect the new sum.

So we know that whenever the value in one of those cells changes, we can manually call `updateTotal`. The next step, of course, would be to make that call automatic—to tell the class to *watch* these cells for changes in value and update the total cell accordingly. In Chapter 8, you'll learn how to do this.

Reuse

Go back to `totaler.js` for a moment so we can reflect on what we've just done. We started with a context-specific function and turned it into a general class that can easily be reused. The original function made broad assumptions about the naming of elements, the output, and even the structure of the markup itself. The `Totaler` class, on the other hand, can be applied wherever we need to extract a sum from several different places in the markup.

```
// HTML:
<p id="apple_anecdote">
  I bought <span class=”number”>11</span> apples at the grocery store.
  On the way home I bought <span class=”number”>7</span> more from the
  back of a truck parked alongside the highway. When I got home I found
  I already had <span class=”number”>6</span> apples in the pantry. So
  now I've got <span id=”apple_total”></span> apples.
</p>

// JavaScript:
new Totaler('apple_anecdote', 'apple_total');
```

Moreover, we can declare as many `Totalers` on one page as necessary. Each one operates in blissful ignorance of whatever happens outside of the element it's assigned to.

For these reasons, the DOM behavior pattern is used for all the script.aculo.us widgets. And we'll use this pattern for all the widgets we build as well. More on this in the chapters to come.

Functional Programming

You first learned about functional programming in Chapter 2. We know that functions are "first-class objects" in JavaScript—`function` is a data type, just like `string` or `object`, so we can treat it the same way we treat other data types.

This means, for instance, that functions can be passed as arguments to other functions. We used this technique to great effect in Chapter 3 where, with the help of `Enumerable` methods, we applied a function to each item of a collection.

The following sections describe some other things functions can do.

Functions Can Have Their Own Methods

Imagine I've got a function that adds numbers together:

```
function sum() {
  var num = 0;
  for (var i = 0; i < arguments.length; i++)
    num += arguments[i];

  return num;
}
```

The `sum` function will add its arguments together and return a number. If no arguments are given, it will return `0`.

We know the syntax for calling this method:

```
var result = sum();
```

To call this function, we append empty parentheses. If we take the parentheses off, though, we're no longer calling the function—we're just *referring* to it:

```
var result = sum;
```

Be sure you understand the difference. In the first example, `result` is set to `0`—the result of calling `sum` with no arguments. In the second example, `result` is set to the `sum` function *itself*. In effect, we're giving `sum` an alias.

```
var result = sum;
result(2, 3, 4); //-> 9
```

We're going to look at a few instance methods of `Function`. At first, these may seem like magic tricks, but they'll become more intuitive the more you use them.

Using Function#curry

Partial application (or *currying*) is a useful technique in languages where functions are first-class objects. It's the process by which you "preload" a number of arguments into a function. In other words, I could "curry" a function that expects parameters `a`, `b`, and `c` by giving it `a` and `b` ahead of time—getting back a function that expects only `c`.

Confused yet? What I just said is much easier to express in code:

```
function alertThreeThings(a, b, c) {
  alert(a);
  alert(b);
  alert(c);
}

var alertTwoThings = alertThreeThings.curry("alerted first");
alertTwoThings("foo", "bar");
// alerts "alerted first"
// alerts "foo"
// alerts "bar"
```

We've just defined a function that, *if we were to write it out ourselves*, would behave like this:

```
function alertTwoThings(b, c) {
  return alertThreeThings("alerted first", b, c);
}
```

But by using `curry`, we're able to express this more concisely and flexibly.

`Function#curry` can accept any number of arguments. I could load two or three arguments into `alertThreeThings`:

```
var alertOneThing = alertThreeThings.curry("alerted first",
 "alerted second");
alertOneThing("foo");
// alerts "alerted first"
// alerts "alerted second"
// alerts "foo"

var alertZeroThings = alertThreeThings.curry("alerted first",
 "alerted second", "alerted third");
alertZeroThings();
// alerts "alerted first"
// alerts "alerted second"
// alerts "alerted third"
```

Let's look at a less-contrived example. We can curry the `sum` function we defined earlier:

```
var sumPlusTen = sum.curry(10);
typeof sumPlusTen; //-> "function"

sumPlusTen(5);       //-> 15
sumPlusTen();        //-> 10
sumPlusTen(2, 3, 4); //-> 19
```

Each of these examples is equivalent to calling `sum` with a value of `10` as the first argument. It doesn't matter how many arguments we pass to the curried function, since we've defined `sum` in a way that works with any number of arguments. The curried function will simply add `10` to the beginning of the argument list and pass those arguments to the original function.

We'll explore the use cases for `Function#curry` in later chapters. But it's important for you to be familiar with it in the context of the other `Function` instance methods we're about to cover.

Using Function#delay and Function#defer

JavaScript has no formal "sleep" statement—no way to block all script execution for a specified amount of time. But it does have `setTimeout`, a function that schedules code to be run at a certain time in the future.

```
function remind(message) {
  alert("REMINDER:" + message);
}
setTimeout(function() { remind("Be sure to do that thing"); }, 1000);
```

The built-in `setTimeout` function takes two arguments. The first can be either a function or a string; if it's a string, it's assumed to be code, and will be evaluated (with `eval`) at the proper time. (It's much better practice to pass a function.) The second argument must be a number, in milliseconds, that tells the interpreter how long to wait before trying to run the code we gave it. It returns an integer that represents the timer's internal ID; if we want to unset the timer, we can pass that ID into the `clearTimeout` function.

In this example, the `setTimeout` call ensures that the function we've passed it will get called *at least* 1000 ms (1 second) later. Because browser JavaScript executes in a single-threaded context, the interpreter might be *busy* 1 second later, so there's no way of knowing the exact moment. But nothing will slip through: as soon as the interpreter is idle again, it will look at the backlog of things that have been scheduled, running anything that's past due.

So maybe that's a better way to think of `setTimeout`: it adds functions to a queue, along with a "do not run before" sticker that bears a timestamp. In the preceding example, the timestamp is computed by adding 1000 ms to whatever the time was when we called `setTimeout`.

As a thought experiment, then, imagine the second argument to `setTimeout` getting smaller and smaller. What happens when it hits `0`? Does such a thing trigger a wormhole through space-time? No, for our purposes, the result is far more mundane. The interpreter "defers" the function to run whenever it has a spare moment.

Prototype thinks the `setTimeout` function is ugly. Prototype prefers to give functions instance methods named `delay` and `defer`.

Function#delay

`Function#delay` expects one argument: a number that specifies the number of *seconds* to wait (*not* milliseconds, unlike `setTimeout`).

```
function annoy() {
  alert("HEY! You were supposed to do that THING!");
}
annoy.delay(5);
// alerts "HEY! You were supposed to do that THING!" roughly 5 seconds later
```

There are those who might say that this is a just a fancy way of calling `setTimeout`. I think it's a *less* fancy way of calling `setTimeout`, if fanciness can be measured in number of characters typed.

```
// equivalent statements
setTimeout(annoy, 5000);
annoy.delay(5);
```

The gains are more obvious when you're calling a function that takes arguments. Any arguments given to `delay` after the first are passed along to the function itself:

```
function remind(message) {
  alert("REMINDER:" + message);
}

remind.delay(5, "Be sure to do that thing.");
// alerts "REMINDER: Be sure to do that thing." roughly 5 seconds later
```

Now compare the several ways to say the same thing:

```
// equivalent statements
setTimeout(function() { remind("Be sure to do that thing." }, 5000);
setTimeout(remind.curry("Be sure to do that thing."), 5000);
remind.delay(5, "Be sure to do that thing.");
```

We save a few keystrokes through clever use of `Function#curry`, but we save far more by using `Function#delay`.

Function#defer

`Function#defer` is equal to `Function#delay` with a timeout of `0.01` seconds (the smallest practical timeout in a browser environment). To defer a function call is to say, "Don't do this now, but do it as soon as you're not busy."

To illustrate this concept, let's see how `defer` affects execution order:

```
function remind(message) {
  alert("REMINDER:" + message);
}

function twoReminders() {
  remind.defer("Don't forget about this less important thing.");
  remind("Don't forget about this _absolutely critical_ thing!");
}
twoReminders();
// alerts "Don't forget about this _absolutely critical_ thing!"
// alerts "Don't forget about this less important thing."
```

In the `twoReminders` function, we make two calls to `remind`. The first, a deferred call, fires *after* the second. More specifically, the second call fires immediately, and the first call fires as soon as the interpreter exits the `twoReminders` function.

The commonest use case for `defer` is to postpone costly operations:

```
function respondToUserClick() {
  doSomethingCostly.defer(); // instead of doSomethingCostly();
  $('foo').addClassName('selected');
}
```

Here, it's best to defer the call to `doSomethingCostly` until after the function exits. That way, the visual feedback (adding a class name to an element) happens without delay, making the application feel "snappier" to the user.

Using Function#bind

To talk about Prototype's `Function#bind` is to delve into JavaScript's confusing rules about scope. Scope is the meaning of the keyword `this` in a given context.

Let's look back at our `Totaler` example. In Chapter 1, we talked about how a scope is created whenever an object is instantiated. When I call `new Totaler`, I create a new scope for the instance; `Totaler#initialize` runs within this scope, as do all other methods in the `Totaler` class. Therefore, the keyword `this` refers to the instance itself when we're inside any of these methods.

```
var Totaler = Class.create({
  initialize: function(element, totalElement, options) {
    this.element = $(element);
    this.totalElement = $(totalElement);
    this.options = Object.extend({
      selector: ".number"
    }, options || {});
  },
...
```

For this reason, `this` becomes internal shorthand for the instance itself.

Try to take one of these methods out of context, though, and you'll run into problems. If I declare a new `Totaler` on the page, I might want to alias its `updateTotal` method for convenience:

```
window.totaler = new Totaler('cities', 'population_total');
var retotal = totaler.updateTotal;
```

All I've done is hand a method reference to `retotal`. I should be able to call `retotal` on its own later on, but if I try it I run into problems:

```
retotal();
//-> Error: this.element is undefined
```

In JavaScript, scope is bound to execution context. The interpreter doesn't decide what `this` means until a function gets called. Here we run into trouble—`retotal` doesn't know anything about the `Totaler` instance's scope.

`Function#bind` solves this problem. It expects one argument and returns a version of the function whose scope is "bound" to that argument. Here, we want `this` to refer to `totaler`, so let's bind it to that scope to make the function truly portable:

```
var retotal = totaler.updateTotal.bind(totaler);
retotal();
```

The method runs, error free.

`Function#bind` is most useful when dealing with event assignment. Switching the context in which an event handler runs is something we'll need to do quite often in the chapters to come.

Summary

We've taken a brief glance at two useful models for DOM scripting: functional and object-oriented programming. These models, in fact, go far beyond contrived code patterns; they're manifestations of core features of JavaScript. Functional programming is right at home in the event-driven world of browser scripting; OOP is a corollary of JavaScript's principles of mutability and scope.

You'll be able to appreciate these pillars of JavaScript coding philosophy as we delve into the use cases presented in Part 2.

CHAPTER 8

Other Helpful Things: Useful Methods on Built-Ins

As embarrassing as it is for me to have a chapter devoted to "other random stuff," I've decided to write it anyway. This book isn't meant to teach you JavaScript; it's meant to be a survey of a framework that acts as JavaScript's "standard library." Prototype sticks utility methods in appropriate nooks and crannies, some of which are simply too general to have been addressed in an earlier chapter.

This chapter, then, will explore the convenience methods that Prototype adds to built-in objects. Many of them are used within Prototype itself, but they're likely to be useful in your own code as well.

Using String Methods

I'm at a loss here. What can I say about strings? Strings in JavaScript bear good news and bad news. The bad news is that many of the conveniences that other languages possess for dealing with strings simply aren't present in JavaScript. The good news is that, as we've done elsewhere, we can leverage the hackability of the language to *fix* this shortcoming.

String Utility Methods

Prototype adds a bagful of useful instance methods to strings. Some you'll use every day; some rarely, if ever. But they're there in case you need them.

The gsub, sub, and scan Methods

These first three methods all involve searching for text in a string and doing something with the result.

At first glance, `String#gsub` would appear to be redundant—it behaves just like `String#replace` and has a weirder name. At second glance, though, it will become your method of choice for string substitution.

Short for "global substitution," `gsub` is named after the similar method in Ruby. Like `replace`, it takes two arguments: a pattern and a replacement. The pattern can be a string or a regular expression; the replacement can be a string or a function.

Let's look at the simplest case—both arguments as strings:

```
"Never, never pour salt in your eyes.".gsub('never', 'always');
//-> "Never, always pour salt in your eyes."
```

Wait—that's not what we meant. We want to replace both `never` and `Never`, so let's change that first argument to a case-insensitive regular expression.

■ **Tip** Do regular expressions intimidate you? If so, this section might not be for you. Type "regular expressions" into your favorite search engine if you need a crash course.

```
"Never, never pour salt in your eyes.".gsub(/never/i, 'always');
//-> "always, always pour salt in your eyes."
```

OK, that problem was easy to solve—JavaScript allows us to ignore case by using the `i` flag at the end of a regular expression. But now we've got a new problem. The first "never" has a capital *N*, since it's the first word of the sentence. We need to ensure that a capitalized word has a capitalized replacement.

To do this, let's get a little cleverer with our regular expression. We can experiment with JavaScript's `RegExp#exec` until we have one that suits our needs better. `RegExp#exec` accepts a string, applies the given regular expression against it, and returns an array of matches:

```
/never/.exec("Never, never pour salt in your eyes.");
//-> ["never"]

/never/i.exec("Never, never pour salt in your eyes.");
//-> ["Never"]
```

You'll notice the matches themselves are arrays, too. The first item in this array is the full match. If there are any *captures* in the expression—indicated by parentheses—then those submatches are also given in the order they occur. Since we're trying to figure out if "never" is capitalized or not, let's capture the first letter of the pattern:

```
/(n)ever/i.exec("Never, never pour salt in your eyes.");
//-> ["Never", "N"]
```

Armed with this insight, let's go back to `gsub` and swap out the second argument. We'll turn it into a function—one that decides on a proper replacement string for each match.

```
"Never, never pour salt in your eyes.".gsub(/(n)ever/i, function(match) {
  if (match[1] === match[1].toUpperCase()) return "Always";
  else return "always";
});
//-> "Always, always pour salt in your eyes."
```

Notice that this function takes one argument, `match`, which corresponds to each match of the pattern as would be returned by `String#match`.

Two other methods, `sub` and `scan`, work in a very similar way. `String#sub` replaces *only the first match* of a given pattern in a string:

```
"Never, never pour salt in your eyes.".sub(/never/i, 'Always');
//-> "Always, never pour salt in your eyes."
```

And `String#scan` is used for executing a function against each match of a pattern:

```
// find all four-letter words in a phrase
var warning = "Never, never pour salt in your eyes.", fourLetterWords = [];
warning.scan(/\b\w{4}\b/, function(match) { fourLetterWords.push(match[0]); });
console.log("Four-letter words: " + fourLetterWords.join(', ') );
//-> "Four-letter words: pour, salt, your, eyes"
```

To review, all three of these methods let you search for a pattern in a string. Therefore, all three expect either a string or a regular expression as the first argument. Two of these methods, `gsub` and `sub`, replace one substring with another—`gsub` acts on every occurrence, while `sub` acts on only the first. So both can take either a string or a function as the replacement. Finally, `scan` doesn't do any replacement at all; it just calls a function for every occurrence of a pattern.

The strip Method

Sounds scandalous, I know, but it's pretty mundane. `String#strip` simply removes all leading and trailing spaces from a string:

```
"   foo       ".strip(); //-> "foo"
```

In the unpredictable browser environment, where whitespace fills every crack and crevice, `strip` helps normalize strings for comparison:

```
var a = "bar  ", b = " bar";
a == b; //-> false
a.strip() == b.strip(); //-> true
```

The stripTags, escapeHTML, and unescapeHTML Methods

It's frustrating to deal with HTML in string form, but it's often necessary. Many Prototype methods, like `Element#insert` and `Element#update`, accept HTML strings as one way to place content into a page.

It's also important to write code that's both defensive and secure. Let's look at an example:

```
<form id="blog_comment" action="/path/to/action/page">
  <p>
    <label for="comment_name">Name </label><br />
    <input id="comment_name" name="comment_name" type="text" />
  </p>
  <p>
    <label for="comment_text">Comment</label><br />
    <textarea id="comment_text" name="comment_text"></textarea>
  </p>
  </form>

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

Assume that this is a standard blog comment form. We want to let the commenter preview her comment before submitting, so we'll set a listener that will update the `div` with each keystroke:

```
function updateLivePreview() {
  var commentText = $('comment_text').value;
  $('live_preview').update(commentText);
}

Event.observe(window, 'load', function() {
  $('comment_text').observe('keyup', updateLivePreview);
});
```

Load this example in a browser, and you'll see that this code behaves the way we expect. We can type something into the `textarea` and see a live version in the `div` below (Figure 8-1).

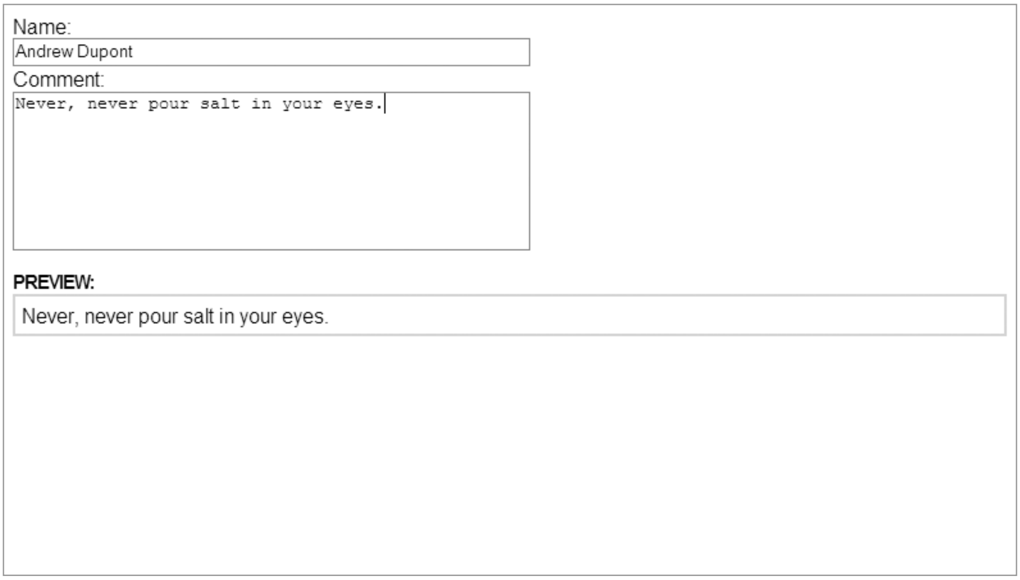
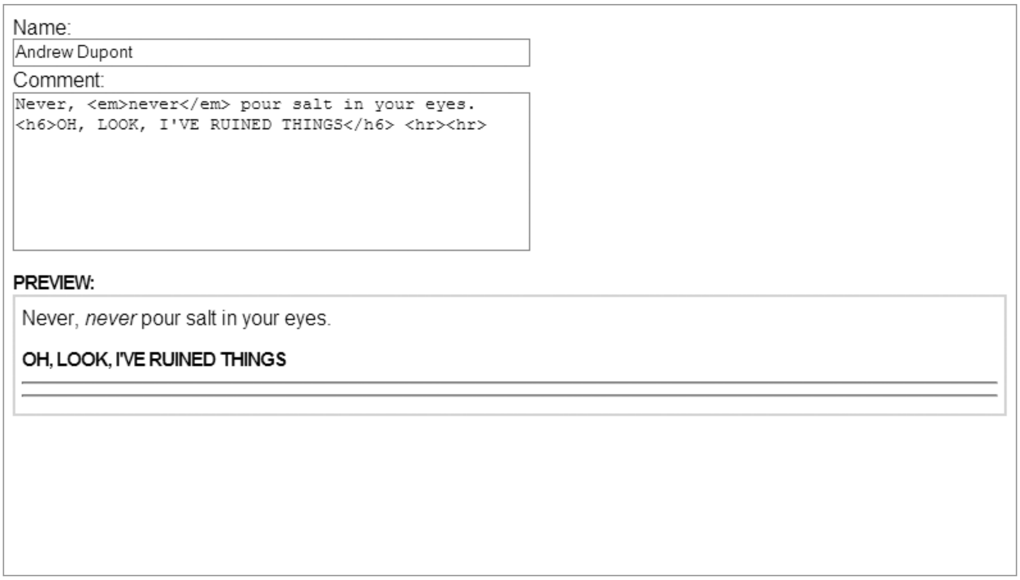

Name:
Andrew Dupont
Comment:
Never, never pour salt in your eyes.
PREVIEW:
Never, never pour salt in your eyes.

Figure 8-1. *Live comment preview*

But it's not enough to test typical input. We've also got to test how resilient it is by feeding it *unexpected* input. Sure enough, this code handles plain text deftly, but doesn't like HTML all that much (see Figure 8-2).


Name:
Andrew Dupont
Comment:

```
Never, <em>never</em> pour salt in your eyes.
<h6>OH, LOOK, I'VE RUINED THINGS</h6> <hr><hr>
```

PREVIEW:

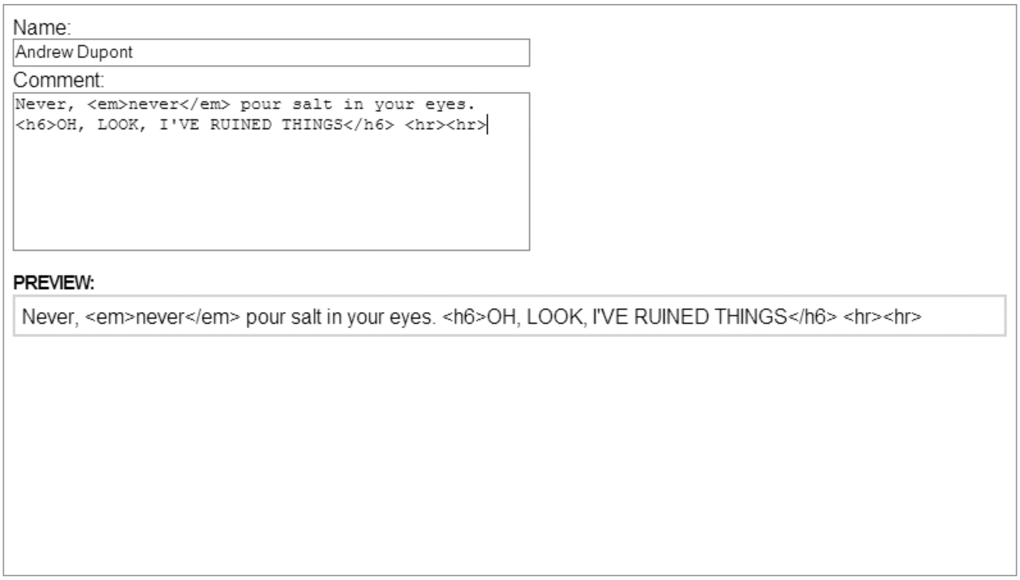
Never, *never* pour salt in your eyes.

OH, LOOK, I'VE RUINED THINGS

Figure 8-2. *The code doesn't sanitize user input.*

From a server-side perspective, failing to sanitize user input is a huge security risk. From a client-side perspective, it's also a monkey wrench in your page design. If we're not allowing HTML in blog comments, then we've got to *escape* the input so that every character is treated as a literal value (see Figure 8-3):

```
function updateLivePreview() {
  var commentText = $('comment_text').value.escapeHTML();
  $('live_preview').update(commentText);
}
```



Andrew Dupont

```
Never, <em>never</em> pour salt in your eyes.
<h6>OH, LOOK, I'VE RUINED THINGS</h6> <hr><hr>
```

Never, <em>never</em> pour salt in your eyes. <h6>OH, LOOK, I'VE RUINED THINGS</h6> <hr><hr>

Figure 8-3. *Comment preview with escaped HTML*

`String#escapeHTML` finds characters with special meaning in HTML—specifically angle brackets (`<>`)—and converts them to their HTML entity equivalents:

```
"Never, <em>never</em> pour salt in your eyes.".escapeHTML();
//-> "Never, &lt;em&gt;never&lt;/em&gt; pour salt in your eyes."
```

As you might expect, `String#unescapeHTML` does the exact opposite:

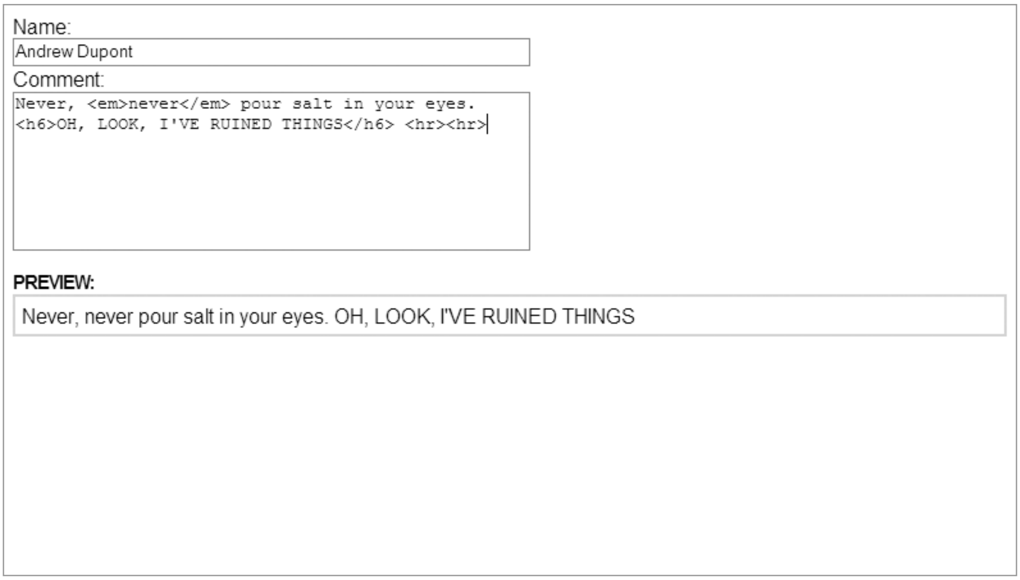
```
"Never, &lt;em&gt;never&lt;/em&gt; pour salt in your eyes.".unescapeHTML();
//-> "Never, <em>never</em> pour salt in your eyes."
```

So that's one approach we can take. Another would be to *ignore* anything that looks like HTML, rather than display it literally:

```
"Never, <em>never</em> pour salt in your eyes.".stripTags();
//-> "Never, never pour salt in your eyes."
```

In fact, this solution better captures the commenter's original intent. `String#stripTags` removes all HTML from a string, leaving behind only plain text (see Figure 8-4).

■ **Caution** Never, *ever* use client-side sanitization as a replacement for server-side sanitization. Client-side sanitization is trivial to bypass and gives you a false sense of security. Instead, decide how the server will handle unusual input, and then write client-side code to mirror that behavior. Live comment preview is a UI feature—*not* a security measure.


Name:
Andrew Dupont
Comment:

```
Never, <em>never</em> pour salt in your eyes.
<h6>OH, LOOK, I'VE RUINED THINGS</h6> <hr><hr>
```

PREVIEW:
Never, never pour salt in your eyes. OH, LOOK, I'VE RUINED THINGS

Figure 8-4. *HTML no longer has an effect on the page.*

The camelize, underscore, and dasherize Methods

These string-formatting methods spring from identical methods in the popular Rails framework. They're used to convert between different methods of word delimiting. I'll let the code explain itself:

```
"lorem-ipsum-dolor".underscore(); //-> "lorem_ipsum_dolor"
"lorem_ipsum_dolor".dasherize();  //-> "lorem-ipsum-dolor"
"lorem-ipsum-dolor".camelize();   //-> "loremIpsumDolor"
```

You can see that `underscore` converts hyphens to underscores, `dasherize` converts underscores to hyphens (yes, *hyphens*, even though the method calls them dashes!), and `camelize` removes hyphens while capitalizing the letters that follow (otherwise known as camel case).

For instance, Prototype allows you to call `Element#setStyle` and pass CSS property names in *either* their hyphenated *or* their camelized variants:

```
$('foo').setStyle({ paddingBottom: '10px' });
$('foo').setStyle({ 'padding-bottom': '10px' });
```

`Element#setStyle` makes this possible by calling `String#camelize` to ensure that all property names are in camel case.

The capitalize and truncate Methods

These methods format strings so that they're fit for user consumption. `String#capitalize` will convert the first letter of a string to uppercase and *all other letters* to lowercase:

```
"never".capitalize(); //-> "Never";
"NEVER".capitalize(); //-> "Never";
"Never".capitalize(); //-> "Never";
```

`String#truncate` is quite interesting. It will return the first *n* characters of a string, along with an ellipsis to indicate the truncation:

```
var warning = "Never, never pour salt in your eyes."
var truncation = warning.truncate(15);  //-> "Never, never..."
truncation.length; //-> 15
```

The first argument, naturally, indicates how long you'd like the resulting string to be. The optional second argument lets you specify a custom truncation, if you're not a fan of the default (`...`). Keep in mind that the length of the truncation *is included* in the length of the returned string.

```
var otherTruncation = warning.truncate(20, ">>>>>");
//-> "Never, never po>>>>>"
```

The include, startsWith, endsWith, empty, and blank Methods

Last of all are five methods that test the content of strings. All five return a Boolean—`true` or `false`.

`String#include` tests for a simple substring match. `String#startsWith` and `String#endsWith` do the same, but test whether the anchored substring exists at the beginning or the end of the string, respectively:

```
var warning = "Never, never pour salt in your eyes."
warning.include("salt");       //-> true
warning.startsWith("salt");    //-> false
warning.startsWith("N");       //-> true
warning.startsWith("n");       //-> false
warning.endsWith("your eyes.") //-> true
```

All three of these methods are case sensitive.

`String#empty` and `String#blank` take no arguments—they simply test if the string is empty (has a length of 0) or blank (contains only whitespace):

```
"   ".blank(); //-> true
"   ".empty(); //-> false

"".empty();    //-> true
"".blank();    //-> true
```

All empty strings are blank, but not all blank strings are empty.

The Template Class and String Interpolation

Think of your favorite programming language right now. (If your favorite language is JavaScript, you're an anomaly. Think of your second favorite.) The language you're thinking of certainly has some handy way of mixing variables into existing strings.

PHP and Ruby, for instance, give us *variable interpolation*:

```
// PHP:
echo "Touchdown scored by ${position} ${first_name} ${last_name}!";

// Ruby:
puts "Touchdown scored by #{position} #{first_name} #{last_name}!";
```

In both these languages, we can mark certain parts of a string to be replaced by variables. We notify the language's interpreter by using a special pattern in the string.

JavaScript doesn't have variable interpolation, but since when has that stopped us? We can fake it.

Using the Template Class

Prototype gives us a class named `Template`. Let's play around with it.

```
var t = new Template("The quick brown #{first} jumps over the lazy #{second}.");
```

We declare a new instance of `Template` and pass it a string. You should recognize the special syntax we use inside the string—it's identical to Ruby's.

Now we can use this template over and over again, passing different values for interpolation, with `Template#evaluate`:

```
t.evaluate({ first: "fox", second: "dog" });
//-> "The quick brown fox jumps over the lazy dog."
t.evaluate({ first: "yak", second: "emu" });
//-> "The quick brown yak jumps over the lazy emu."
t.evaluate({ first: "tree", second: "human" });
//-> "The quick brown tree jumps over the lazy human."
```

Ignore the increasing improbabilities of these statements. Instead, note that `Template#evaluate` takes one argument, an `Object`, with properties corresponding to the names we used in the original string. Note also that we need only create the template once—but we can use it over and over to generate strings that conform to the same pattern.

Since an array is just a special kind of object with numeric property names, you can define a template that uses numbered replacements:

```
var t = new Template("The quick brown #{0} jumps over the lazy #{1}.");
t.evaluate(["bandicoot", "hyena"]);
//-> "The quick brown bandicoot jumps over the lazy hyena."
```

Of course, sometimes you won't need to interpolate more than once. So there's also `String#interpolate`:

```
"The quick brown #{first} jumps over the lazy #{second}."
 .interpolate({ first: "ocelot", second: "ibex" });
//-> "The quick brown ocelot jumps over the lazy ibex."
```

Advanced Replacement

Recall in the previous chapter how we created classes for football players, where each instance of a class was a specific player. Each had properties like `firstName`, `lastName`, and `points` that held useful metadata about the player.

We also defined a custom `toString` method, taking advantage of the fact that JavaScript uses a method by that name whenever it needs to coerce an object into a string. Our `toString` method returned the first and last names of the player:

```
var qb = new Quarterback("Andrew", "Dupont");
qb; //-> [Object];
qb + " just won the Heisman trophy.";
//-> "Andrew Dupont just won the Heisman trophy."
```

Because concatenating (+) a string to another object involves automatic string coercion, our instance of `Quarterback` knew how to represent itself in that context.

`Template` and `String#interpolate` do something similar. If the object you're interpolating with has a special method called `toTemplateReplacements`, then the result of that method will be used to fill in the template string:

```
var blatantLie = new Template(
 "#{position} #{firstName} #{lastName} just won the Heisman trophy.");
blatantLie.evaluate(qb);
//-> " Andrew Dupont just won the Heisman trophy."

// adding an instance method to the Quarterback class
Class.extend(Quarterback, {
  toTemplateReplacements: function() {
    return {
      position:  "QB",
      firstName: this.firstName,
      lastName:  this.lastName
    };
  }
});

blatantLie.evaluate(qb);
//-> "QB Andrew Dupont just won the Heisman trophy."
```

Here, we've done a before-and-after comparison. The first time we call `Template#evaluate`, the `Quarterback` instance itself is used—its own properties filling in the holes of the template string. That instance has firstName and lastName properties, but it *doesn't* have a position property, so an empty string is used instead.

When we add Quarterback#toTemplateReplacements, though, we're supplying a *substitute* object to use for interpolation. So we're able to specify properties above and beyond those that already exist on the object. In this case, we're defining a position property to go into the evaluated string.

The larger purpose of defining toTemplateReplacements is the same as the purpose as defining toString: it allows you to specify in *one place* how your object will behave in a given context. For user-defined classes, it also promotes encapsulation, one of those hallowed OOP virtues.

It's all part of the theme that was established in Chapter 3: ensuring that objects come with *instructions for use*. Just like we rely on objects that mix in Enumerable to know how to enumerate themselves, here we're relying on objects to know how to represent themselves in a Template context.

Bringing It Back to String#gsub

The versatility of JavaScript objects gives us a bonus way to use Template. Remember that an array is really just an object with numerals as keys, so Template and String#interpolate can be used with arrays as well:

```
var sample = new Template("The quick brown #{0} jumps over the lazy #{1}");
sample.evaluate(["fox", "dog"]);
//-> "The quick brown fox jumps over the lazy dog."
```

Items of an array respond to property lookups just like any other object.

Finally, then, I can share an Easter egg of sorts—template strings like these can be used in String#gsub:

```
var warning = "Never, never pour salt in your eyes."
warning.gsub(/(salt) in your (eyes)/, "#{2} in your #{1}");
//-> "Never, never pour eyes in your salt."
```

You may also recall that the first item in a match array is the *entire string* that matches the pattern; any subsequent items are *substrings* that match specific captures in the regular expression. The preceding example, then, is shorthand for either of these:

```
warning.gsub(/(salt) in your (eyes)/, function(match) {
  return match[2] + " in your " + match[1];
});
// or...
warning.gsub(/(salt) in your (eyes)/, function(match) {
  return "#{2} in your #{1}".interpolate(match);
});
```

OK, I take it back: maybe strings *are* more awesome than we realized.

Using JSON

Strings are the building blocks of high-level protocols like HTTP. A client and a server communicate in plain text, which, while much less friendly than the rendered view of a browser, is nonetheless human-readable.

Each time your browser requests an HTML file, it receives one gigantic string, which it then converts into a tree of objects for rendering. It converts between the two according to the rules of HTML. It can be said, then, that HTML is a *serializer*: it takes something inherently nonlinear and makes it linear for storage purposes.

The interesting stuff is done at higher levels, with more complex data types. But on the Web, sooner or later, it all ends up as a string. *JSON (JavaScript Object Notation)* is simply a way to represent these complex structures as strings.

What Does JSON Look Like?

Prototype likes to leverage the literal syntax for object creation, so code like this should be nothing new to you:

```
var vitals = {
  name: "Andrew Dupont",
  cities: ["Austin", "New Orleans"],
  age: 25
};
```

We can describe data structures in JavaScript with a minimum of syntactic cruft. By comparison, let's see what this data would look like in XML:

```
<vitals>
  <name>Andrew Dupont</name>
  <cities>
    <city>Austin</city>
    <city>New Orleans</city>
  </city>
  <age>25</age>
</vitals>
```

XML is verbose by design. What if we don't need its extra features? That's where JSON comes in.

Why JSON?

There are a million different ways to exchange data between client and server; what's so special about JSON?

Very little, really. It's not magical; it's simply the right tool for the job in JavaScript-heavy web apps. There are JSON libraries for all the common server-side languages: PHP, Ruby, Python, Java, and many others. It's far simpler than XML, and thus far more useful when you don't need all of XML's bells and whistles.

We'll revisit JSON in Part 2 of this book when we look into advanced topics in Ajax. But let's familiarize ourselves with the basics right now.

Serializing with Object.toJSON

The best way to learn about the structure of JSON is to try it out yourself. Let's create a few different JavaScript data types and see how they look in JSON:

```
var str = "The Gettysburg Address";
var num = 1863;
var arr = ["dedicate", "consecrate", "hallow"];
var obj = {
  name: "Abraham Lincoln",
 location: "Gettysburg, PA",
  length: 269
};
```

```
Object.toJSON(str); //-> '"The Gettysburg Address"'
Object.toJSON(num); //-> '1863
Object.toJSON(arr);
//-> '["dedicate", "consecrate", "hallow"]'
Object.toJSON(obj);
//>  '{ "name": "Abraham Lincoln", "location": "Gettysburg, PA",
 "length": 269 }'
```

These examples teach you several new things:

- `Object.toJSON` will convert *anything* to JSON, regardless of type.
- `Object.toJSON` *always* returns a string.
- The way items are represented in JSON is *identical* to how they're represented in JavaScript. JSON conforms to the syntax and grammar of JavaScript.

In other words, in each of these cases, the string representation of the data matches the *keyboard characters you'd type to describe them yourself*.

JSON can serialize any JavaScript data type except for functions and regular expressions. The `Date` type gets half credit: there's no literal notation for dates, so they're converted to a string representation.

Unserializing with String#evalJSON

The ease of dealing with JSON in JavaScript should be obvious: since it's valid code, it can simply be evaluated. JavaScript includes an `eval` function that will evaluate a string as though it were code:

```
var data = eval('{ "name": "Abraham Lincoln", "location": "Gettysburg, PA",
 "length": 269 }');
//-> [Object]
```

But there's a problem. It's always dangerous to evaluate arbitrary code unless you know exactly where it's coming from. Prototype's `String#evalJSON` will evaluate a string as code, but it takes an optional Boolean; if `true`, it will make sure the given string is valid JSON—and therefore not a security risk.

```
var str = '{ "name": "Abraham Lincoln", "location": "Gettysburg, PA",
 "length": 269 }';

str.evalJSON();     //-> [Object]

// Ensuring it's valid JSON
str.evalJSON(true); //-> [Object]
// Now try it with invalid, malicious JSON
str = '{ "name": "Abraham Lincoln" }; doSomethingMalicious();'.evalJSON(true);
//-> SyntaxError: Badly formatted JSON string
```

Most of the time, you'll be using JSON simply as a way to communicate with your own server, so security won't be an issue. But if you happen to be handling JSON from a third party, you *must* make sure it's safe to use.

Overriding the Default Serialization

`Object.toJSON` will produce a generic serialization for any object—but, as with `toString` and `toTemplateReplacements`, you can override this default. For instance, we can give our `Player` class (and all its subclasses) instructions on how to serialize themselves—reporting some properties and ignoring all others:

```
// Just first name, last name, and points
Class.extend(Player, {
  toJSON: function() {
    return Object.toJSON({
      firstName: this.firstName,
      lastName:  this.lastName.
      points:    this.points
    });
  }
});

// But maybe subclasses should also report their position...
Class.extend(Quarterback, {
  toJSON: function() {
    return Object.toJSON({
      position:  'QB',
      firstName: this.firstName,
      lastName:  this.lastName,
      points:    this.points
    });
  }
});

var player = new Player("Johnny", "Generic");
Object.toJSON(player);
//-> '{ "firstName": "Johnny", "lastName": "Generic", "points": 0 }'

var qb = new Quarterback("Andrew", "Dupont");
Object.toJSON(qb);
//-> '{ "position": "QB", "firstName": "Andrew",
//->    "lastName": "Dupont", "points": 0 }'
```

You need only define a `toJSON` method to tell your object how to encode itself into JSON.

The preceding example illustrates one of JSON's drawbacks: it isn't a "lossless" format. Since these classes contain functions, there's no way they can be converted to JSON and come back wholly intact. But this is the minimum amount of information we need to restore the class as it existed originally. JSON can't do this automatically, but we can do it manually without much effort.

Using Object Methods

Prototype defines quite a few methods on `Object`, the generic constructor and patriarch of the JavaScript family. Unlike the `String` methods just covered, these aren't instance methods—they're attached to `Object` itself.

Type Sniffing with Object.isX

The hasty conception and standardization of JavaScript in the 1990s left a few gaping holes in the language. One of the biggest holes is the `typeof` operator; in theory it gives us useful information about an item, but in practice it acts like Captain Obvious.

Sure, it handles the simple cases just fine:

```
typeof "syzygy"; //-> 'string'
typeof 37;       //-> 'number'
typeof false;    //-> 'boolean'
```

But when it's asked to do more complex checks, it falls flat on its face:

```
typeof [1, 2, 98];       //-> 'object'
typeof new Date();       //-> 'object'
typeof new Error('OMG'); //-> 'object'
```

`Arrays`, `Dates`, and `Errors` *are* objects, of course, but so are *all* non-primitives. Imagine if you asked me, "Do you know what time it is?" and I responded only with "Yes." My answer is both narrowly correct and completely useless.

The one other value returned by `typeof` is `function`, but that applies both to functions *and* regular expressions:

```
typeof $;           //-> 'function'
typeof /ba(?:r|z)/; //-> 'function'
```

In other words, `typeof` arbitrarily singles out functions, even though they're instances of `Object` just like `Array`, `Date`, and `Error` types. And the fact that `RegExp` is a function (if we're being technical about it) just makes it harder to distinguish them from true functions—what a useless taxonomy.

Prototype includes the type checking that the language left out. In a dynamic, browser-based scripting environment, some checks happen again and again. Prototype defines a handful of methods that test for certain data types: they all accept one argument and return either `true` or `false`.

The Object.isString, Object.isNumber, Object.isFunction Methods

The three simplest of these functions behave exactly like their `typeof` equivalents:

```
Object.isString("foo");   //-> true
Object.isNumber("foo");   //-> false

Object.isFunction($);     //-> true;

Object.isNumber(4);       //-> true
Object.isNumber(3 + 9);   //-> true
Object.isNumber("3" + 9); //-> false
```

Skeptics may wonder why these methods are defined at all—if the behavior is identical to `typeof`, why not use `typeof` instead? These methods are slightly shorter to type than the alternative, but they exist mostly so that you'll get out of the habit of using `typeof`. If that bothers you, feel free to ignore them for philosophical reasons.

The Object.isUndefined Method

Here's another nugget of JavaScript trivia, in case you didn't know: `null` and `undefined` are two separate values. If you asked to borrow a dollar, `null` would be a firm, loud reply of "No!"; `undefined` would be a distant stare, as if I hadn't heard your question.

In other words, `undefined` means that a value has not been set for a variable, and `null` means that the value has been *explicitly* set to nothing. For example, if I omit a named argument from a function, it will be set to `undefined`:

```
function amigos(first, second, third) {
  alert(typeof third);
}

amigos("Lucky", "Ned", "Dusty"); // alerts "string"
amigos("Lucky", "Ned");          // alerts "undefined"
```

JavaScript treats these two properties differently just to screw with our heads:

```
typeof undefined; // "undefined"
typeof null;      // "object"
```

Huh? Why is `null` an object? Do you mean for us to *scream*?

Fortunately, you can test for `null` by using a strict equivalence check (using three equals signs, rather than two). For example, consider `Element#readAttribute`, which returns `null` if the specified attribute does not exist:

```
if ($('lastname').readAttribute('required') === null)
  alert("Last name not required.");
```

On the other hand, the canonical way to check for `undefined` is to use `typeof`. We cannot allow that, however. Instead, we'll use `Object.isUndefined`:

```
function amigos(first, second, third) {
  alert(Object.isUndefined(third));
}
```

The Object.isArray, Object.isHash, Object.isElement Methods

The three remaining type-checking methods test for arrays, hashes (instances of Prototype's `Hash` class), and DOM element nodes. Each of these would respond to `typeof` with "object," but this is unacceptable for arrays and DOM nodes in particular, since they're among the most commonly used objects in the browser environment.

```
var amigos = ["Lucky", "Ned", "Dusty"];
typeof amigos;          //-> 'object'
Object.isArray(amigos); //-> true

var villain = $('el_guapo');
typeof villain;           //-> 'object'
Object.isElement(villain); //-> true

var partyFavors = $H({
  cake:    1,
  amigos:  3,
  pinatas: "plethora"
});
typeof partyFavors;         //-> 'object'
Object.isHash(partyFavors); //-> true
```

Using Type-Checking Methods in Your Own Functions

JavaScript's weak typing is a boon to API developers, since it allows for functions that can take many different combinations of arguments. Consider Prototype's own `$`: it can accept any number of arguments, each of which can be either a string *or* an element. Let's look at the source code for `$`:

```
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);
}
```

The first `if` statement handles the case where there is more than one argument: `$` loops through the arguments, cleverly calls *itself* on each one, and then places each result into an array, returning that array.

The final three lines deal with the common case: `element` is either a DOM element node or a string, so the function sniffs for strings and calls `document.getElementById` to convert them into elements. Thus, by the last line, `element` is guaranteed to be a true DOM node.

Less dynamic languages like Java would require that we define `$` twice: once for passing it a string and once for passing it an element. (Passing an indeterminate number of arguments wouldn't work at all.) JavaScript, on the other hand, lets us write one function for all these use cases; we have the tools to inspect these arguments ourselves.

In the case of `$`, performing a simple type check within the function saves the developer's time: she can write functions that accept strings and DOM nodes indifferently. Calling `$` within those functions will ensure that the end result is an element.

Think how much time you could save by automating the annoying type coercions that your functions demand? Prototype's type-checking methods allow your code to read your mind just a bit better.

Using Array Methods

Way back in Chapter 3, I covered `Enumerable` and the methods it adds to collections. Arrays receive these methods, of course, but they also receive a handful of methods tailored specifically for arrays.

The reverse and clear Methods

The first two methods document themselves. `Array#reverse` inverts the order of an array; `Array#clear` removes all of an array's items.

`Array#reverse` returns a new array by default, but takes an optional Boolean argument to reverse the original array:

```
var presidents = ["Washington", "Adams", "Jefferson", "Madison", "Monroe"];

presidents.reverse();
//-> ["Monroe", "Madison", "Jefferson", "Adams", "Washington"]

presidents;
//-> ["Washington", "Adams", "Jefferson", "Madison", "Monroe"];

presidents.reverse(true);
//-> ["Monroe", "Madison", "Jefferson", "Adams", "Washington"]

presidents;
//-> ["Monroe", "Madison", "Jefferson", "Adams", "Washington"]
```

For obvious reasons, `Array#clear` always acts on the original array:

```
presidents.clear();
presidents; //-> []
```

The uniq and without Methods

Two other methods winnow the contents of the array. `Array#uniq` takes its behavior and its odd name from the corresponding Ruby method—it returns an array without any duplicate values:

```
var presidents = ["Washington", "Adams", "Jefferson",
                  "Madison", "Monroe", "Adams"];

presidents.uniq();
//-> ["Monroe", "Madison", "Jefferson", "Adams", "Washington"]

[1, 2, 3, 3, 2, 3, 1, 3, 2].uniq();
//-> [1, 2, 3]
```

`Array#without` accepts any number of arguments and returns a new array without the specified values:

```
var presidents = ["Washington", "Adams", "Jefferson",
                  "Madison", "Monroe", "Adams"];

presidents.without("Adams", "Monroe");
//-> ["Washington", "Jefferson", "Madison"];
```

Summary

Now that I look back on it, I realize that this chapter hasn't been too embarrassing for either of us. I managed to fill in a few of the cracks I had skipped over in previous chapters, and you got a broader look at the some of the APIs Prototype provides.

Part 2 of this book will focus on script.aculo.us and its robust effects library and set of UI controls. We'll revisit everything covered in this chapter as we explore script.aculo.us through hands-on examples.

PART 2

script.aculo.us

CHAPTER 9

What You Should Know About DHTML and script.aculo.us

DHTML is the term assigned to a collective set of technologies used to create dynamic web content, including HTML, JavaScript, and CSS. The dynamism in DHTML comes from the fact that we're modifying the structure and presentation layers of the document (HTML and CSS) *after* the page has been loaded. This definition is broad enough to cover animations (elements moving across the page over time), new interaction models (drag-and-drop), and new controls (sliders, combo boxes, and in-place editors).

DHTML isn't a special language; it's just a term for what becomes possible when the DOM APIs for markup and style are made available to JavaScript. Some of it relies on stable, agreed-upon standards like the DOM; some of it relies on the mystical APIs that were introduced a decade ago as part of the browser wars. It exists at the intersection of the visual and the analytical, making designers and developers equally uncomfortable.

We'll deal with the analytical parts as we go. The visual parts require a crash course in CSS concepts and a quick look at the APIs that let us control how elements are displayed through JavaScript.

Introducing the CSS Box Model

Web design is all about rectangles, to put it plainly. Every element represented visually is a rectangle of some sort. (That we aren't dealing with circles, triangles, or irregular polygons is a blessing. Be thankful.)

Typically, these rectangles are placed onto the page in a way that represents their hierarchical relationship. The rectangle created by a table cell, for instance, will appear inside the one created by its table row. A parent element, then, acts as a *containing block* for its children.

This is the default rendering behavior for most elements because it's a visual conveyance of the elements' semantics. Most parent-child relationships have *meaning*. A list item belongs to an unordered list; a legend belongs to a fieldset; a table cell belongs to a table row.

As you'll learn, though, there are ways to override this default arrangement. With CSS, you can have near-perfect control of how elements are placed.

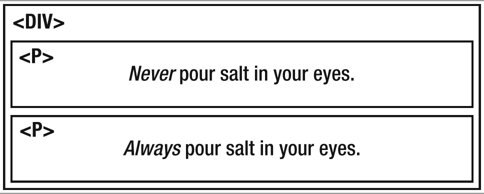
Visualizing with Block-Level Elements

Visually, most elements are represented as blocks—*big* rectangles, if you prefer—and are thus called *block-level elements*.

As the primary pieces of page construction, block-level elements are broadly customizable. Through CSS, you can control their dimensions, colors, and spacing. Also remember that many CSS properties *inherit*—some properties defined on an element propagate to its children unless specifically overridden.

Block-level elements have implicit line breaks. If there are two paragraph elements in a row, for instance, the second will render *below* the first. By default, they won't compete for horizontal space. Figure 9-1 illustrates this behavior.

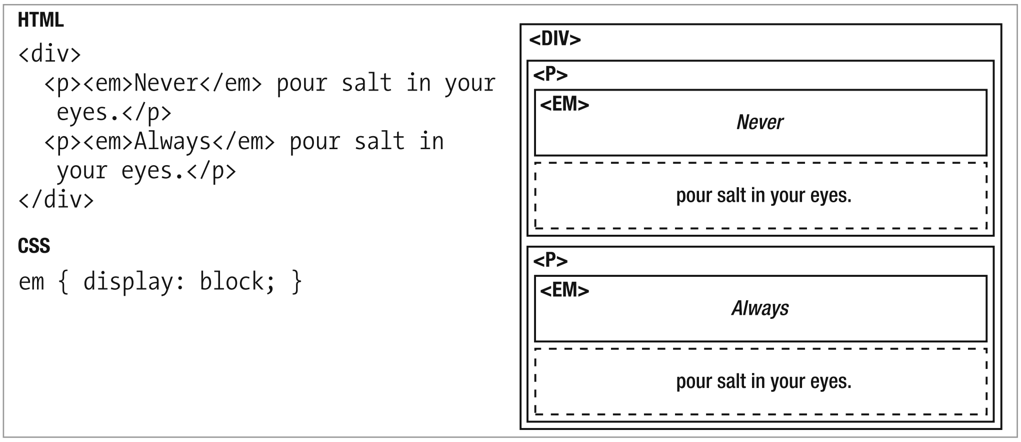
```
<div>
  <p><em>Never</em> pour salt in your
   eyes.</p>
  <p><em>Always</em> pour salt in
   your eyes.</p>
</div>
```



<DIV>
<P>
Never pour salt in your eyes.
<P>
Always pour salt in your eyes.

Figure 9-1. *The markup on the left translates roughly to the structure on the right.*

But, as Figure 9-2 illustrates, any element can be made to behave like a block-level element, even if it isn't one, by setting that element's CSS `display` property to `block`.


HTML

```
<div>
  <p><em>Never</em> pour salt in your
   eyes.</p>
  <p><em>Always</em> pour salt in
   your eyes.</p>
</div>
```

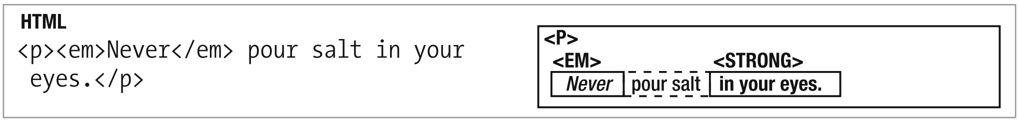
CSS

```
em { display: block; }
```

Figure 9-2. *An element meant for inline display can act like a block-level element if its CSS display property is changed.*

Formatting Blocks with Inline Elements

Other elements don't lend themselves to block formatting. Many—like anchor (`a`), emphasis (`em`), and strong (`strong`)—are meant to be formatted within the lines of a paragraph or other block-level element. They're *inline* elements—*small* rectangles, if you prefer—and are not as greedy or imposing, space-wise, as their block-level brethren. Figure 9-3 illustrates.


HTML

```
<p><em>Never</em> pour salt in your
 eyes.</p>
```

Figure 9-3. *By default, inline elements adopt the size and shape of the text they envelop.*

Inline elements can contain text and other inline elements, but not block-level elements:

```
<!-- RIGHT: -->
<p><strong>Never pour salt in your eyes.</strong></p>

<!-- WRONG: -->
<strong><p>Never pour salt in your eyes.</p></strong>
```

Inline elements *don't* have implicit line breaks. If you wish to start a new line, you must use a line break (`br`) or wrap at least one of the elements inside a block-level element.

You can make any element behave like an inline element by setting its `display` property to `inline`.

We'll revisit inline display later. For now, though, let's take a closer look at block display and the box model.

Thinking Outside the Box: Margins, Padding, and Borders

The box model is day-one material for those learning CSS. But even experts can be surprised by its nuances.

The dimensions of an element are nominally controlled by the CSS `width` and `height` properties, but these control the dimensions of the *usable content area*, not the entire box. A block-level element can also have *padding* (space inside the box) and *margin* (space outside the box).

In between margin and padding lies the box's *border*. A developer can also control a border's thickness, color, and style (see Figure 9-4).

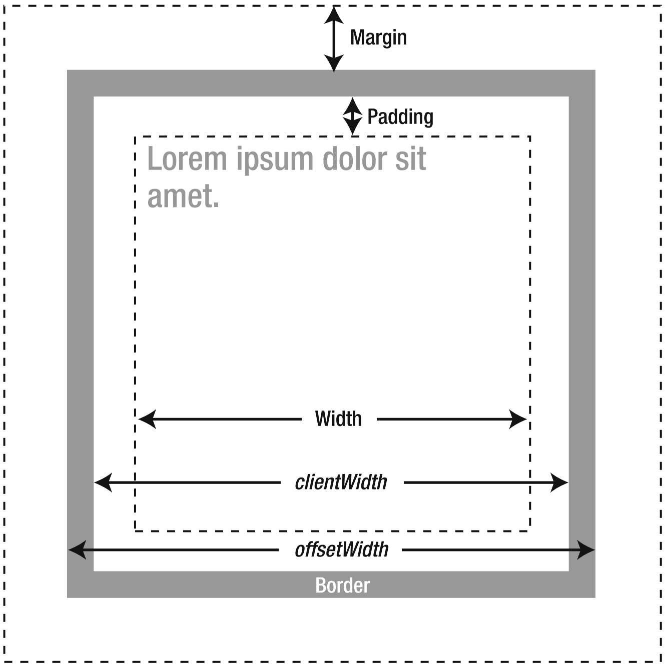
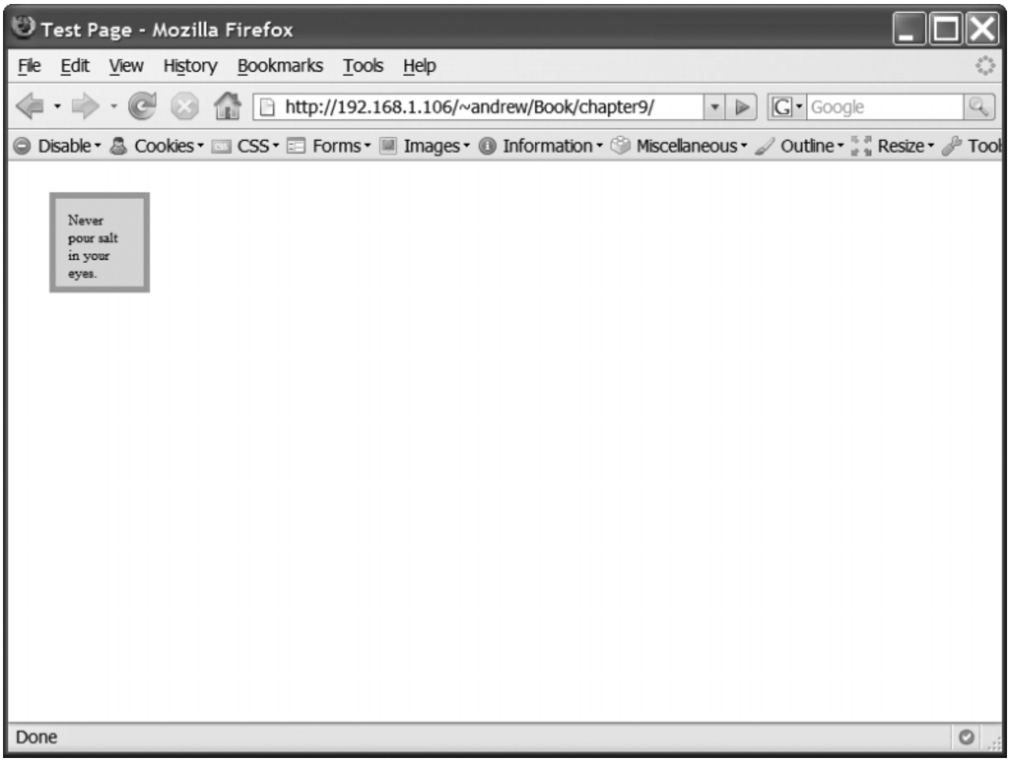


Figure 9-4. *An illustration of common measurements in the CSS box model, along with related DHTML properties*

This might all be clearer in example form (see Figure 9-5):

```
<!-- HTML: -->
<p id="box">Never pour salt in your eyes.</p>

/* CSS: */
#box {
  width: 50px;
  height: 50px;
  background-color: #ddd;
  margin: 25px;
  padding: 10px;
  border: 5px solid #aaa;
}
```


Test Page - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/Book/chapter9/
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tools
Done

Never pour salt in your eyes.

Figure 9-5. *The visual result of the given markup and CSS*

Giving one value for `margin` or `padding` will apply that value to all sides of an element. And the `border` property is shorthand for specifying a border's width, style, and color all at once. So we've got a box with 10 pixels of padding, 25 pixels of margin, and 5 pixels of border *on all sides*.

There are several ways to measure the box's dimensions:

> The most obvious way to the human eye is to measure from the outside edges of the border. By this measurement, our box is 80 pixels wide (50 pixels of content, plus 10 pixels of padding on either side and 5 pixels of border on either side) and 80 pixels tall. Let's call this the *border box*. This corresponds to the `offsetWidth` and `offsetHeight` DHTML properties.

> A related approach would be to measure from the *inside* edges of the border. By this measurement, our box is 70 pixels wide (50 pixels of content, plus 10 pixels of padding on either side) and 70 pixels tall. Let's call this the *padding box*. It corresponds to the `clientWidth` and `clientHeight` DHTML properties.

> The way CSS approaches it is to measure the dimensions of the invisible content box. How much usable space is there *within* the box? By this measurement (excluding all margins, padding, and border), our box would be 50 pixels square, just like we wrote in the CSS. Let's call this the *content box*. A computed style call (i.e., `Element.getStyle`) would report these dimensions.

> The all-encompassing approach involves the *total* amount of space this element occupies. In other words, how big would its parent element need to be in order to contain it? By this measurement, our box would be *130* pixels square: 25 pixels of margin, 5 pixels of border, 10 pixels of padding, 50 pixels of content, 10 more pixels of padding, 5 more pixels of border, and 25 more pixels of margin. Let's call this the *margin box*.

All four of these approaches have an argument for representing the "true" dimensions of a box. In an ideal world, we'd have a pleasant, uniform API for retrieving all four. But in the *actual* world, only two come easily; the rest are left as an arithmetic exercise for developers.

DHTML Properties

Two properties, `clientWidth` and `clientHeight`, are available on every DOM node to report the dimensions of an element's *padding* box. These two properties return integers:

```
$('box').clientWidth;  //-> 50
$('box').clientHeight; //-> 50
```

Likewise, `offsetWidth` and `offsetHeight` report the dimensions of an element's *border* box:

```
$('box').offsetWidth;  //-> 80
$('box').offsetHeight; //-> 80
```

For the content box, it's not quite as simple. The quickest path to the answer is a call to `Element.getStyle`, parsing the result into an integer:

```
var width = $('box').getStyle('width'), height = $('box').getStyle('height');

width;  //-> "50px"
height; //-> "50px"

parseInt(width,  10); //-> 50
parseInt(height, 10); //-> 50
```

This tactic has the disadvantage of being far slower. Luckily, `offsetWidth` and `clientWidth` are often good enough, approximating the actual values nearly enough to justify the optimization. Don't use the computed style approach unless you need total accuracy.

CSS Positioning (Static, Absolute, and Relative)

Those of you who are comfortable with CSS may not need a crash course, but CSS positioning is complex enough that it warrants coverage. CSS defines four different values for the `position` property, corresponding to four different ways to determine an element's position on the page. Three of them enjoy wide support: `static`, `absolute`, and `relative`.

Static Positioning

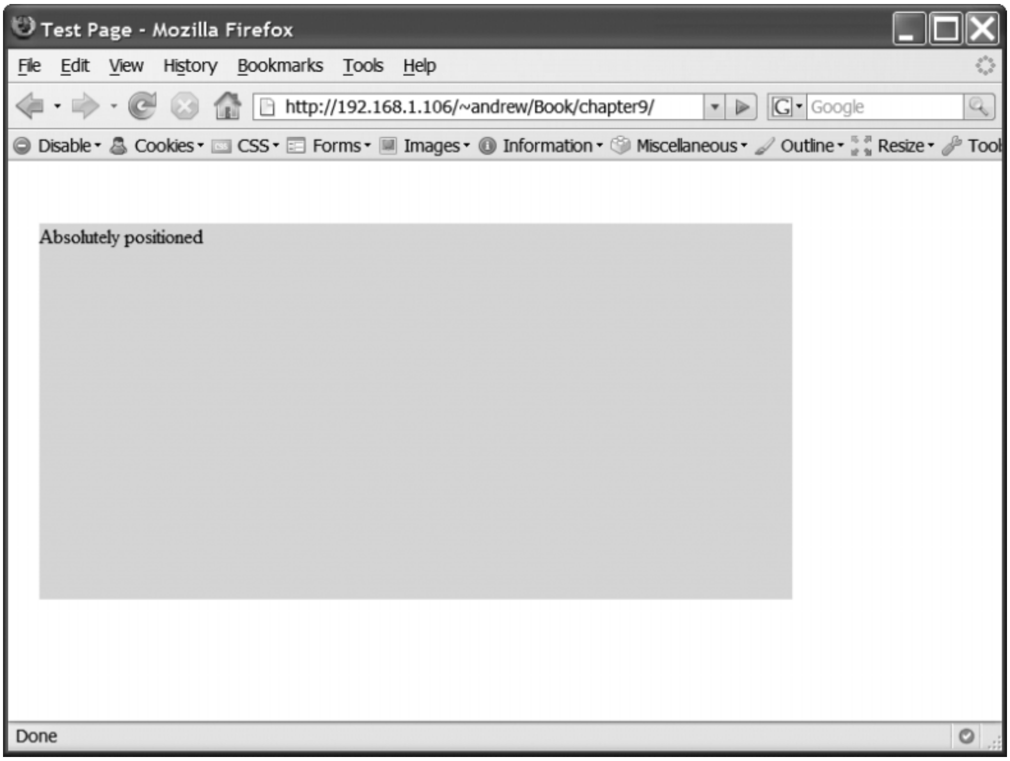
The default value, `static`, works the way you're used to. Elements are arranged from top to bottom in the order they appear on the page. In this mode, the CSS properties for positioning (`top`, `bottom`, `left`, and `right`) have no effect.

Absolute Positioning

Here's where things get weird. Absolute positioning doesn't care about the order of elements on the page. It treats the entire document as a grid, letting you place elements according to pixel coordinates.

For instance, I can create a box and place it on the screen anywhere I want (see Figure 9-6):

```
<!-- HTML: -->
<div id='box'>Absolutely positioned</div>

/* CSS: */
#box {
  position: absolute;
  width: 600px;
  height: 300px;
  top: 50px;
  left: 25px;
  background-color: #ddd;
}
```


Test Page - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/Book/chapter9/
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tool
Done

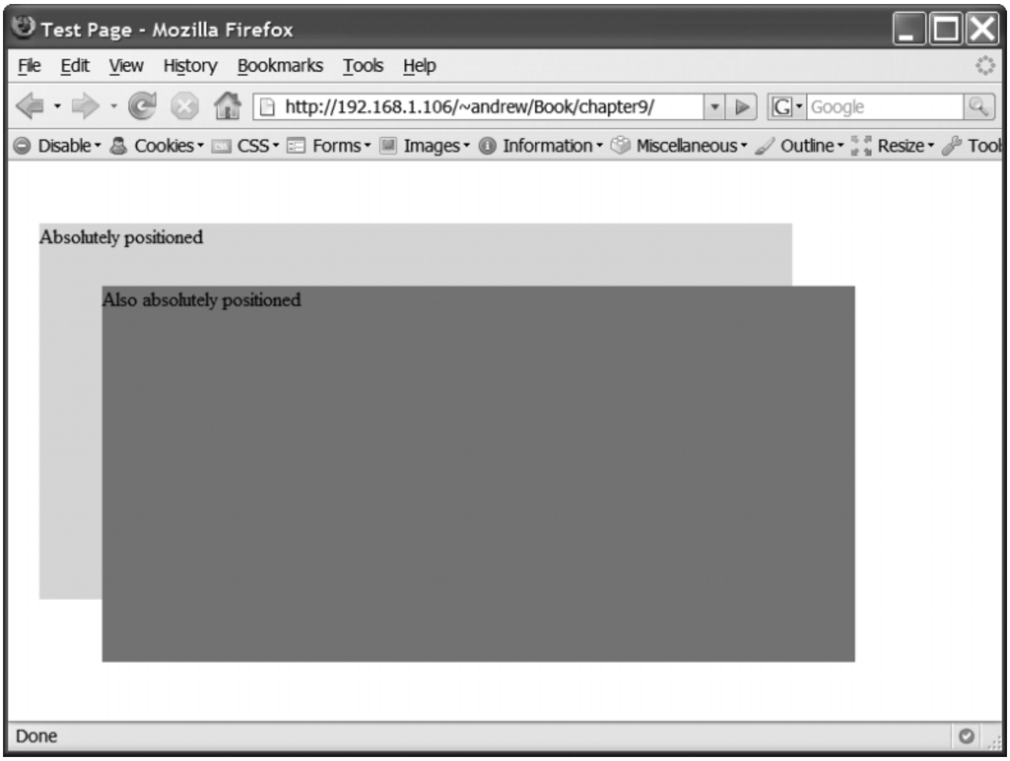
Absolutely positioned

Figure 9-6. *An absolutely positioned block-level element*

In this example, the top-left corner of the page is the origin of the grid. A `top` value of `50px` means that the top edge of the box will be placed 50 pixels from the top edge of the canvas. Likewise, the left edge of the box will be placed 25 pixels from the left edge of the canvas (see Figure 9-7).

```
<!-- HTML: -->
<div id='box'>Absolutely positioned</div>
<div id='box2'>Also absolutely positioned</div>

/* CSS: */
#box {
  position: absolute;
  width: 600px;
  height: 300px;
  top: 50px;
  left: 25px;
  background-color: #ddd;
}

#box2 {
  position: absolute;
  width: 600px;
  height: 300px;
  top: 100px;
  left: 75px;
  background-color: #888;
}
```

When an element is given absolute positioning, it's taken out of the ordinary document "flow"—its statically positioned brethren no longer account for it or make space for it.


Test Page - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/Book/chapter9/
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tool
Done

Absolutely positioned

Also absolutely positioned

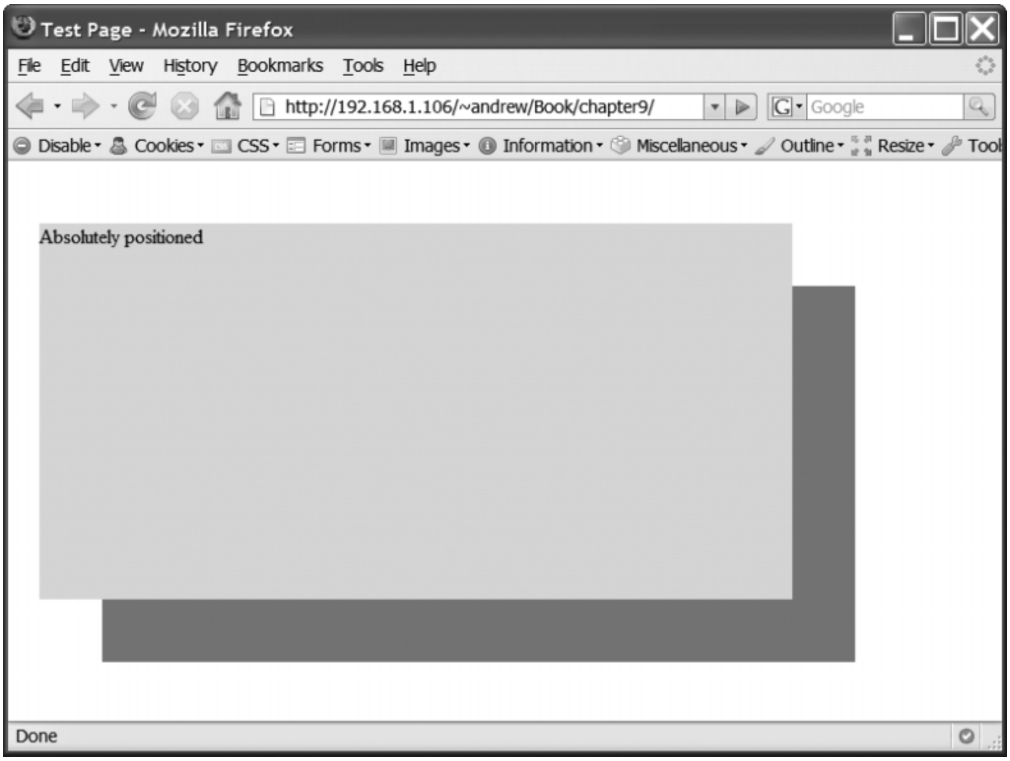
Figure 9-7. *Two absolutely positioned block-level elements*

The z-index Property

I wasn't entirely accurate when I said that order doesn't matter with absolute positioning. In this example, the two elements overlap quite a bit, so order is used to determine which one should be "on top."

But the CSS `z-index` property lets us override that default ordering (see Figure 9-8):

```
#box {
  position: absolute;
  width: 600px;
  height: 300px;
  top: 50px;
  left: 25px;
  background-color: #ddd;
  z-index: 2;
}
```


Test Page - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/Book/chapter9/
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tool
Done

Absolutely positioned

Figure 9-8. *Two absolutely positioned elements. The first has a larger z-index value than the second, so it appears in front.*

The grid analogy serves us well here. Imagine that `left/right` and `top/bottom` control the x and y axes of the grid; `z-index` controls the z axis, otherwise known as the depth. Numbers refer to layers on the page; a higher number represents a layer "closer" to the user. That's why the second box appears on top of the first.

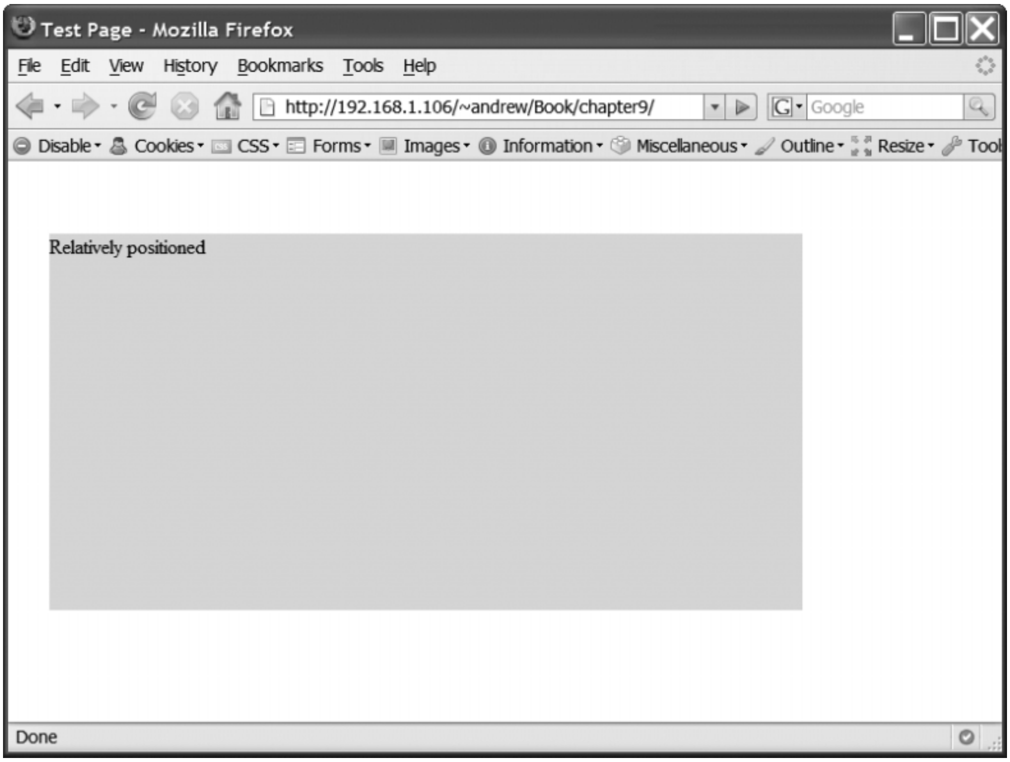
If two elements have the *same* `z-index` value (or have no `z-index` defined), then their order on the page determines which one is on top. But here, by giving the first box a `z-index` of `2`, we've ensured that it will appear above the second box.

Relative Positioning

Relative positioning is the most confusing—it's a mix of static and absolute positioning. Like absolute positioning, it responds to the `top`, `bottom`, `left`, and `right` properties. Unlike absolute positioning, it does *not* remove an element from the document flow.

Also, the element is positioned relative to *its own* top-left corner—not the top-left corner of the document (see Figure 9-9).

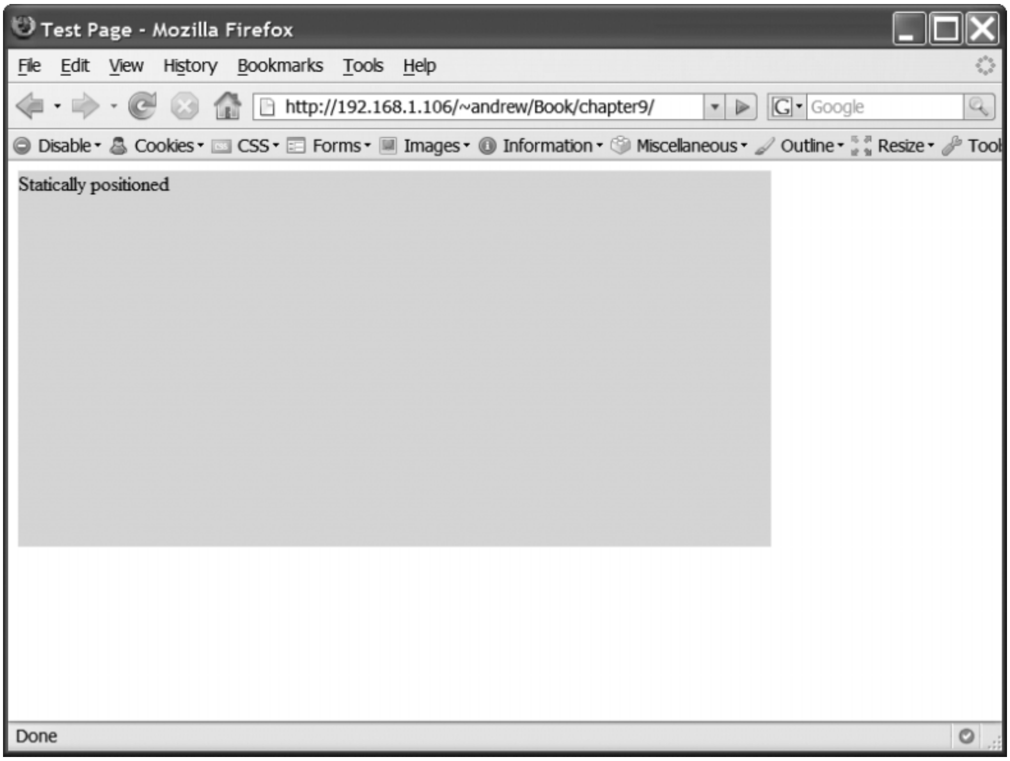
```
#box {
  position: relative;
  width: 600px;
  height: 300px;
  background-color: #ddd;
  top: 50px;
  left: 25px;
}
```


Test Page - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/Book/chapter9/
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tool
Done

Relatively positioned

Figure 9-9. *A relatively positioned element*

This example means to say, "Render the element as you would normally, except place it 50 pixels lower and 25 pixels to the right." Figure 9-10 shows the difference when compared to static positioning.


Test Page - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/Book/chapter9/
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tool
Done

Statically positioned

Figure 9-10. *The same element, but with static positioning*

Offset Parents and Positioning Context

There's one final wrinkle. Elements can be placed relative to the document as a whole, but they can also be placed relative to any other element. We'll call this the element's *positioning context*. It alters the meaning of CSS properties like `top`.

We can see for ourselves how changing the positioning mode of an element affects how that element's children are rendered (see Figure 9-11):

```
<!-- HTML: -->
<div id="box">
  <div id="box2"></div>
</div>
```

```
/* CSS: */
#box {
  width:  300px;
  height: 300px;
  position: static;
  background-color: #ddd;
}

#box2 {
  width:  100px;
  height: 100px;
  position: absolute;
  bottom: 15px;
  right: 15px;
  background-color: #333;
}
```

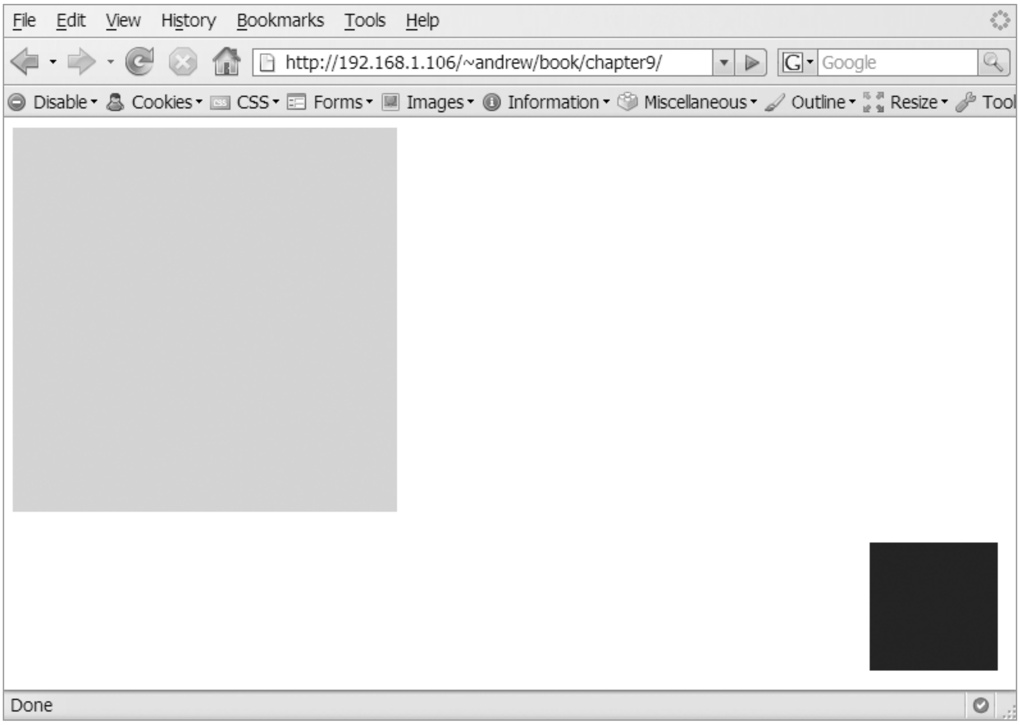

File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/book/chapter9/
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tool
Done

Figure 9-11. *The black box positions itself in the context of the entire document.*

In this example, the child box is positioned relative to the entire document. But when we change its parent's `position` from `static` to `relative`, things look much different (see Figure 9-12).

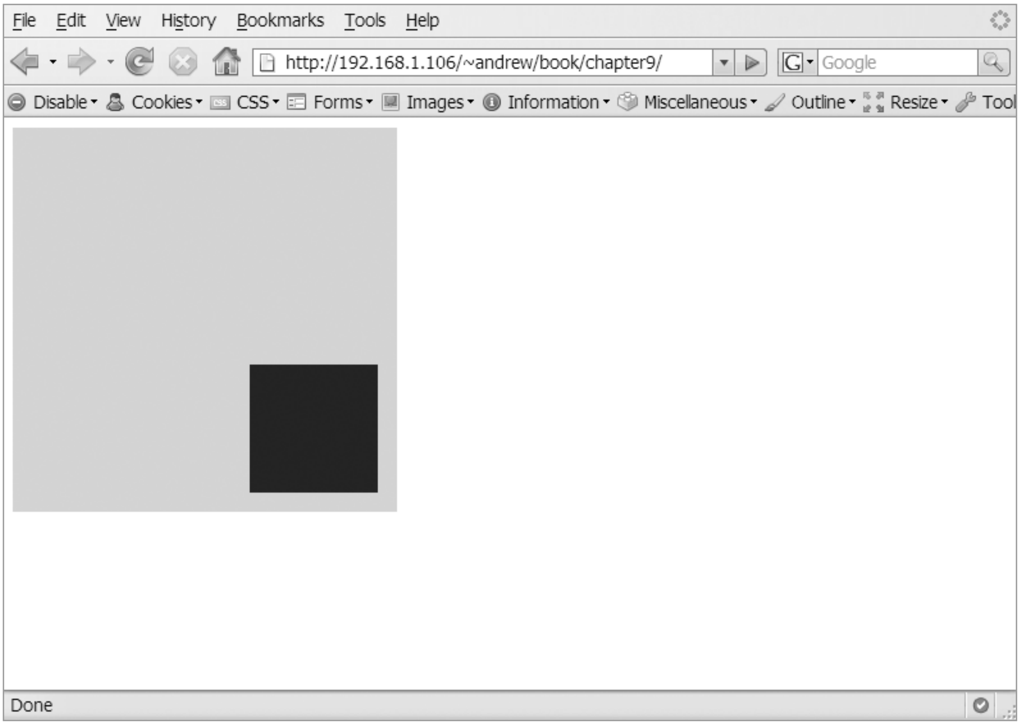

File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/book/chapter9/
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tool
Done

Figure 9-12. *By changing the gray box's position to relative, we define a new context for its child, the black box.*

Any value for `position` other than the default `static` creates a new positioning context for its children. In both examples, the child box is positioned 15 pixels from the bottom and right edges of the canvas; but when we changed the parent box's positioning, we changed the canvas.

Assigning relative positioning to an element won't affect its placement unless it's accompanied by pixel values for `left`, `right`, `top`, or `bottom`. A `position` value of `relative`, then, can be used to assign a new positioning context for all the element's children while leaving it otherwise unchanged.

When we discuss an element's *offset parent*, we're referring to the parent that defines that element's positioning context.

Positioning with Offset Properties

The previously discussed `offsetWidth` and `offsetHeight` properties, which measure an element's border box, have three useful cousins. The first two, `offsetLeft` and `offsetTop`, measure the distance from an element's outside edge to the outside edge of its offset parent.

And, luckily, the `offsetParent` property exists so that we can easily determine an element's offset parent.

```
var box = $('box2');
box.offsetTop;    //-> 185
box.offsetLeft;   //-> 185
box.offsetParent; //-> <div id="box">
```

Once again, these properties are nonstandard. But even the most rabid of standardistas will find it hard not to use them. Stop worrying and learn to love them.

Introducing script.aculo.us

Is your head spinning yet? Aren't you glad script.aculo.us is here to make all this easier?

script.aculo.us is a UI library built upon Prototype. It includes an effects framework, implementations of several kinds of UI widgets, and a drag-and-drop system. Prototype solves general problems; script.aculo.us uses Prototype to solve *specific* problems.

script.aculo.us was developed by Thomas Fuchs, a talented JavaScript developer who is also a member of Prototype Core. The two projects—Prototype and script.aculo.us—have been separate since almost the beginning, but have always had coordinated release schedules. They're fraternal twins.

Similarities to Prototype

If you know Prototype, you'll have no trouble picking up script.aculo.us.

- It's MIT-licensed. script.aculo.us carries the same permissive license as Prototype and Ruby on Rails, so it can be used in any project, whether it's commercial or noncommercial.

- It's bundled with Ruby on Rails. The built-in helpers in Rails use script.aculo.us, but the library can easily be used in any context.

- It has its own site with documentation. The script.aculo.us documentation is collaborative and wiki-based.

- It follows the API conventions of Prototype. It uses many of the patterns you've seen already.
- It has an emphasis on solving the 80 percent of problems common to all web apps. If you've got very specific needs, you can write custom code on top of script.aculo.us.

The script.aculo.us Web Site

You can learn more about script.aculo.us at its web site, at `http://script.aculo.us`, which includes collaborative, wiki-based documentation and other resources to help you when you get stuck.

Contributing to script.aculo.us

Like Prototype, script.aculo.us welcomes bug fixes and enhancements. The bug tracker is open to the public and contains a list of pending script.aculo.us bug reports and feature requests. Patches of all kinds are enthusiastically encouraged.

Getting Started with script.aculo.us

Unlike Prototype, script.aculo.us is distributed as a set of files, instead of just one file. That's because it's divided into modules, most of which are optional.

Figure 9-13 shows what script.aculo.us looks like when you download and unzip it.

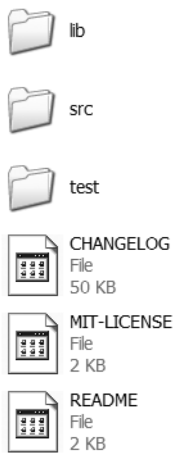


Figure 9-13. *The directory structure of a downloaded script.aculo.us bundle*

The `lib` folder contains a copy of Prototype, just in case you don't have it already. We'll stick with the one we've already got; oftentimes the version bundled with script.aculo.us is a little behind the stand-alone version. But the latest script.aculo.us and the latest Prototype, each fetched from its respective web site, are guaranteed to work together.

The `src` folder contains the files we're interested in:

- `scriptaculous.js` is the main file, the one that declares the script.aculo.us version number and ensures that Prototype is already loaded.
- `effects.js` provides animations and advanced UI flourishes.
- `dragdrop.js` provides drag-and-drop support—the ability to define certain elements on a page as "draggables" that can be dropped onto other elements.
- `controls.js` provides several advanced UI controls, among them an autocompleter (a text field that offers suggestions as you type) and an in-place editor (allowing a user to view and edit content on the same page).
- `slider.js` provides a scrollbar-like "slider"—a button that a user can drag to any point along a track.
- `sound.js` provides a simple API for playing sounds on a web page.
- `builder.js` is a utility file for DOM element creation. Because none of the aforementioned scripts rely on it and it provides no end-user functionality, we won't be covering this part.
- `unittest.js` is a utility file that's used for script.aculo.us unit tests.

Speaking of unit tests, the `test` folder contains a bunch of unit and functional (automated and manual) tests. These tests assert that script.aculo.us does what it claims to do in a cross-browser manner. We won't be bothering with this folder either.

Loading script.aculo.us on a Page

There are several ways to load script.aculo.us into a web page. All of them begin the same way that Prototype is loaded. First, as in Chapter 1, create a boilerplate `index.html` file, and include a `script` tag that references `prototype.js`:

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

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

  <body>
    <h1>Blank Page</h1>
  </body>
</html>
```

To include script.aculo.us, you'll need to copy the contents of the `src` folder into the folder where `index.html` resides. For this example, your JavaScript and HTML code all lives in the same place. It doesn't *have* to, of course; just make sure that all the script.aculo.us files reside in the same directory. Treat them as one unit.

Once you've copied the files over, include a reference to `scriptaculous.js` in your HTML:

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

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

  <body>
    <h1>Blank Page</h1>
  </body>
</html>
```

You're done. The file you included, `scriptaculous.js`, inserts references to the other files (`builder.js`, `effects.js`, `dragdrop.js`, `controls.js`, `slider.js`, and `sound.js`) into your document dynamically.

In other words, script.aculo.us loads every module by default. If there are certain modules you don't need, you can leave them out by explicitly stating which modules you want to load:

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

    <script type="text/javascript" src="prototype.js"></script>
    <script type="text/javascript"
     src="scriptaculous.js?load=effects,dragdrop"></script>
  </head>

  <body>
    <h1>Blank Page</h1>
  </body>
</html>
```

The query string at the end of the URL should contain a comma-separated list of the modules you want to load.

Loading these script.aculo.us modules through one central script is helpful to you, the developer; among other things, it verifies that your versions of Prototype and script.aculo.us are compatible with one another. When your site goes live, however, it's a good idea to reference the modules you need *directly*. A standard Ruby on Rails application uses a subset of script.aculo.us and references the modules by name:

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

Summary

Part 1 of this book dealt in the abstract, but Part 2 will be concrete and hands-on. You'll build a site from scratch, using Prototype and script.aculo.us to ensure a rich, intuitive UI model.

Because script.aculo.us is so rooted in Prototype, you'll recognize some methods and classes along the way, and you'll even notice how groundwork laid by Prototype enables a specific feature of script.aculo.us.

In other words, Part 2 isn't separate and distinct from Part 1; it's an application of what you've already learned. Feel free to jump back to Part 1 whenever you need a refresher on how a certain aspect of Prototype works.

CHAPTER 10

Introduction to script.aculo.us Effects

What Are Effects?

Bear with me for the next few pages. I'm going to introduce a lot of terms that, while general, are used in script.aculo.us in very specific ways.

In script.aculo.us, an *effect* is the modification, over time, of any aspect of an element. For an animation of any sort, you need three things:

- The target of the animation: an element
- Starting and ending points: typically integers, like pixel values
- A way to get from beginning to end incrementally: a function

All the effects in script.aculo.us—and any custom effects you create on your own—consist of these three things. For example, I can make an element move across the page by positioning it absolutely and then changing its `left` CSS property little by little. To describe the effect, I'd need to identify the element I want moved, figure out starting and ending values, and then explain how to apply those values to the `left` property.

More abstractly, some effects can be seen as time-lapse versions of other transformations. Instead of simply hiding an element (by calling `Element#hide` on it), I can call a "fade" effect, which hides the element *over time* by gradually decreasing its opacity.

Why Effects?

I can hear the mob outside my door. Don't waste our time and patience with superfluous animations! Surely you're not suggesting we take our design cues from 30-second Flash intros so ubiquitous that even the dog catcher's campaign site includes one?

Extinguish your torches. Put down your pitchforks. Let me explain.

Animations and effects are not the same thing. Effects are well established in the desktop world. When you minimize a window, Windows will show the window collapsing into its title bar and then shrinking into its slot in the task bar. Mac OS X will show the window getting sucked, genie-like, into its reserved space in the dock.

These animated effects are meant to reinforce the result of an action. They're not Disney-esque pixie dust following your mouse pointer or window frames pulsating to the tempo of techno music.

For our purposes, we can divide animations into two groups: the purposeful and the superfluous. A window that shrinks when minimized belongs to the former group. It's *friendly* because it serves as a guide that reinforces the action. It illustrates that the window has assumed a different form.

Effects are designed to attract the user's attention, so be sure to use them only when the user's attention is needed. Constantly calling attention to mundane things is a great way to annoy people. ("Look! There's a *wastebasket*!") Don't be that guy.

But the point remains—there are *legitimate use cases* for animated effects in your web app. Don't avoid all effects; avoid *gratuitous* effects.

When Effects Are Good

Effects in web apps grab the user's attention, so your application should embrace effects as a way to mark what's important. Here are a handful of good reasons to use effects in your web app:

To show motion: Many applications employ some sort of spatial metaphor—a user can move things around on the page. A "to-do list" application might organize items into two sections, "completed" and "not yet completed"; checking a box would move an item from the latter into the former. Without a reinforcing effect, this action could startle the user. It's important to show that the item hasn't disappeared outright. The obvious way to express this is through actual motion—the item moves in a straight line from its original spot to its destination—but other effects can be used, as well.

To alert the user of new content: As web apps move toward a single-page model, in which all content is fetched through Ajax instead of through traditional page loads, it becomes more important to show the user when things change on the page. Imagine a news reader application that fetches headlines from major sites. When new headlines get added to the page, they'll more easily be noticed if they fade in. If, on the other hand, they suddenly *appear* on the page, the user may not notice unless he's looking at the screen at that very instant.

To show what's changed on a page: A similar (but slightly different) use case is alerting the user when *existing* content changes.

The Basics of Effects

Consider an element that's 50 pixels square and absolutely positioned at the top-left corner of the page, as in Figure 10-1.

```
/* CSS: */
#box {
  position: absolute;
  width: 50px;
  height: 50px;
  top: 0;
  left: 0;
  background-color: #999;
  border: 2px solid #000;
  font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;
  text-transform: uppercase;
  text-align: center;
  line-height: 50px;
  font-size: 10px;
}
<!-- HTML: -->
<div id="box">Box</div>
```

Figure 10-1. *An absolutely positioned div*

If we want to move it down and to the right, we can do so instantaneously:

```
var box = $('box');
box.setStyle({ left: '50px', top: '50px' });
```

Here, we're changing the element's `left` and `top` properties from `0px` to `50px`. When we set the new style, it happens instantaneously—the box jumps down to its new place, as in Figure 10-2.

Figure 10-2. *The div's style has been dynamically altered.*

But let's try moving the element gradually—by increasing the `top` and `left` values little by little until we arrive at the desired `50px`. Our first attempt might look something like this:

```
var box = $('box');
for (var i = 1; i <= 50; i++)
  box.setStyle({ left: i + 'px', top: i + 'px' });
```

Unfortunately, when we run this code, we find that it behaves exactly the same way as the first example—the box seems to jump to its new coordinates without any steps in the middle. Why?

Take a moment to think like a computer. It doesn't take that much time to change a CSS property—let's say around 1ms. Multiply that by 50 and you've got 50ms, still less than one-tenth of a second. The animation is happening, but far too fast for a human to notice. It might even be happening faster than the rendering engine can update.

But speed problems are easy to work around. We need to slow things down by pausing after each frame of the animation so that our human eyes (and the browser's rendering engine) can catch up. Remember that JavaScript has no `sleep` statement; we can't tell the interpreter to halt, but we can tell it to do other things for a while, and that's good enough:

```
function incrementBox(value) {
  $('box').setStyle({ left: value + 'px', top: value + 'px' });
  if (value < 50) incrementBox.delay(0.01, ++value);
}

incrementBox(1);
```

Remember `Function#delay`? It's an excellent way to schedule a function to run later—one hundredth of a second later, in this case. Here, `incrementBox` schedules itself to run later, over and over, until we reach the desired value. We need to call it only once to set the effect in motion. Figure 10-3 shows the "before" and "after" states of the box.

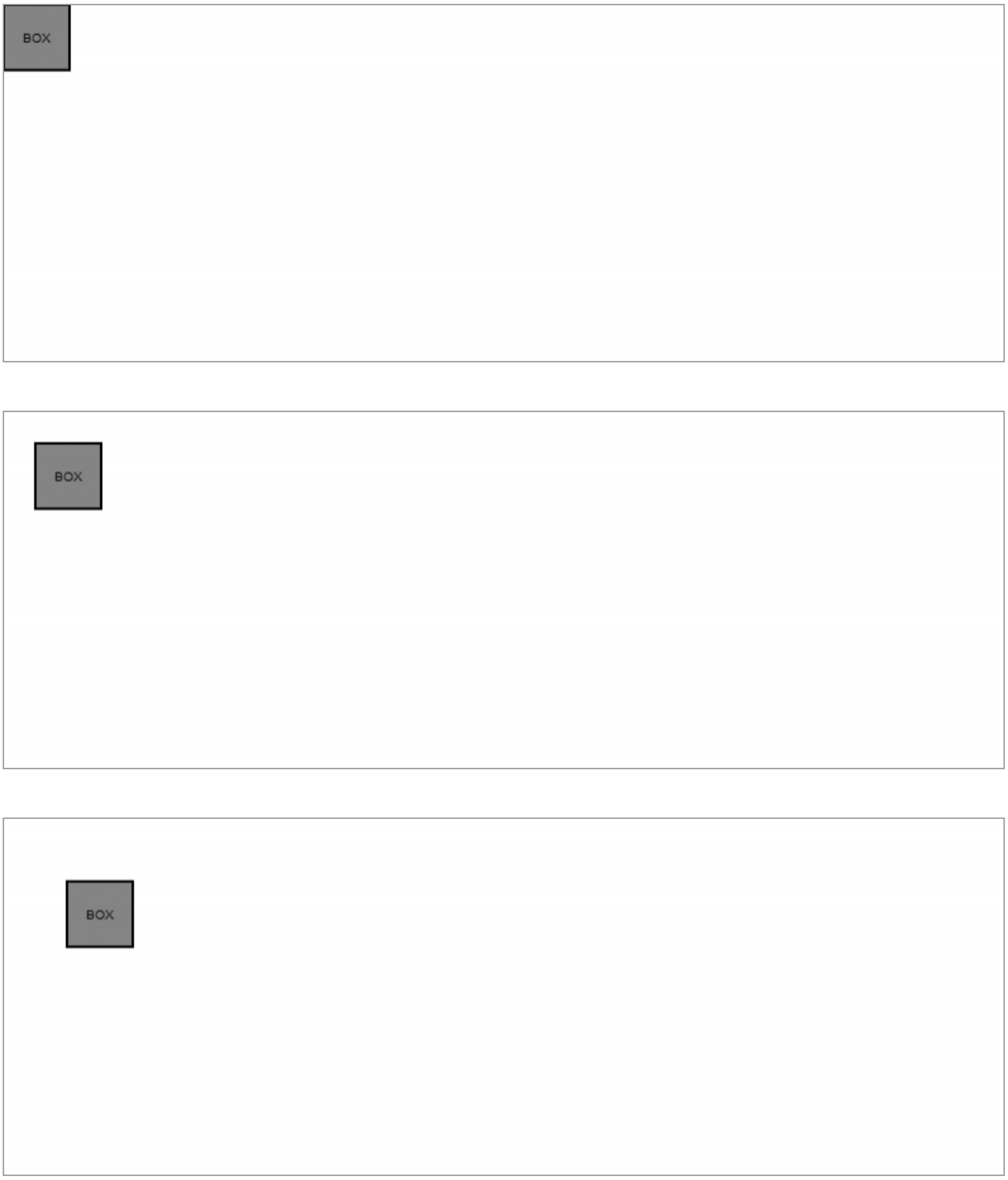


Figure 10-3. *The element animates into its new position.*

So let's look at the ingredients of this effect:

- The element (`<div id="box">`) is the target of the effect.
- The starting and ending points are pixel coordinates (`0, 0` to `50, 50`).
- The incrementing is done by `Element#setStyle`.

This method works, but not as well as it could. It's far too dependent on the speed of the computer it runs on, and it's not guaranteed to move at a constant pace. Our call to `Function#delay` guarantees that our function will be run *at least* 10ms later, but if the JavaScript engine is busy with something else, we could be waiting for quite a bit longer. We're guaranteed to get 50 frames of animation, but the element might not move at a constant speed.

script.aculo.us Effects

Fortunately, script.aculo.us manages all these annoying details for you. It creates a base class for running effects, and then defines scads and scads of specific effects that inherit from that base class. The effects themselves vary wildly, but they all have several things in common:

- They all act on an element.
- They all involve transforming that element's style from a starting point to an ending point over a specified amount of time. No matter how slow the computer or how overwhelmed the JS engine, script.aculo.us effects will always execute in the amount of time specified.
- They all accept *callbacks* that let us run our own functions at certain milestones in the animation. These callbacks are very similar to the Ajax callbacks covered in Chapter 4.

The previous exercise was meant to give you an idea of how effects work under the hood. script.aculo.us defines a base class that solves all the problems we encountered—a boilerplate for any sort of browser-based effect. It then defines subclasses that do specific things: one animates opacity, another animates position, and so on. These are called *core effects.* Finally, it defines some functions that build on these core effects by running two or more in parallel. These are called *combination effects.*

Using Effect.Morph

Simple cases first, though—let's look at `Effect.Morph`, the most generic (and most versatile) of all effects. It works like a time-lapse version of `Element#setStyle`: it takes an element and some CSS rules to apply to the element, and slowly "morphs" the element according to the new rules (see Figure 10-4).

```
new Effect.Morph('box', {
  style:    "left: 50px; top: 50px;",
  duration: 1.0
});
```

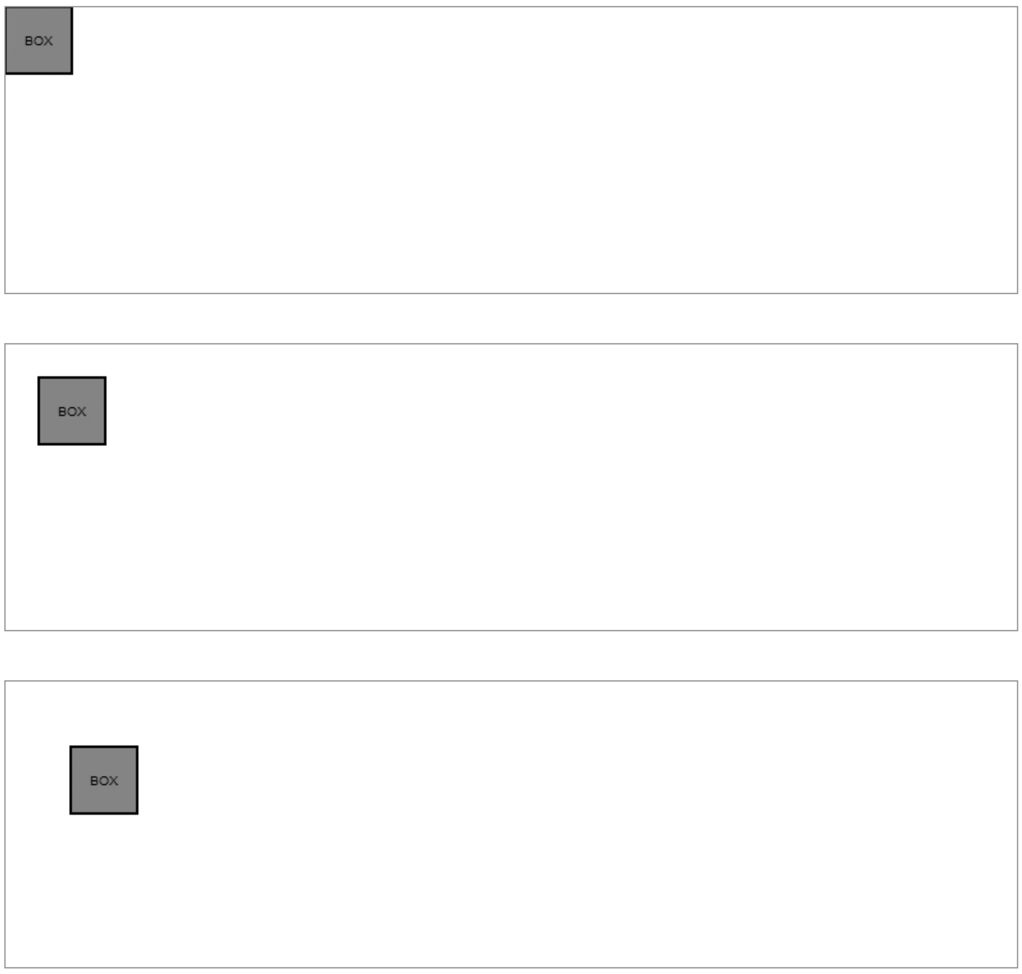


Figure 10-4. *The Effect.Morph call animates the box into the desired position.*

In other words, from the starting and ending values of each element's properties, `Effect.Morph` figures out which CSS properties it needs to change and gradually adjusts them over the specified period of time.

`Effect.Morph` uses the options-argument pattern we've come to love. The first parameter, `style`, can be either a string or an object in the form expected by `Element#setStyle`. An optional `duration` argument indicates how long the effect should last; if absent, it defaults to 0.5 seconds.

script.aculo.us also adds `Element#morph` for maximum convenience. This code example is equivalent to the preceding one:

```
$('box').morph("left: 50px; top: 50px;", { duration: 1.0 });
```

`Effect.Morph` works because a vast majority of CSS properties lend themselves to *tweening*—figuring what goes in between a starting point and an ending point. Anything with a quantitative value—pixels, ems, hex color values, and so on—can be morphed.

```
$('box').morph("width: 500px");
$('box').morph("font-size: 1.6em");
$('box').morph("background-color: #cc5500");
```

How Does It Do That?

As we've established, all effects need to know several things: the element to change, the aspect of that element to change, and starting and ending values. `Effect.Morph` lets us specify, in very precise form, all of those things except for one—we don't need to specify a starting value because it's assumed we want to start at *whatever state the element is already in*. This shortcut limits the overall expressive power of `Effect.Morph`, but broadens its applicability.

To illustrate this, let's place our element at a different starting point:

```
#box {
  position: absolute;
  width: 50px;
  height: 50px;
  top: 200px;
  left: 100px;
  background-color: #999;
  border: 2px solid #000;
  font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;
  text-transform: uppercase;
  text-align: center;
  line-height: 50px;
  font-size: 10px;
}
```

Our JavaScript needn't be changed:

```
new Effect.Morph('box', {
  style:    "left: 50px; top: 50px;",
  duration: 1.0
});
```

Now, instead of moving down and to the right, the box moves up and to the left, as shown in Figure 10-5. `Effect.Morph` is smart enough to figure out how to get from A to B.

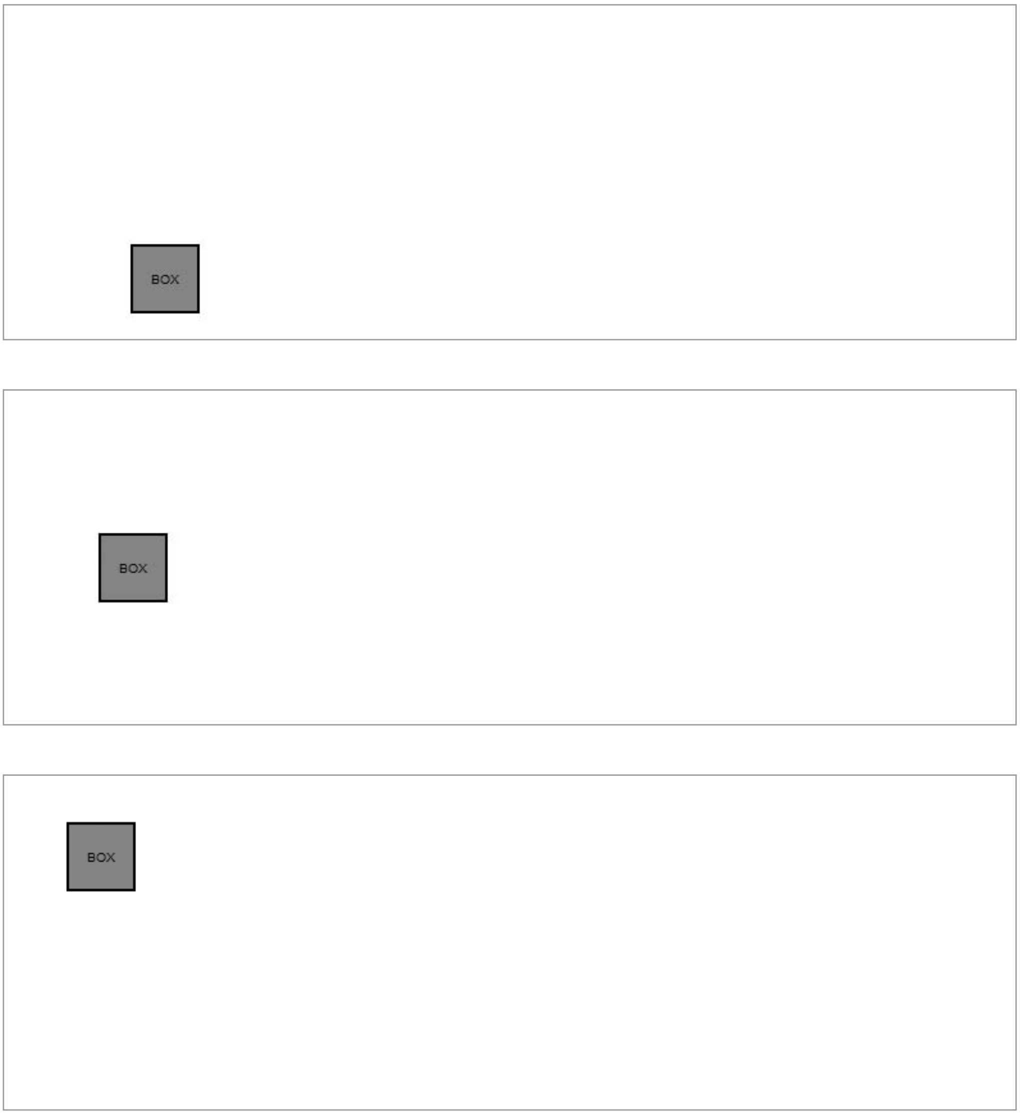


Figure 10-5. *The element starts somewhere else, but ends up where the previous animation started.*

Morphing in Parallel

Perhaps the best thing about `Effect.Morph` is its ability to animate several different things at once. Let's see how it handles some curveballs:

```
/* CSS: */
#box {
  position: absolute;
  width: 50px;
  height: 50px;
  top: 0;
  left: 0;
  background-color: #999;
  border: 2px solid #000;
  font-family: "Helvetica Neue";
  text-transform: uppercase;
  text-align: center;
  line-height: 50px;
  font-size: 10px;
}

/* JS: */
$('box').morph({
  width: "100px",
  height: "100px",
  top: "125px",
  left: "150px",
  backgroundColor: "#000",
  borderColor: "#bbb",
  color: "#fff",
  lineHeight: "100px",
  fontSize: "18px",
  textTransform: "lowercase",
}, { duration: 2.5 });
```

Still with me? We're animating a lot of properties here: the box's size, positioning, background color, and border color; plus the text's color, size, and line height. (We change line height in order to keep the text vertically aligned in the center of the box.)

We set a longer duration for this effect to make sure all these changes are happening in parallel. `Effect.Morph` handles it with aplomb (see Figure 10-6).

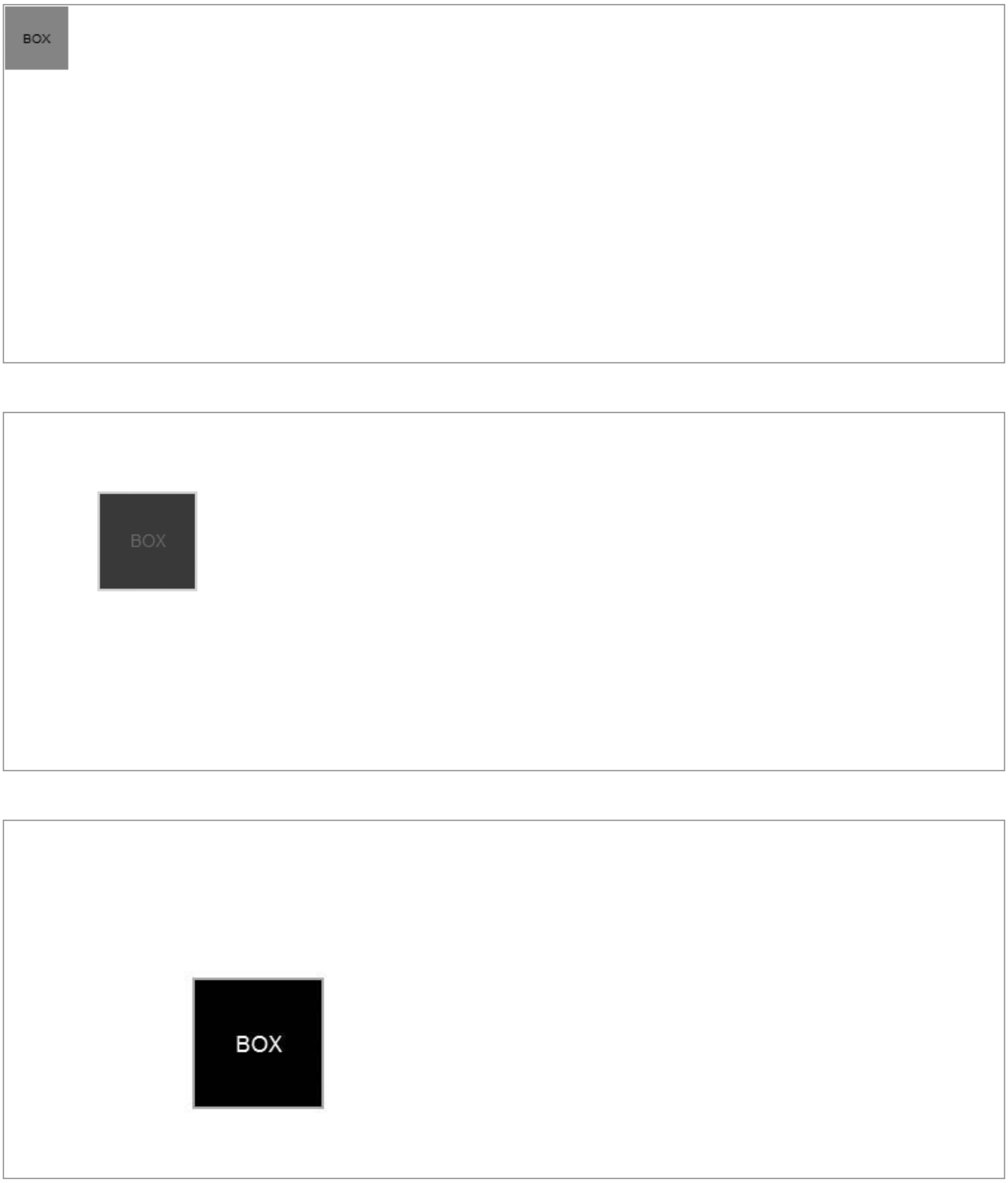


Figure 10-6. *Effect.Morph can animate any number of CSS properties at once.*

Notice, however, that our `text-transform` property was ignored. There's no way to *animate* the switch from uppercase to lowercase, so `Effect.Morph` can't do anything with it.

Before we move on to other effects, let's try one more thing. Take all the styles in the preceding `morph` call and place them into their own CSS class:

```
#box.big-box {
  width: 100px;
  height: 100px;
  left: 150px;
  top: 125px;
  background-color: #000;
  border-color: #bbb;
  color: #fff;
  line-height: 100px;
  font-size: 18px;
  text-transform: lowercase;
}
```

After all, this is where style information belongs. Stuffing it all into a JavaScript string made things awkward. Moving it into CSS makes our code much cleaner (see Figure 10-7):

```
$('box').morph('big-box', { duration: 2.5 });
```

There's some magic going on in Figure 10-7. `Effect.Morph` can tell we're giving it a class name instead of a string of CSS rules, so it reads the style information to figure out what needs to be animated. It performs the animation just as before, but at the very end it *adds the class name* onto the element.

So you've seen that `Effect.Morph` is a time-lapse version of both `Element#setStyle` and `Element#addClassName`. In other words, instead of using these two methods to change element styles instantly, you can use `Effect.Morph` to change them gradually over a span of time.

You've also seen the advantage of giving `Effect.Morph` a class name: our untweenable property, `text-transform`, is no longer ignored. It takes effect at the end of the animation—when the `big-box` class name is added to our box.

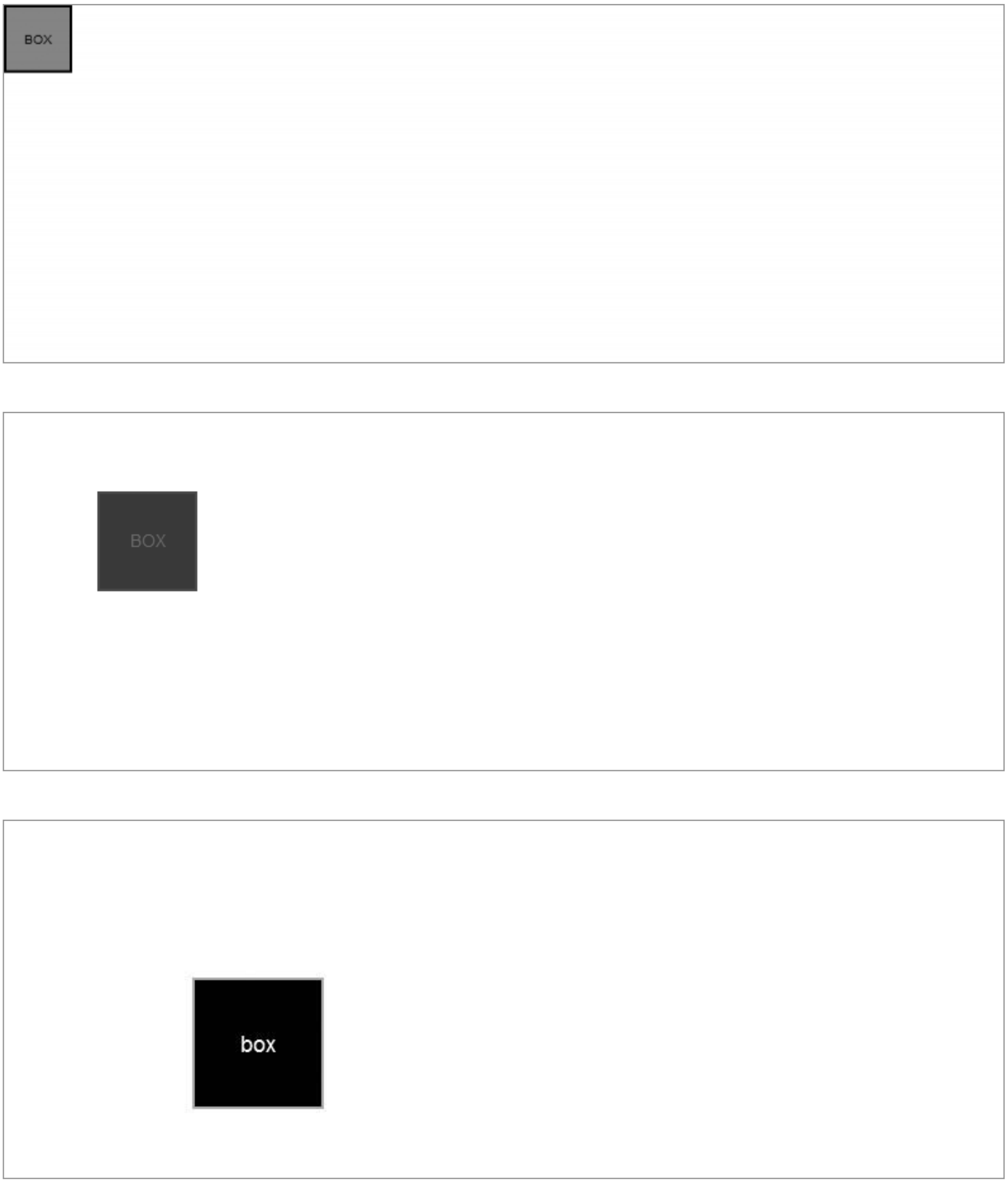


Figure 10-7. *Effect.Morph can handle CSS class names as well.*

Other Core Effects

script.aculo.us sports several core effects for use when `Effect.Morph` can't do the job. They're called *core* effects because they're the building blocks of the effects library—each one modifies elements in one specific way. They can be combined to form more elaborate effects, some of which we'll look at later on.

Effect.Move

`Effect.Move` handles positioning—moving an element any number of pixels (see Figure 10-8):

```
new Effect.Move('box', { x: 50, y: 50 });
```

`Effect.Move` takes a `mode` parameter with two possible values—`relative` and `absolute`—which mirror the two CSS positioning modes of the same names. The default, `relative`, indicates motion relative to the element's current position. The `x` and `y` parameters, then, indicate movement in the horizontal and the vertical directions. In `absolute` mode, the `x` and `y` parameters indicate where to place the element relative to its offset parent.

`Effect.Move` handles nearly all block-level elements with grace, even those that have a CSS `position` of `static`. It will "absolutize" or "relativize" the element before changing its position.

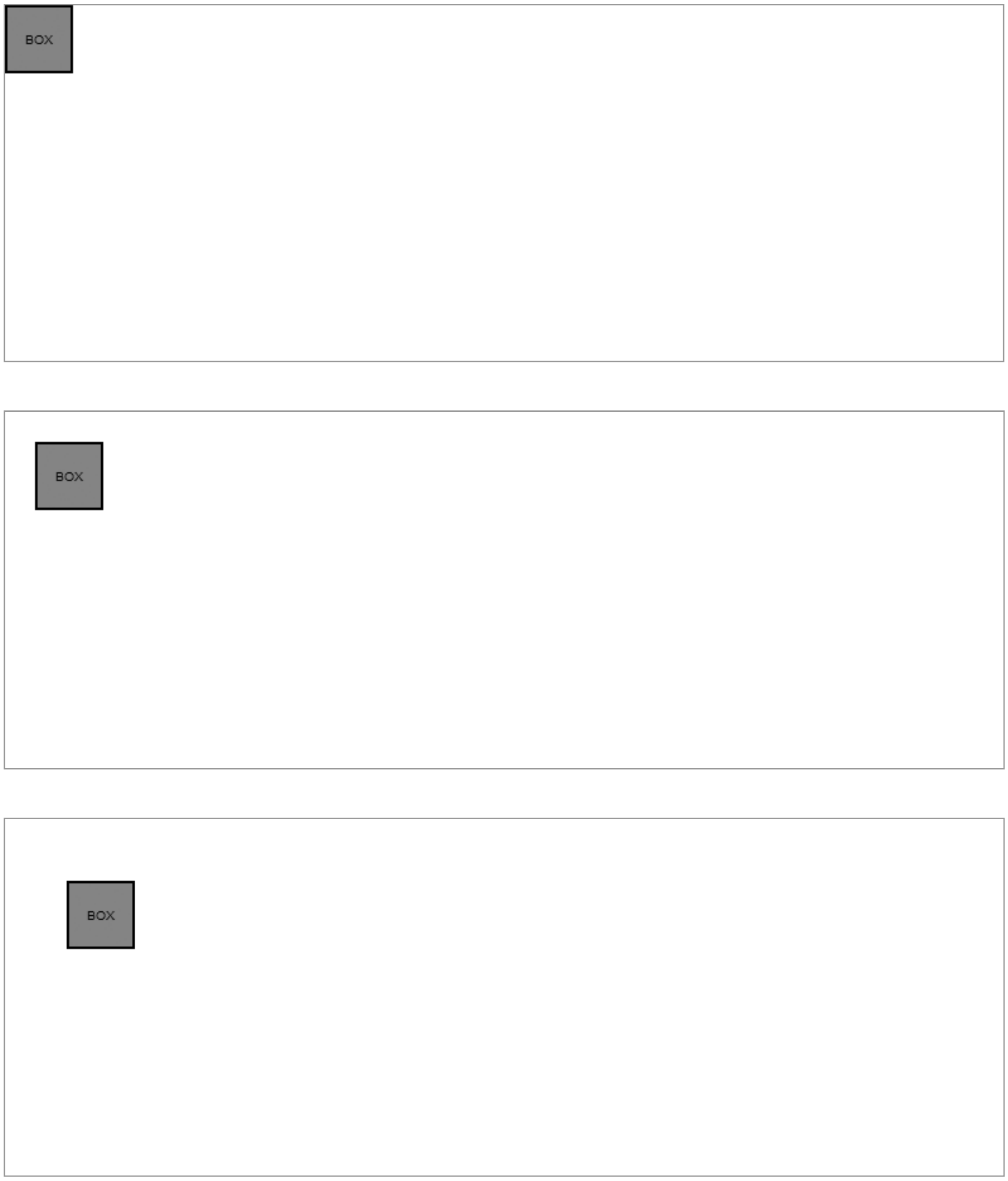


Figure 10-8. *Effect.Move animates an element's position.*

Effect.Scale

`Effect.Scale` handles, well, scaling—changing the size of an element (and, optionally, the size of its contents) by a given percentage (see Figure 10-9):

```
new Effect.Scale('box', 150);
```

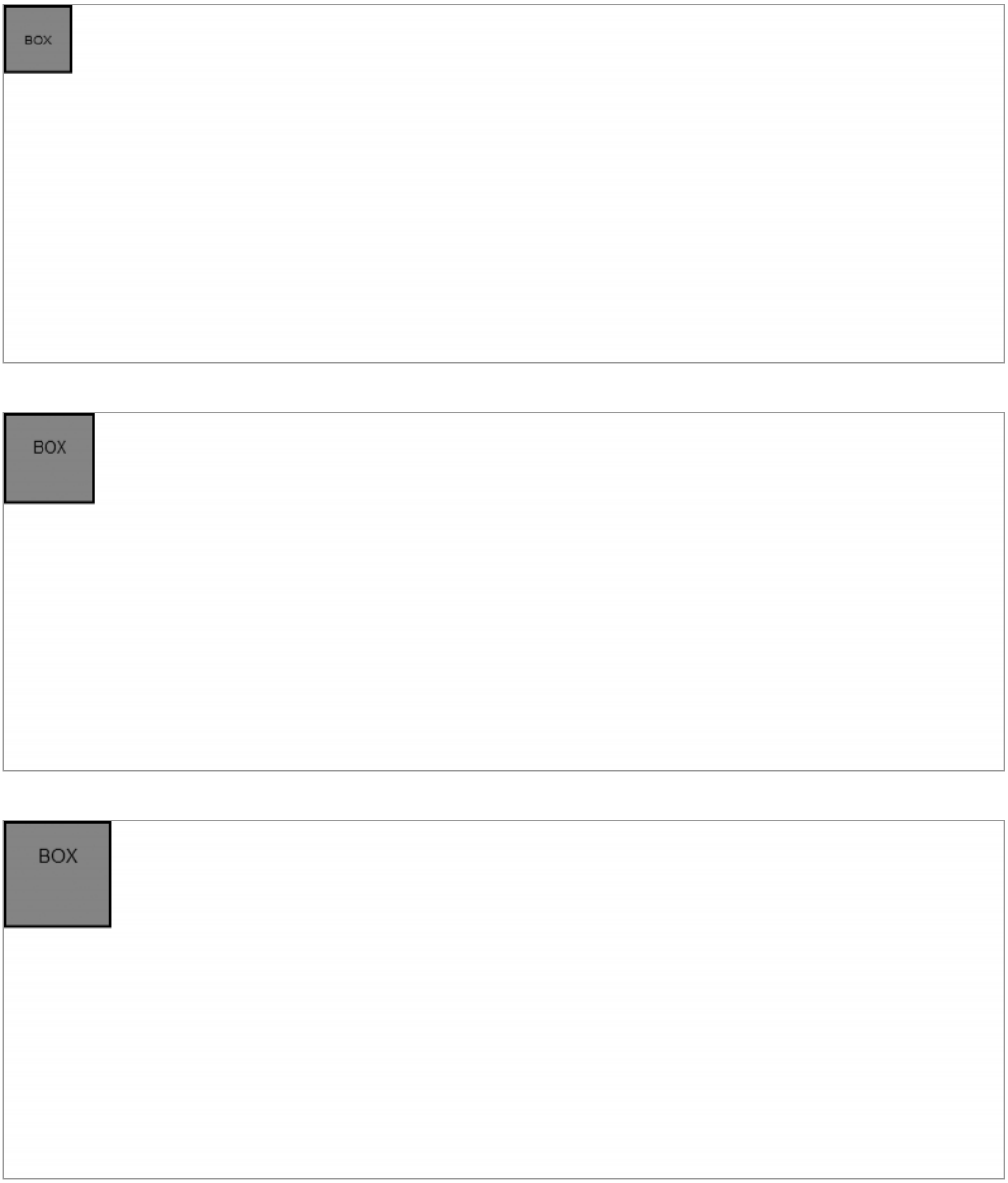


Figure 10-9. *Effect.Scale animates an element's height and width.*

The second argument is an integer that represents the scaling percentage. In this example, the element grows to 150 percent of its original size. (Note that the contents of the box have been scaled as well—the text is also 150 percent bigger.) `Effect.Scale` also supports several optional parameters for more granular control of the effect (see Figure 10-10):

```
new Effect.Scale('box', 100,
 { scaleContent: false, scaleFrom: 50.0, scaleFromCenter: true });
```

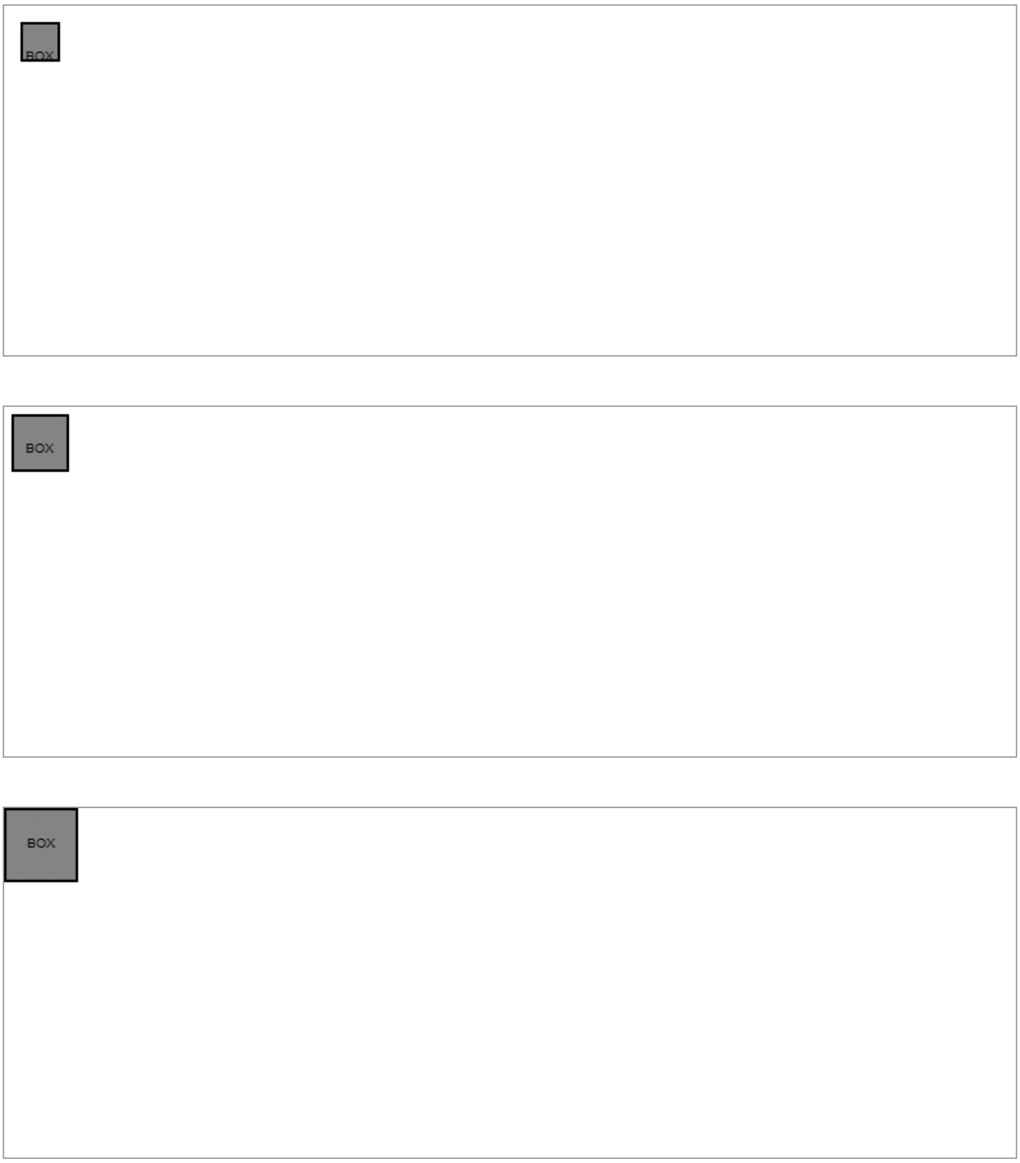


Figure 10-10. *Effect.Scale is broadly configurable.*

First, we have `scaleContent`, which, when set to `false`, will scale only the box and not the text inside. The second parameter, `scaleFrom`, can be used to set a different initial percentage. Here, the box will jump to 50 percent of its original size, and then animate back to 100 percent of its original size. Finally, by setting `scaleFromCenter` to `true`, we can ensure that the center of the box, not the top-left corner, remains fixed throughout the effect. Figure 10-11 shows the result.

```
new Effect.Scale('box', 150, { scaleX: false, scaleY: true });
```

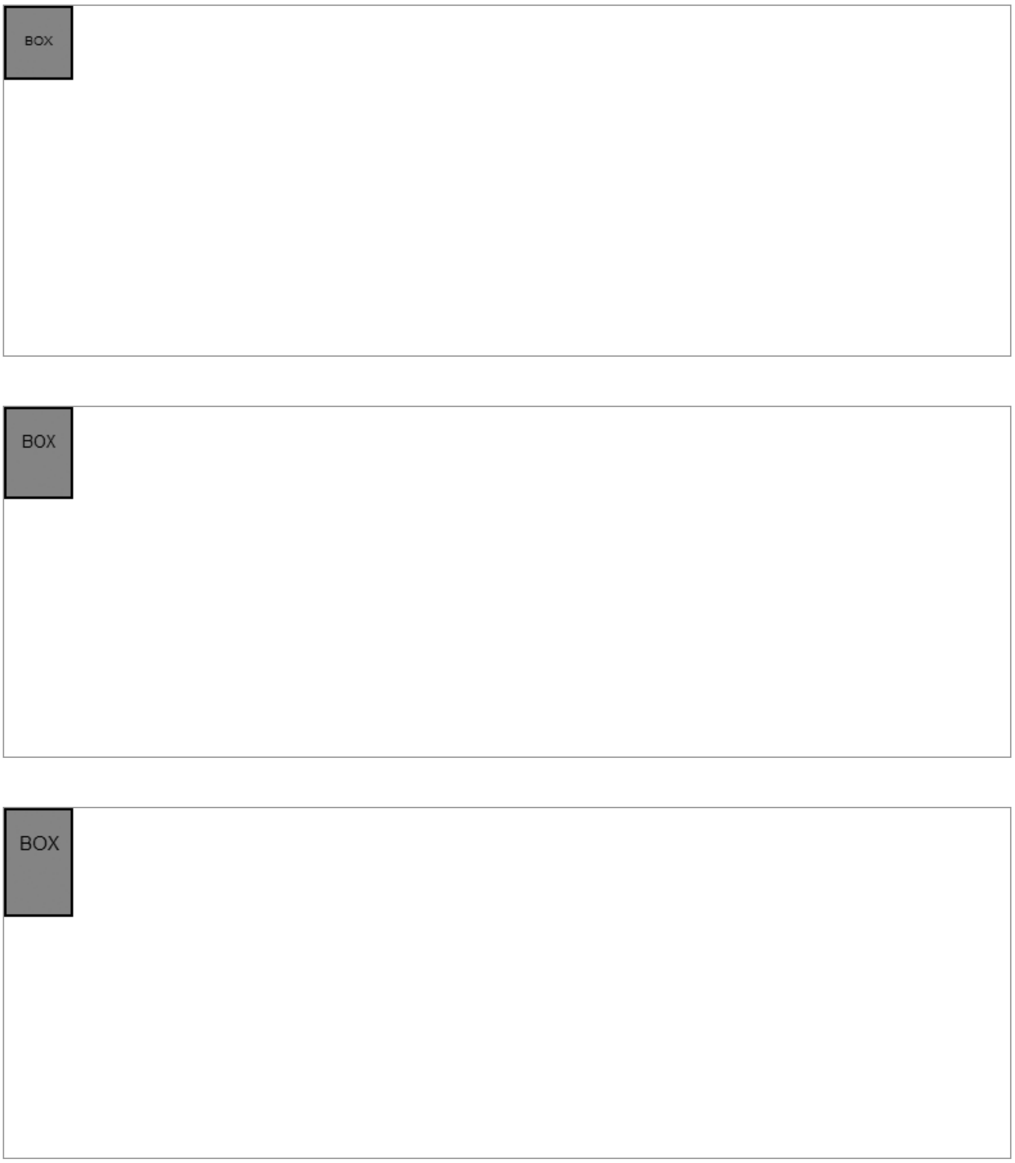


Figure 10-11. *Effect.Scale can be told to animate only height or width.*

We can also restrict the scaling to one dimension: `scaleX` and `scaleY` both default to `true`, but setting one to `false` will prevent it from growing along that axis. (Setting *both* to `false` would be plainly silly.)

Effect.Highlight

`Effect.Highlight` simplifies a common use case for effects: the pulse-like animation of an element's background color, otherwise known as the "yellow fade technique." Popularized by 37signals's web apps, the effect is an elegant, subtle way to call attention to a region of the page that has changed—for instance, as the result of an Ajax call. Figure 10-12 shows the technique.

```
new Effect.Highlight('box');
```

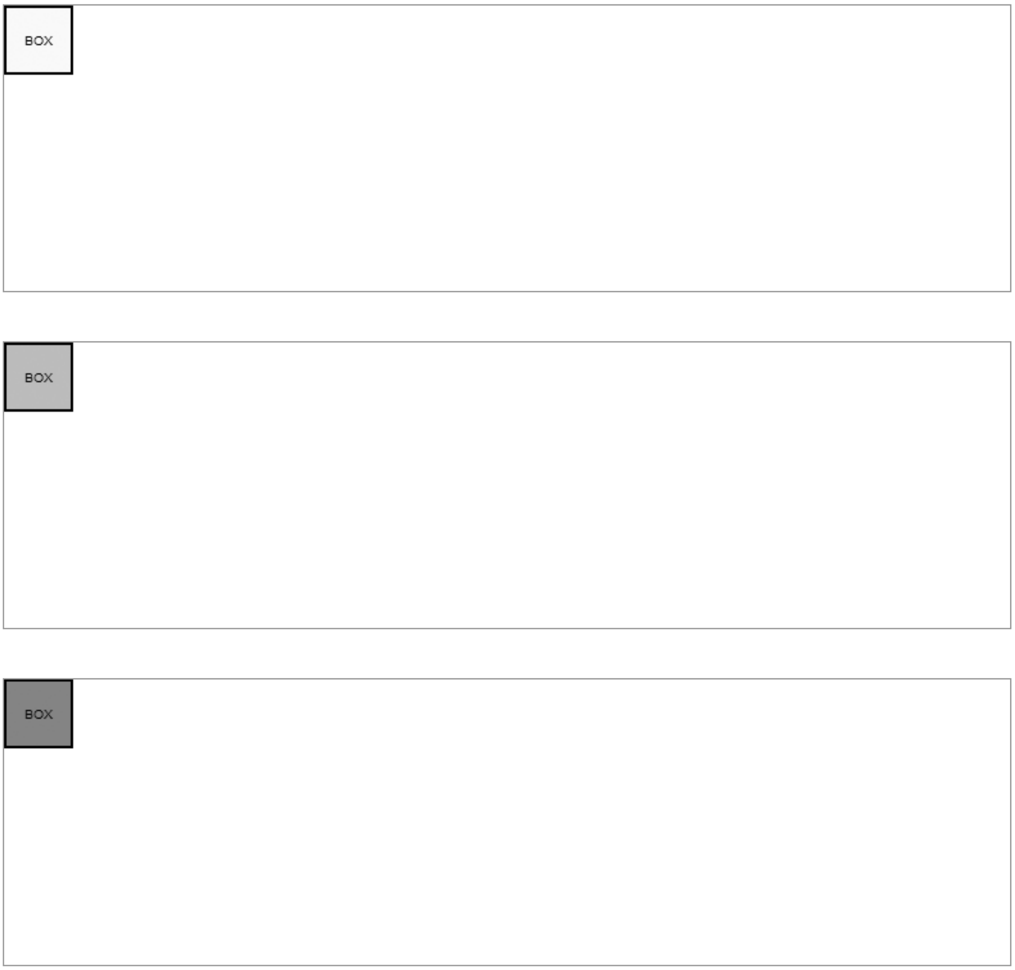


Figure 10-12. *Effect.Highlight "pulses" a background color on an element.*

As you might expect, the default highlight color is a light shade of yellow, but the parameters `startcolor` and `endcolor` let us set the colors for the first and last frames of the effect, respectively; and `restorecolor` lets us set the color that the element will become after the effect is complete (see Figure 10-13):

```
new Effect.Highlight('box',
 { startcolor: "#ffffff", endcolor: "#000000", restorecolor: "#999999" });
```

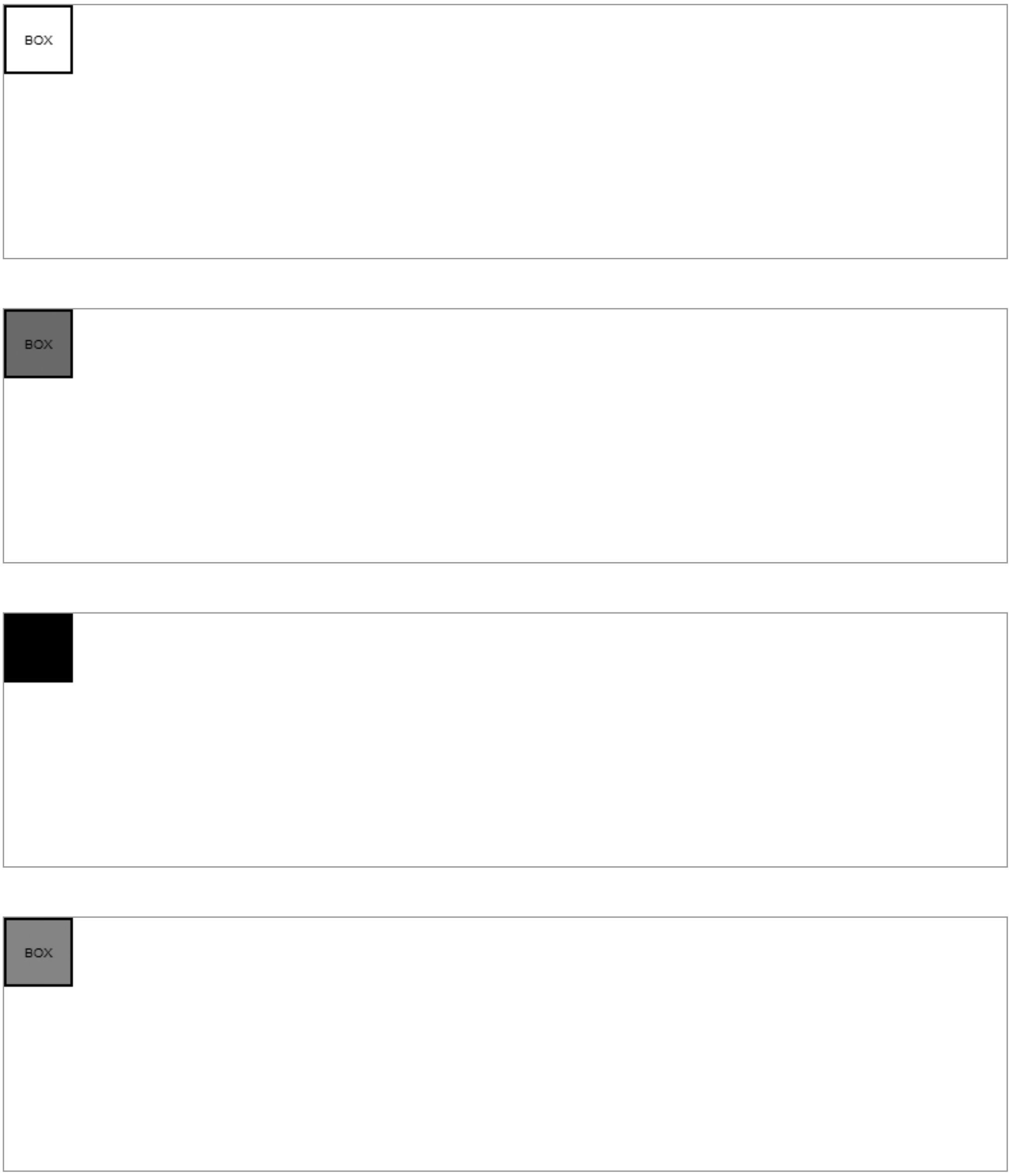


Figure 10-13. *Effect.Highlight with custom colors*

Effect.ScrollTo

`Effect.ScrollTo` is far more focused than the other core effects. It animates the scrolling of the browser viewport to bring the specified element into view. In other words, it's a time-lapse version of Prototype's `Element#scrollTo` function.

To illustrate this effect, let's change the CSS so that our box is "below the fold" on page load. My viewport isn't very tall, so bumping it down 500 pixels will do it. Let's also change the height of the page so that we'll have some space below the box:

```
/* CSS: */
#box {
  position: absolute;
  width: 50px;
  height: 50px;
  top: 500px;
  left: 0;
  background-color: #999;
  border: 2px solid #000;
  font-family: "Helvetica Neue";
  text-transform: uppercase;
  text-align: center;
  line-height: 50px;
  font-size: 10px;
}

body {
  height: 1500px;
}

/* JS: */new Effect.ScrollTo('box');
```

This effect, shown in Figure 10-14, simulates the "smooth scrolling" behavior that's now used in many applications. It helps the reader jump to a different part of the page without losing her original position.

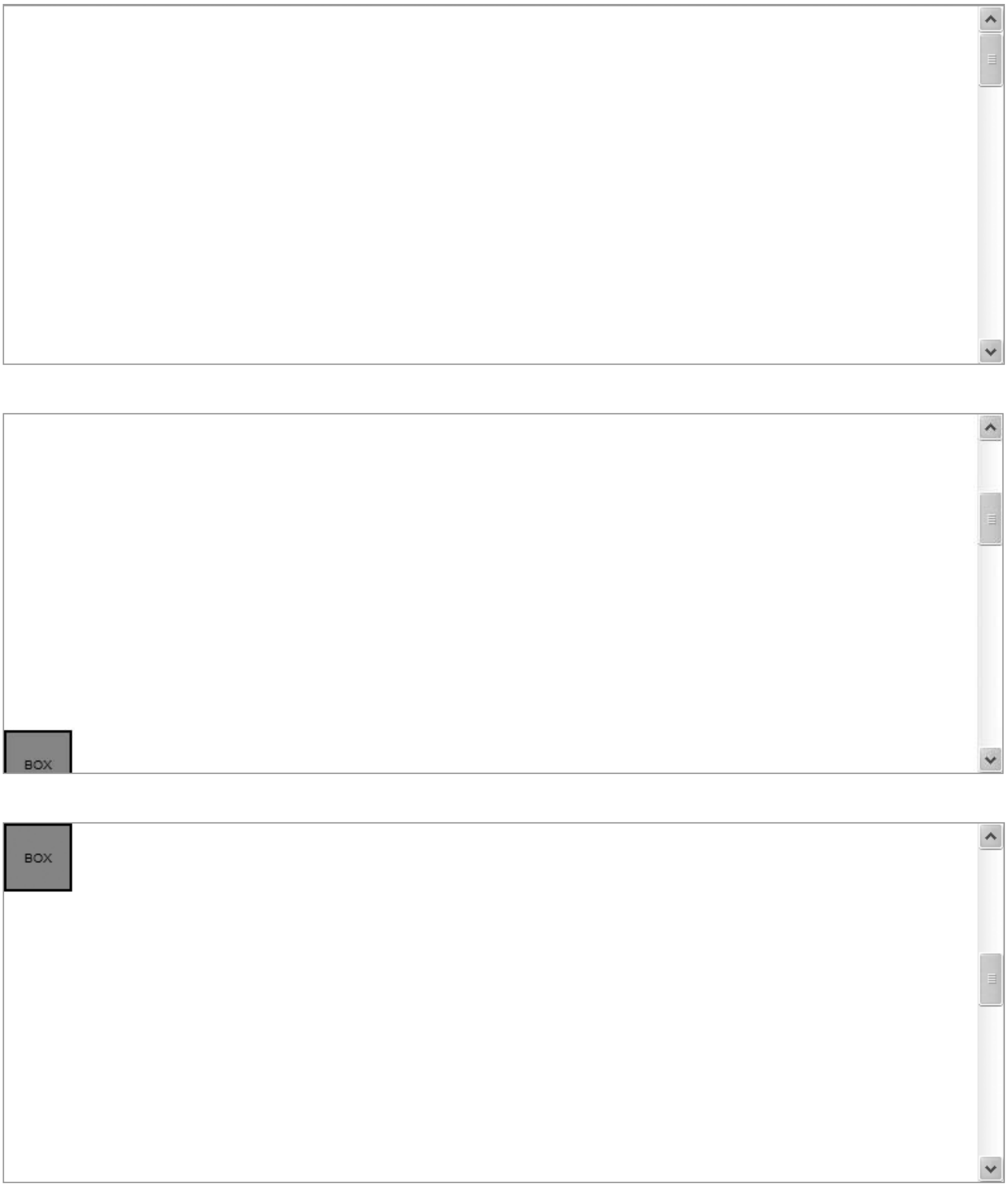


Figure 10-14. *Effect.ScrollTo animates the viewport's scroll offset.*

Introduction to Combination Effects

We've only scratched the surface of script.aculo.us effects. As I mentioned earlier, the true power lies in writing *combination effects*—groups of core effects that run in parallel to create more complex animations. You'll get to write your own effects later on, but first let's look at some of the combination effects given to you out of the box.

The first three examples of combination effects illustrate different ways to animate hiding and showing elements—each pair is a time-lapse version of `Element#hide` and `Element#show`. There are many different ways to get from visible to invisible.

Effect.Fade and Effect.Appear

The first pair, `Effect.Fade` and `Effect.Appear`, animate the opacity of the element. Thus, `Effect.Fade` will decrease the element's opacity until it's invisible, and then hide it; and `Effect.Appear` will show the element fully transparent, and then increase opacity gradually until it fades into view (see Figure 10-15).

```
new Effect.Fade("box");
```

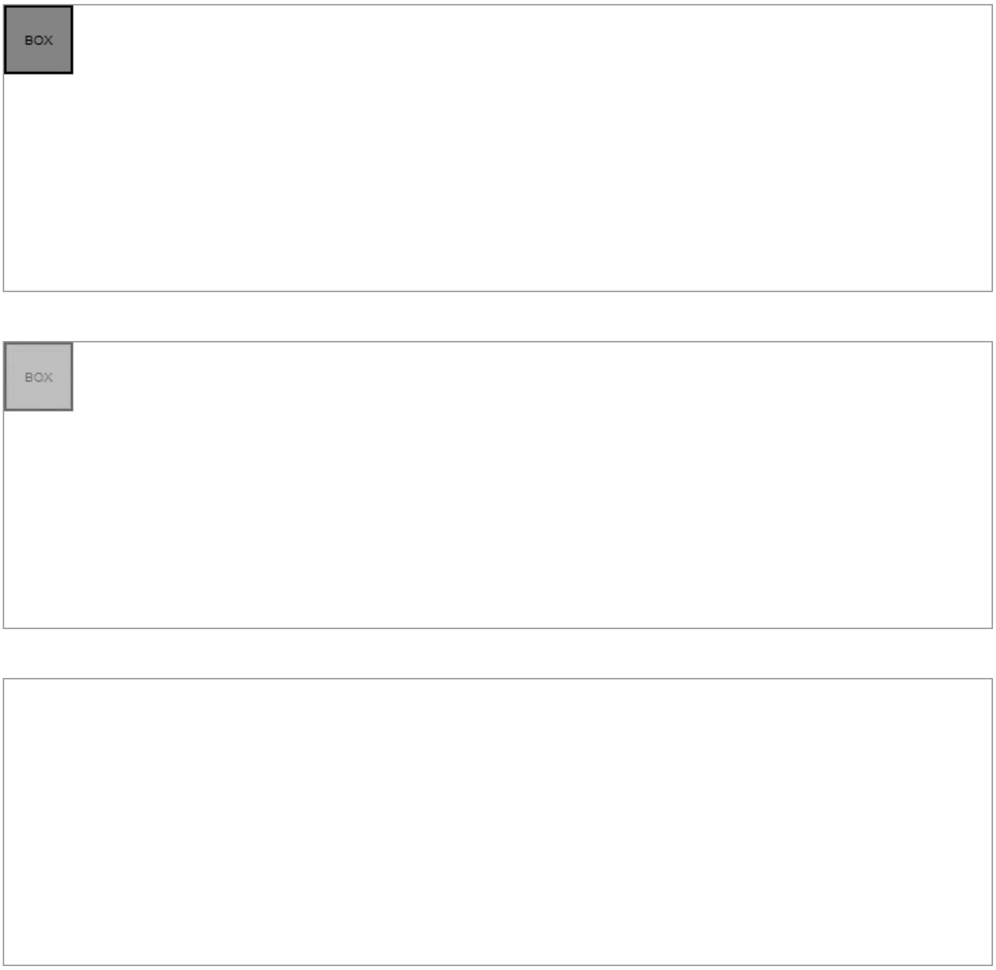


Figure 10-15. *Effect.Fade decreases an element's opacity until it's invisible.*

Naturally, `Effect.Fade` communicates something “fading away” (like an old soldier), so it’s best used on items that will disappear and not be used again. Likewise, `Effect.Appear` suggests the creation of something new, so it’s best used on items that haven’t been shown on the page before.

Effect.BlindUp and Effect.BlindDown

The next pair, `Effect.BlindUp` and `Effect.BlindDown`, work like the window blinds they’re named after. Calling `Effect.BlindUp` will cover the element vertically, line by line, starting with the *bottom* of the element (see Figure 10-16). `Effect.BlindDown` is the same animation in reverse.

```
new Effect.BlindUp('box');
```

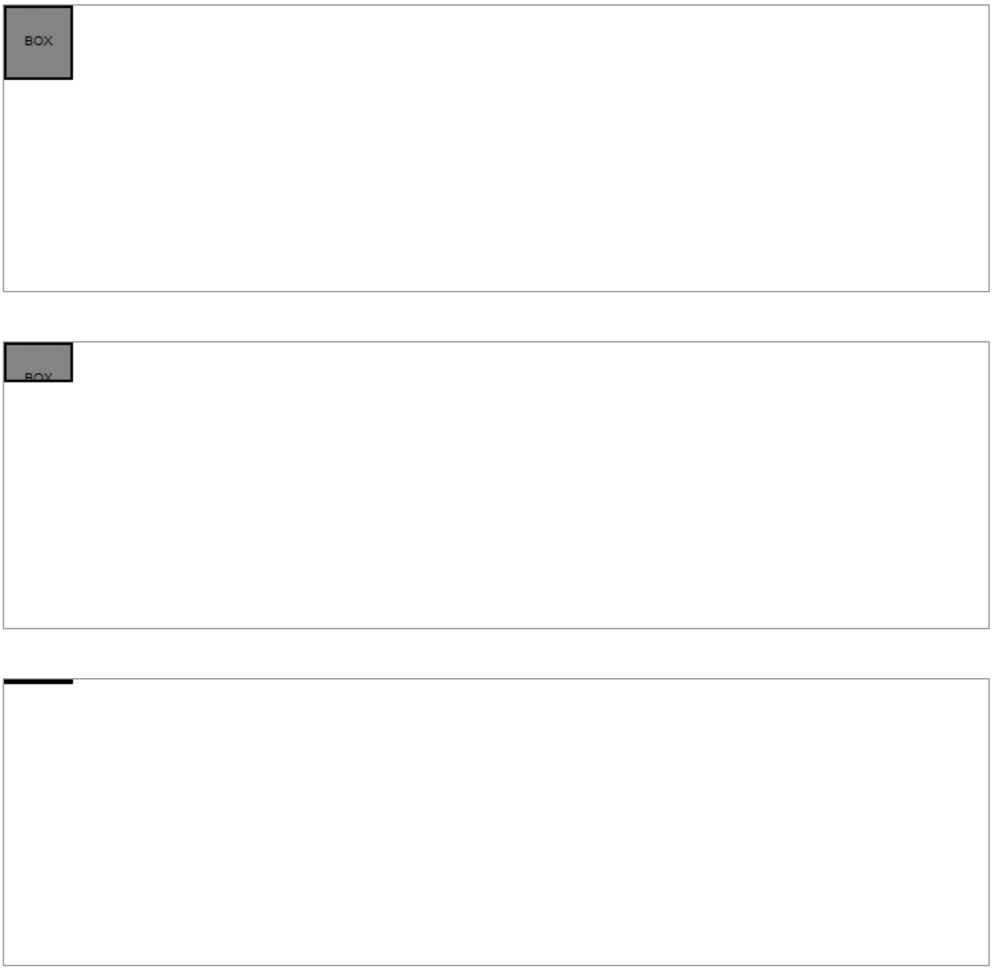


Figure 10-16. *Effect.BlindUp hides an element by covering up a progressively larger part of the element, starting from the bottom.*

`Effect.BlindUp` suggests that the element has simply been covered up; thus, it feels less “permanent” than `Effect.Fade`. Use this pair of effects for items that can be hidden, but can also be shown again.

Effect.SlideUp and Effect.SlideDown

Finally, `Effect.SlideUp` and `Effect.SlideDown` work like dresser drawers. Calling `Effect.SlideUp` will hide the element vertically, line by line, starting with the *top* of the element (see Figure 10-17). `Effect.SlideDown` is the same animation in reverse.

These effects carry one caveat: to work properly, they require the element in question be double-wrapped in a block-level element. Here, we’re using a `div`, so we’ll place another `div` inside it:

```
<!-- HTML: -->
<div id="box"><div>Box</div></div>

/* JS: */
new Effect.SlideUp('box');
```

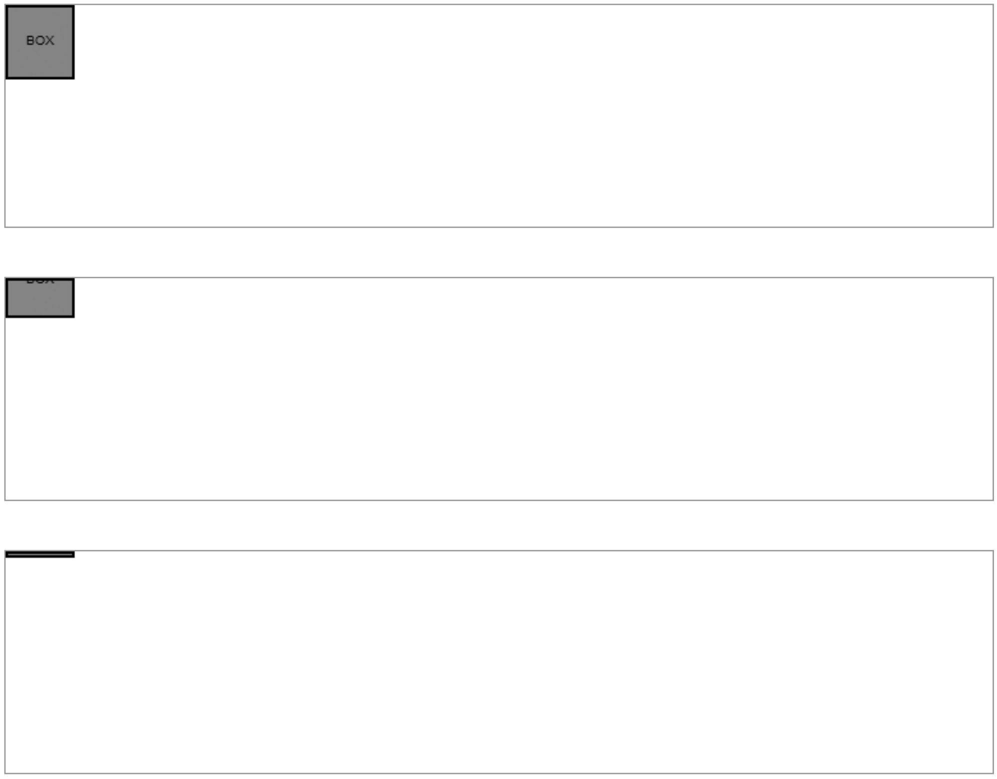


Figure 10-17. *Effect.SlideUp hides an element by “pushing” it from the bottom and covering it from the top.*

`Effect.SlideUp` suggests that the element is *disappearing* into the element right above it. So this pair of effects is useful for collapsible panels, accordion menus, and even things like expandable trees.

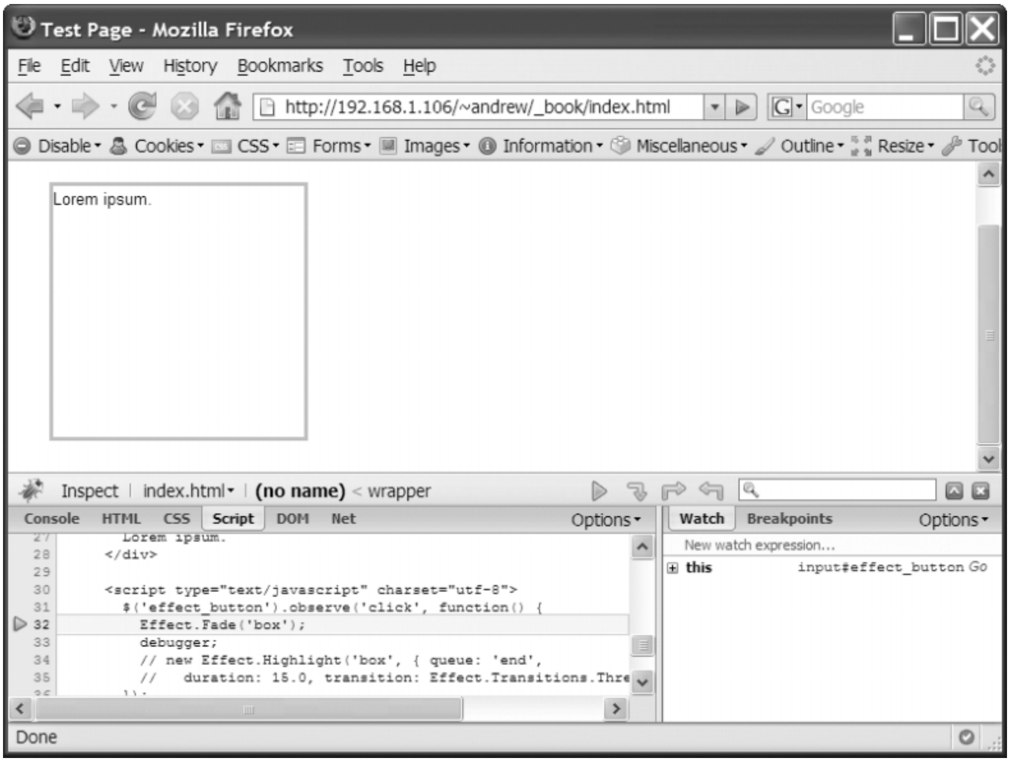
Effects Are Asynchronous

We've already covered how JavaScript has no `sleep` statement, so it can't halt all execution. Remember how effects work at their core: a function is called, over and over again, via `setTimeout`. Each call advances the animation by one frame. But any other code can run in the "gaps" between these "scheduled" animation frames.

To illustrate, we can start an effect, and then invoke the Firebug debugger with the special `debugger` keyword. Script execution will be paused so that we can step through the code.

```
Effect.Fade('box'); debugger;
```

Figure 10-18 shows the state of the page when the debugger starts.


Test Page - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/_book/index.html
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tool
Lorem ipsum.
Inspect | index.html | (no name) < wrapper
Console HTML CSS Script DOM Net Options
Watch Breakpoints Options
New watch expression...
Done

```
      Lorem ipsum.
    </div>

    <script type="text/javascript" charset="utf-8">
      $('effect_button').observe('click', function() {
        Effect.Fade('box');
        debugger;
        // new Effect.Highlight('box', { queue: 'end',
        //   duration: 15.0, transition: Effect.Transitions.Thre
```

```
this    input#effect_button Go
```

Figure 10-18. *The debugger has paused execution before our effect has finished.*

The element we told to fade out is still visible—it's only just begun animating. That's because the interpreter doesn't wait around; it schedules the frames of the animation, and then moves on to the next line.

script.aculo.us gives us two ways around this. The first way is using *effect callbacks*, which are very similar to the Ajax callbacks you learned about in Chapter 4. The second is using *effect queues*, which can force effects to wait their turn behind other effects.

Callbacks

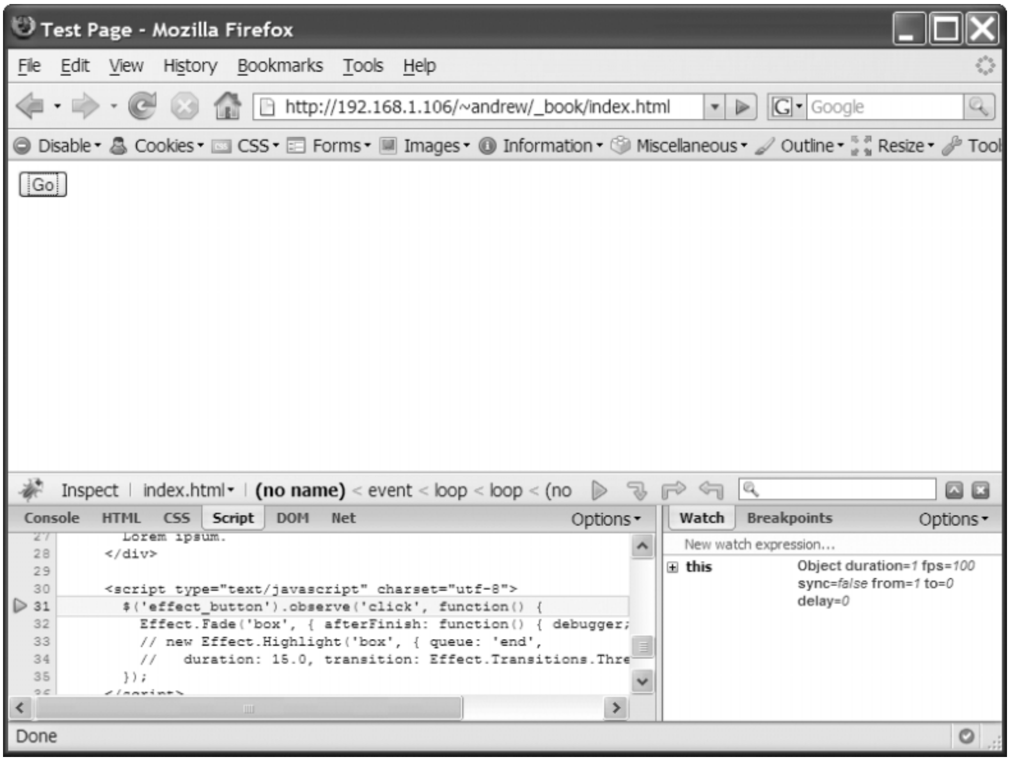
Both core effects and combination effects treat certain parameters passed into the `options` argument as callbacks:

- `beforeSetup` and `afterSetup` are called before and after the "setup" stage of an effect—in which the effect makes any necessary modifications to the element itself.
- `beforeStart` and `afterStart` are called before and after the first frame of an effect, respectively.
- `beforeUpdate` and `afterUpdate` are called before and after *each* frame of an effect.
- `beforeFinish` and `afterFinish` are called before and after the "finish" stage of an effect, in which the effect reverts any changes it made in the "setup" stage.

To ensure that our `alert` dialog appears after the effect is complete, we can use the `afterFinish` callback:

```
Effect.Fade('box', {
  afterFinish: function(effect) { debugger; }
});
```

Figure 10-19 shows that the element is completely invisible before the debugger starts.


Test Page - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/_book/index.html
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tool
Go
Inspect index.html (no name) < event < loop < loop < (no
Console HTML CSS Script DOM Net Options
Watch Breakpoints Options
New watch expression...
Done

```
    Lorem ipsum.
  </div>

  <script type="text/javascript" charset="utf-8">
    $('effect_button').observe('click', function() {
      Effect.Fade('box', { afterFinish: function() { debugger;
      // new Effect.Highlight('box', { queue: 'end',
      //   duration: 15.0, transition: Effect.Transitions.Thre
    });
```

```
this    Object duration=1 fps=100
        sync=false from=1 to=0
        delay=0
```

Figure 10-19. *By moving the debugger statement to a callback, we ensure that it runs after the effect has finished.*

Taken together, these callbacks provide hooks into every single frame of an effect. They're the ultimate override. Most of the time, `afterFinish` will be all you need, but it's nice to know the rest are there.

Each callback takes the effect instance itself as the first argument. Since the `Effect.Base` class (from which all effects derive) defines some instance properties, you're able to use these same properties in your callbacks:

```
Effect.Fade('box', {
  afterFinish: function(effect) {
    effect.element.remove();
  }
});
```

You may also use several other properties: `options` to access the effect's options, `startOn` and `finishOn` to access the integer timestamps that mark the effect's duration, and `currentFrame` to access the number of the last frame rendered.

Queues

Effect queues address a subset of the collision problem described earlier. Let's wire up a button to our box—one that will fade the box out when clicked:

```
<input type="button" name="go" value="Go" id="effect_button" />

<div id="box">
  Lorem ipsum.
</div>

<script type="text/javascript" charset="utf-8">
  $('effect_button').observe('click', function() {
    new Effect.Highlight('box');
  });
</script>
```

Clicking the button pulses the box, just like we'd expect. But clicking twice rapidly fires *two* highlight effects—they get tangled almost immediately, leaving the element permanently yellow (see Figure 10-20).

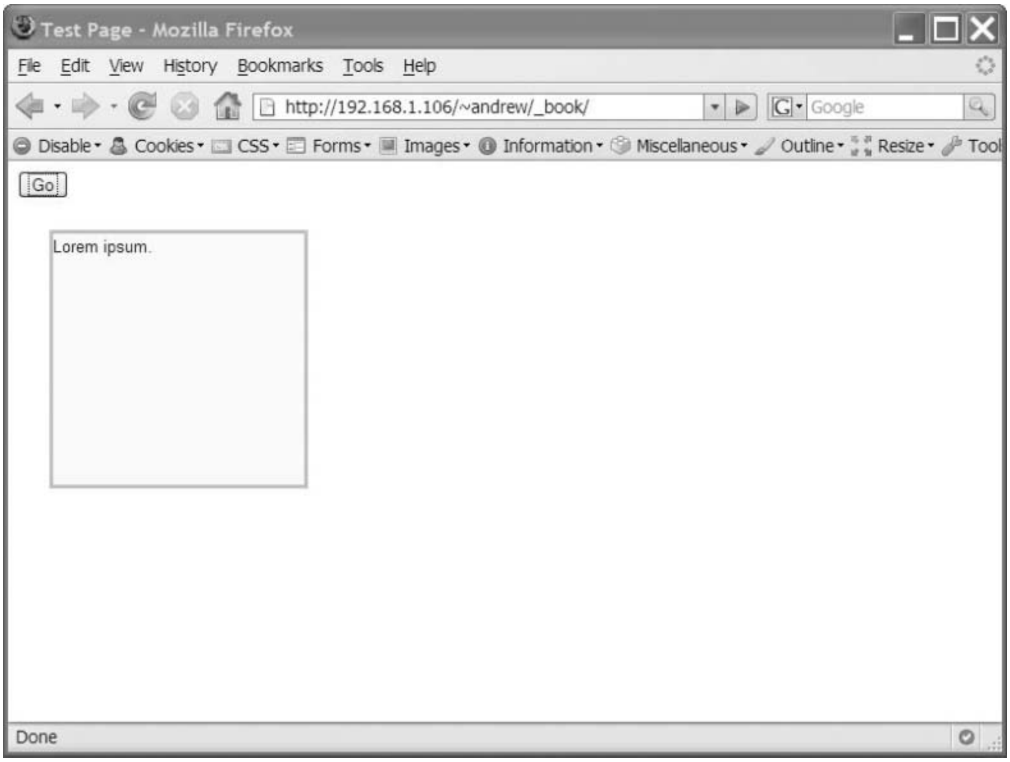

Test Page - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/_book/
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tool
Go
Lorem ipsum.
Done

Figure 10-20. *Triggering Effect.Highlight twice in rapid succession leaves the element with a permanent yellow background color.*

To be fair, it's only doing what you tell it to do. By default, an effect starts animating as soon as it's called. But the savvy developer can manage the state of an effect—telling it when to wait and when not to run at all.

When we click the button twice, we want the first effect to start right away, but the second effect to start *once the first is done*. The `queue` option lets us do so:

```
new Effect.Highlight('box', { queue: 'end' });
```

The `end` value merely says to place the effect at the end of whatever effect is already running on the page, rather than the default `parallel`. A third option, `front`, *halts* whatever effect is already playing in order to give the new effect priority.

Now we can click the button as much as we want—if we click it ten times in the span of a second, we'll see ten highlight effects fire patiently, one after the other.

Putting It All Together

We're going to go through one last example in this chapter. It pulls in some of the work we did in previous chapters and adds an effects-inspired garnish.

In Chapters 4 and 5, we wrote the code to simulate a data provider—one that supplies ever-changing statistics for our fictional fantasy football game. We hooked it up to an Ajax poller that checks for new scores every 30 seconds and fires a custom event whenever it gets a response. We laid all this groundwork without a clear example of when we'd be able to use it next.

Similarly, in Chapter 7 we wrote a utility class for adding up numbers. The use case we had in mind was a data table in which each row had a numeric value—and a "total" row in the table footer that would show the sum of the values from all rows.

Effects give us the final piece. We're going to build a game summary page for our fantasy football league. It will display both teams' scores, side by side, with a breakdown by player of where the points are coming from.

Writing the Markup

First, let's get this all onto the page in the form of markup. Then we can style it to aesthetic perfection.

Create a new file called `game.html` with `script` tags pointing to `prototype.js` and `scriptaculous.js`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">

    <title>Game Page</title>

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

  </head>
  <body>

  </body>
</html>
```

In the body of the page, we'll place some `div`s to serve as rough guides for where our elements will go:

```
<div id="wrapper">
  <h1>Box Score</h1>

  <div id="teams">

    <div id="team_1_container">
      <h2>The Fighting Federalists</h2>

      <!-- TABLE GOES HERE -->

    </div> <!-- #team_1_container -->

    <div id="team_2_container">
      <h2>Washington's Generals</h2>

      <!-- TABLE GOES HERE -->

    </div> <!-- #team_2_container -->


  </div> <!-- #teams -->

</div> <!-- #wrapper -->
```

Finally, let's look at the data table itself. It's going to have four columns: position, name, score, and summary (a short text description of the player's stats). All these pieces of data exist in the JSON feed *except* for the player's name; instead, we'll be using our shorthand to refer to players by position (e.g., `QB`, `WR1`, `RB2`).

So one table might look like this:

```
<table id="team_1">
  <thead>
    <tr>
      <!-- We can't fit the word "position" in this column, so let's
           abbreviate and put the full word inside a "title" attribute. -->
      <th class="pos" title="Position">Pos.</th>
      <th>Name</th>
      <th>Stats</th>
      <th class="score">Points</th>
    </tr>
  </thead>

  <!-- In accordance with the HTML spec, the TFOOT occurs _before_ the TBODY
       in the markup, even though it's placed _after_ the TBODY visually. -->
  <tfoot>
    <tr>
      <td colspan="3" class="total">Total</td>
       <!-- This table cell will display the total. It has an ID so that we
            can grab it easily. -->
      <td id="team_1_total" class="score"></td>
    </tr>
  </tfoot>

  <tbody>

    <!-- Each table row has the position shorthand (RB1, WR2, etc.) as a
         class name. Table cells have class names as hooks for both scripting
         and styling. -->
    <tr class="QB">
      <td class="pos">QB</td>
      <td>Alexander Hamilton</td>
      <td class="summary"></td>
      <td class="score">0</td>
    </tr>
    <tr class="RB1">
      <td class="pos">RB</td>
      <td>James Madison</td>
      <td class="summary"></td>
      <td class="score">0</td>
    </tr>
    <tr class="RB2">
      <td class="pos">RB</td>
      <td>John Jay</td>
      <td class="summary"></td>
      <td class="score">0</td>
    </tr>
    <tr class="WR1">
      <td class="pos">WR</td>
      <td>John Marshall</td>
      <td class="summary"></td>
      <td class="score">0</td>
    </tr>
    <tr class="WR2">
      <td class="pos">WR</td>
      <td>Daniel Webster</td>
      <td class="summary"></td>
      <td class="score">0</td>
    </tr>
    <tr class="TE">
      <td class="pos">TE</td>
      <td>Samuel Chase</td>
      <td class="summary"></td>
      <td class="score">0</td>
    </tr>
  </tbody>
</table>
```

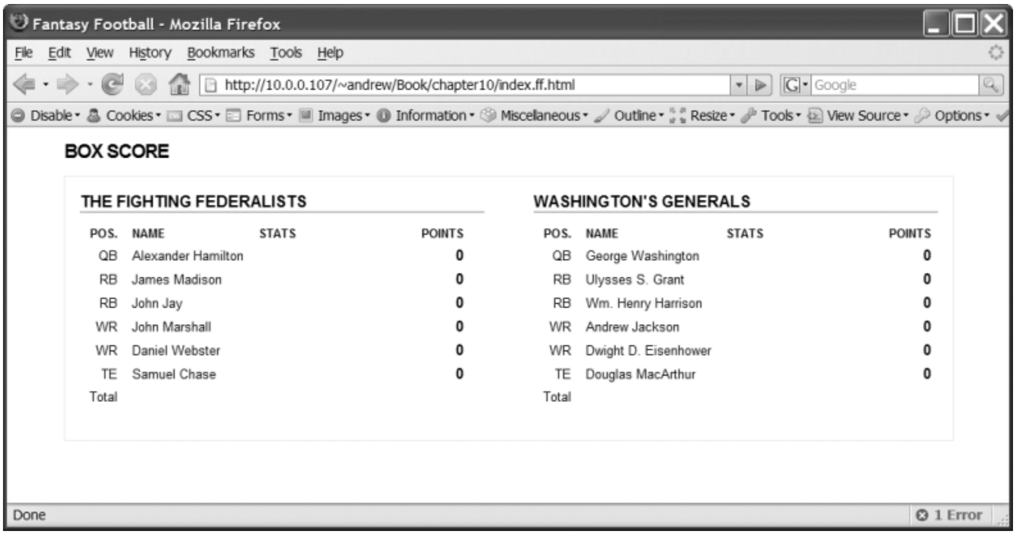
We'll place two such tables into our page—one for each team.

Adding Styles

This page is begging for some styling. Create a new file, `styles.css`, and link to it from `game.html`:

```
<head>
  <meta http-equiv="Content-type" content="text/html; charset=utf-8">

  <title>Game Page</title>

  <link rel="stylesheet" type="text/css" href="styles.css" />

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

</head>
```

Now you're free to style this page however you like. Let your inner designer run wild. I'm going to go with something minimal (see Figure 10-21).


Fantasy Football - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://10.0.0.107/~andrew/Book/chapter10/index.ff.html
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tools View Source Options
Done
1 Error

BOX SCORE

THE FIGHTING FEDERALISTS

POS.	NAME	STATS	POINTS
QB	Alexander Hamilton		0
RB	James Madison		0
RB	John Jay		0
WR	John Marshall		0
WR	Daniel Webster		0
TE	Samuel Chase		0
Total			

WASHINGTON'S GENERALS

POS.	NAME	STATS	POINTS
QB	George Washington		0
RB	Ulysses S. Grant		0
RB	Wm. Henry Harrison		0
WR	Andrew Jackson		0
WR	Dwight D. Eisenhower		0
TE	Douglas MacArthur		0
Total			

Figure 10-21. *The scoring page with styles applied*

The technical details aren't very interesting; I changed the default font, sized the tables so that they appear side by side, and styled the "score" table cells to stand out.

Bringing in Help

Now we're going to search back through the examples from earlier chapters. We'll be salvaging three files in all. The first two, `scores.php` and `score_broadcaster.js`, were created in Chapter 5. The third, `totaler.js`, was created in Chapter 7. Move all three to the same directory as `game.html`.

As you may remember, `score_broadcaster.js` is our beacon in the sky: it talks to our "server" (the `scores.php` file) and fires a custom event every 30 seconds with the latest player data. Soon we'll write code to *listen* for this event and act upon it.

The third file will handle the boring job of calculating the totals for each team. The `Totaler` class we wrote is perfectly suited to summing and resumming each team's point totals.

First, include the two JavaScript files in the document head. Be sure to include them *after* Prototype and script.aculo.us.

```
<head>
  <meta http-equiv="Content-type" content="text/html; charset=utf-8">

  <title>Game Page</title>

  <link rel="stylesheet" type="text/css" href="styles.css" />

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

  <script src="score_broadcaster.js" type="text/javascript"></script>
  <script src="totaler.js" type="text/javascript"></script>

</head>
```

Finally, we'll write about 20 lines of code in another `script` block that will handle the logic of updating our player tables.

Bells and Whistles

Now we're going to write some page-specific code to wire all this together. It will update the player table with the raw JSON data, highlight those rows that have changed, and finally tell the tables to recompute their totals.

Start with the following code in your document head:

```
<script type="text/javascript" charset="utf-8">
  document.observe("dom:loaded", function() {
    var team1totaler = new Totaler('team_1', 'team_1_total', {
     selector: 'tbody .score' });
    var team2totaler = new Totaler('team_2', 'team_2_total', {
     selector: 'tbody .score' });
  });
</script>
```

On the `dom:loaded` event (the custom event that fires as soon as the DOM can be modified) we're declaring two instances of `Totaler`—one for each table. The `selector` option is a CSS selector that identifies all the cells we want to add together.

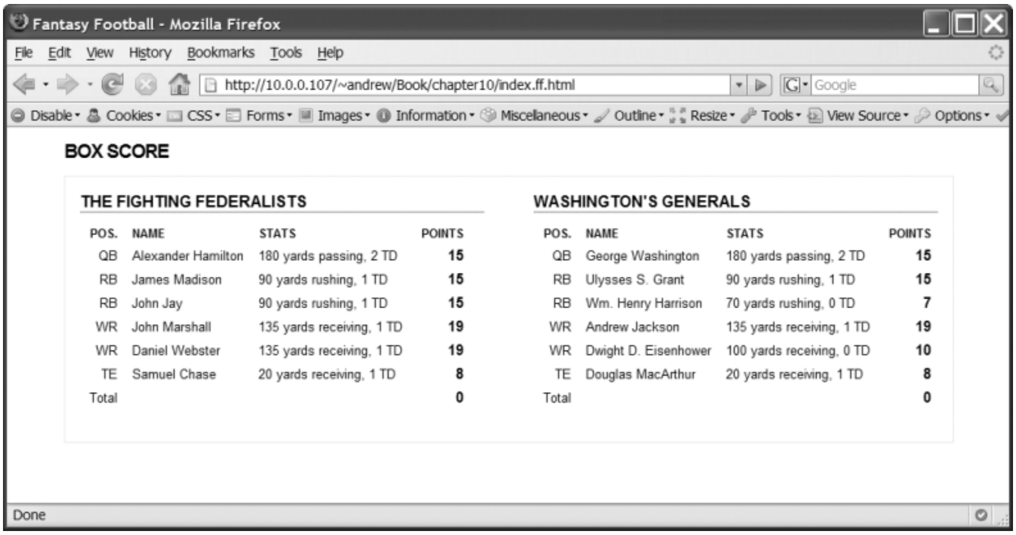
The next step is to write the logic to iterate through the data and update each table. The JSON we receive will be divided up by team, so think of this as taking the data, breaking it in half, and feeding one half to the first table and the other half to the second. Both sides need the same logic, so let's write a function that expects *one* table and *one* team's stats.

```
<script type="text/javascript" charset="utf-8">
  function updateTeam(table, json) {
    table = $(table);

    // a team is divided into several positions
    var positionStats, row;
    for (var position in json) {
      positionStats = json[position];

      // match up the JSON property name (WR1, RB2, TE, etc.) with the
      // table row that has the corresponding class name
      row = table.down('tr.' + position);

      // update the score cell with the player's point total from the JSON
      row.down('td.score').update(positionStats.points);
    }
  }

  document.observe("dom:loaded", function() {
    var team1totaler = new Totaler('team_1', 'team_1_total', {
     selector: 'tbody .score' });
    var team2totaler = new Totaler('team_2', 'team_2_total', {
     selector: 'tbody .score' });
  });
</script>
```

Remember the `score:updated` custom event? That's the indicator that new stats have arrived. So let's listen for that event:

```
<script type="text/javascript" charset="utf-8">
  function updateTeam(table, json) {
    table = $(table);
    // a team is divided into several positions
    var positionStats, row;
    for (var position in json) {
      positionStats = json[position];

      // match up the JSON property name (WR1, RB2, TE, etc.) with the
      // table row that has the corresponding class name
      row = table.down('tr.' + position);

      // update the score cell with the player's point total from the JSON
      row.down('td.score').update(positionStats.points);
    }
  }

  document.observe("dom:loaded", function() {
    var team1totaler = new Totaler('team_1', 'team_1_total', {
     selector: 'tbody .score' });
    var team2totaler = new Totaler('team_2', 'team_2_total', {
     selector: 'tbody .score' });

     document.observe("score:updated", function() {
       // the "memo" property holds the custom data we attached to the event
       var json = event.memo;

       // break the JSON in half -- one piece for each table.
       updateTeam('table_1', json.team_1);
       updateTeam('table_2', json.team_2);
     });
  });
</script>
```

Let’s see if this works the way we expect. Open up `game.html` in your browser. It should look like Figure 10-22.


Fantasy Football - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://10.0.0.107/~andrew/Book/chapter10/index.ff.html
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tools View Source Options
Done

BOX SCORE

THE FIGHTING FEDERALISTS

POS.	NAME	STATS	POINTS
QB	Alexander Hamilton	180 yards passing, 2 TD	15
RB	James Madison	90 yards rushing, 1 TD	15
RB	John Jay	90 yards rushing, 1 TD	15
WR	John Marshall	135 yards receiving, 1 TD	19
WR	Daniel Webster	135 yards receiving, 1 TD	19
TE	Samuel Chase	20 yards receiving, 1 TD	8
Total			0

WASHINGTON'S GENERALS

POS.	NAME	STATS	POINTS
QB	George Washington	180 yards passing, 2 TD	15
RB	Ulysses S. Grant	90 yards rushing, 1 TD	15
RB	Wm. Henry Harrison	70 yards rushing, 0 TD	7
WR	Andrew Jackson	135 yards receiving, 1 TD	19
WR	Dwight D. Eisenhower	100 yards receiving, 0 TD	10
TE	Douglas MacArthur	20 yards receiving, 1 TD	8
Total			0

Figure 10-22. *The players' up-to-date point totals are now shown in the table.*

You'll have to wait at least 30 seconds (one interval) to confirm that the scores are updating with each Ajax response—but it should work. There are two problems, though. First, unless you're looking right at a row, you won't notice when its point value changes. Second, the totals at the bottom of each table aren't updating.

Let's take the second problem first. We can fix it easily by using the `updateTotal` instance method that we wrote when we created `Totaler`. Once the `updateTeam` function has set the new data, we'll tell our `Totaler` instances to recompute the sums:

```
<script type="text/javascript" charset="utf-8">
  function updateTeam(table, json) {
    table = $(table);

    // a team is divided into several positions
    var positionStats, row;
    for (var position in json) {
      positionStats = json[position];

      // match up the JSON property name (WR1, RB2, TE, etc.) with the
      // table row that has the corresponding class name
      row = table.down('tr.' + position);
      // update the score cell with the player's point total from the JSON
      row.down('td.score').update(positionStats.points);
    }
  }

  document.observe("dom:loaded", function() {
    var team1totaler = new Totaler('team_1', 'team_1_total', {
     selector: 'tbody .score' });
    var team2totaler = new Totaler('team_2', 'team_2_total', {
     selector: 'tbody .score' });

     document.observe("score:updated", function() {
       // the "memo" property holds the custom data we attached to the event
       var json = event.memo;

       // break the JSON in half -- one piece for each table.
       updateTeam('table_1', json.team_1);
       updateTeam('table_2', json.team_2);

       team1totaler.updateTotal();
       team2totaler.updateTotal();
     });
  });
</script>
```

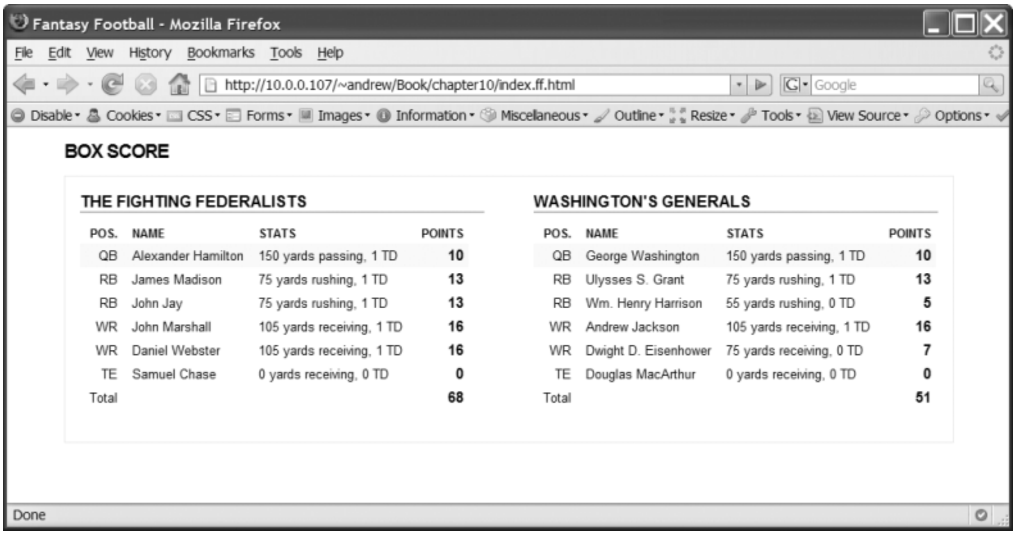
In other words, recomputing the total needs to be the last thing we do when a score updates.

Now we need to emphasize rows when they change. This is the perfect use case for `Effect.Highlight`—the effect that “pulses” an element’s background color to draw attention to it. For this, we’ll need to add some logic to our `updateTeam` function:

```
<script type="text/javascript" charset="utf-8">
  function updateTeam(table, json) {
    table = $(table);

    // a team is divided into several positions
    var positionStats, row;
    for (var position in json) {
      positionStats = json[position];
      // match up the JSON property name (WR1, RB2, TE, etc.) with the
      // table row that has the corresponding class name
      row = table.down('tr.' + position);

      var scoreCell = row.down('td.score'), oldValue = Number(scoreCell.innerHTML);
      // update the score cell with the player's point total from the JSON
      scoreCell.update(positionStats.points);

      // is the new value larger than the old value?
      if (position.points > oldValue) {
        new Effect.Highlight(row);
      }
    }
  }

  document.observe("dom:loaded", function() {
    var team1totaler = new Totaler('team_1', 'team_1_total', {
     selector: 'tbody .score' });
    var team2totaler = new Totaler('team_2', 'team_2_total', {
     selector: 'tbody .score' });

     document.observe("score:updated", function() {
       // the "memo" property holds the custom data we attached to the event
       var json = event.memo;

       // break the JSON in half -- one piece for each table.
       updateTeam('table_1', json.team_1);
       updateTeam('table_2', json.team_2);

       team1totaler.updateTotal();
       team2totaler.updateTotal();
     });
  });
</script>
```

Now, as we cycle through the table rows, we check whether the new point value for that row is greater than the current one. If so, we declare a new `Effect.Highlight` to draw attention to that row.

We're done! Reload the page and congratulate yourself for doing something awesome. Figure 10-23 shows the fruits of your labor.


Fantasy Football - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://10.0.0.107/~andrew/Book/chapter10/index.ff.html
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tools View Source Options
Done

BOX SCORE

THE FIGHTING FEDERALISTS

POS.	NAME	STATS	POINTS
QB	Alexander Hamilton	150 yards passing, 1 TD	10
RB	James Madison	75 yards rushing, 1 TD	13
RB	John Jay	75 yards rushing, 1 TD	13
WR	John Marshall	105 yards receiving, 1 TD	16
WR	Daniel Webster	105 yards receiving, 1 TD	16
TE	Samuel Chase	0 yards receiving, 0 TD	0
Total			68

WASHINGTON'S GENERALS

POS.	NAME	STATS	POINTS
QB	George Washington	150 yards passing, 1 TD	10
RB	Ulysses S. Grant	75 yards rushing, 1 TD	13
RB	Wm. Henry Harrison	55 yards rushing, 0 TD	5
WR	Andrew Jackson	105 yards receiving, 1 TD	16
WR	Dwight D. Eisenhower	75 yards receiving, 0 TD	7
TE	Douglas MacArthur	0 yards receiving, 0 TD	0
Total			51

Figure 10-23. *Rows are highlighted as their point values change.*

Summary

We've spent quite a few pages on effects for good reason: they're the meat and potatoes of script.aculo.us, and the only piece that's a dependency for all the rest. The script.aculo.us UI widgets all rely on the subtle and purposeful animations we've surveyed.

In fact, certain other parts of script.aculo.us allow for pluggable effects—for instance, if you don't like how an element animates when it's dragged and dropped, you may specify a custom or built-in effect to replace it. These parts of script.aculo.us will be covered in the next two chapters.

CHAPTER 11

Enabling Draggables, Droppables, and Sortables

Drag-and-drop is a UI pattern that seems to have been around since the mouse was invented, but until recently it wasn't used very often in web applications. But why not? The technological capabilities are there. DOM scripting lets us listen for the `mousedown`, `mousemove`, and `mouseup` events that comprise a drag. It also lets us modify the CSS positioning of an element so that it can "follow" the mouse pointer around the screen.

In this chapter, we'll look at the two low-level objects, `Draggable` and `Droppables`, provided by script.aculo.us for drag-and-drop. Then we'll look at a high-level object, `Sortable`, that adapts drag-and-drop for a specific task.

Exploring Draggables

In script.aculo.us, a *draggable* is anything that can, not surprisingly, be dragged around the page. There are a number of things you can customize about the drag itself, but the simplest way to make something draggable is to declare a new instance of the `Draggable` class:

```
new Draggable('some_element');
```

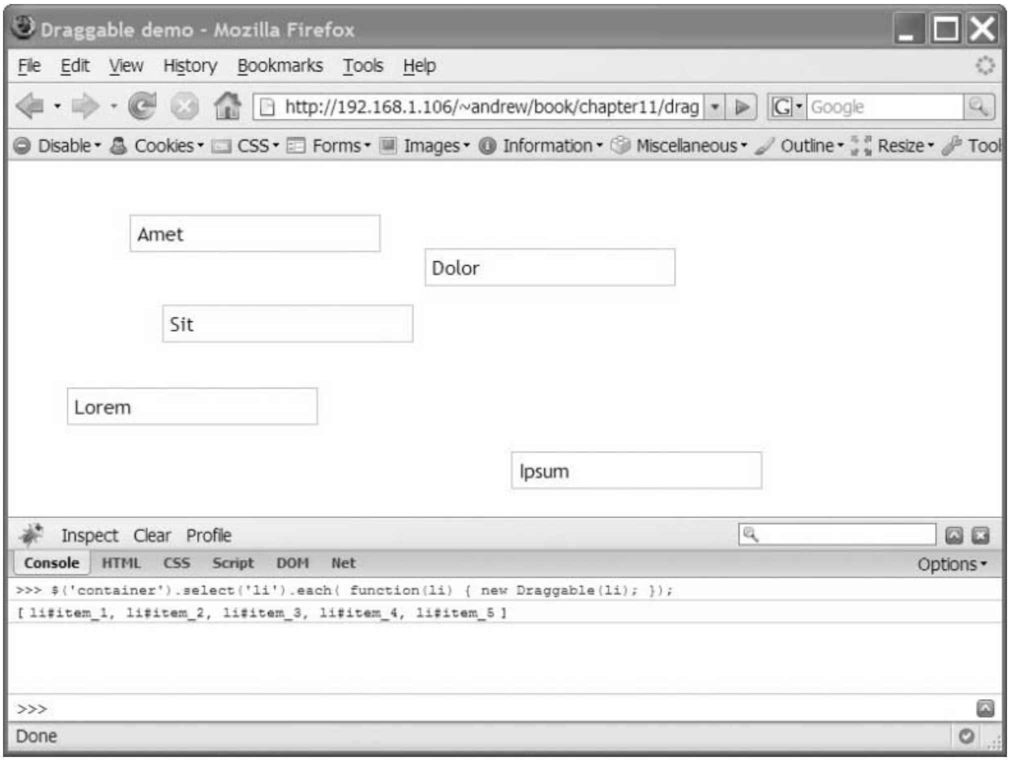
Let's create a sandbox for playing around with draggables. Create `draggable.html` and add this markup:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
    <title>Draggable demo</title>
    <style type="text/css" media="screen">
      body {
        font-family: 'Trebuchet MS', sans-serif;
      }

      #container {
        width: 200px;
        list-style: none;
        margin-left: 0;
        padding-left: 0;
      }

        #container li {
          border: 1px solid #ccc;

          margin: 10px 0;
          padding: 3px 5px;
        }
    </style>

    <script src="js/prototype.js" type="text/javascript"></script>
    <script src="js/scriptaculous.js" type="text/javascript"></script>
  </head>
  <body>

    <ul id="container">
      <li id="item_1">Lorem</li>
      <li id="item_2">Ipsum</li>
      <li id="item_3">Dolor</li>
      <li id="item_4">Sit</li>
      <li id="item_5">Amet</li>
    </ul>

  </body>
</html>
```

The bold section of markup represents the items we'll make draggable. Save this page, load it in Firefox, and then type the following into the Firebug console:

```
$$('container li').each( function(li) {
  new Draggable(li);
});
```

We're grabbing all the `li` elements within the `ul` and declaring each to be a draggable. Run this code, and you'll be able to drag the five list items anywhere on the page, as shown in Figure 11-1.


Draggable demo - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/book/chapter11/drag
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tool
Amet
Dolor
Sit
Lorem
Ipsum
Inspect Clear Profile
Console HTML CSS Script DOM Net
Options
Done

```
>>> $('container').select('li').each( function(li) { new Draggable(li); });
[ li#item_1, li#item_2, li#item_3, li#item_4, li#item_5 ]
>>>
```

Figure 11-1. *Declaring an instance of Draggable adds dragging behavior to an element.*

These draggables don't *do* anything yet, of course, but we'll address that when we go over droppables. This example tells you a few things about the `Draggable` class, but the fact that they can now be dragged around on the screen illustrates how the `Draggable` class works.

Making Draggables

Making draggables is easy. The `Draggable` constructor (the `initialize` method that gets called when you instantiate a class) takes care of assigning the events that control dragging; it also handles communication with any droppables it encounters.

The default options for `Draggable` enable an element to be moved anywhere on the page—but to make an element interact with potential drop targets, you'll have to specify some options of your own.

So let's add some options. Reload the page, and then run this code in the console:

```
$('container').select('li').each( function(li) {
  new Draggable(li, { revert: true });
});
```

Play around with the draggables and you'll notice the difference: they now *move back* (or *revert)* to their original positions at the end of the drag.

Another option controls how much freedom the draggable has in its movement. Reload and try this code:

```
$('container').select('li').each( function(li) {
  new Draggable(li, { snap: 50 });
});
```

Notice that these draggables move far less fluidly. The `50` value for `snap` tells them to "snap" to points on the page that are separated by 50 pixels in both directions. Before, each movement of the mouse also moved the draggable; now, the draggable moves to the closest x/y coordinates that are both multiples of `50`.

We could also have specified two values for `snap`; `[10, 50]` creates a grid of points 10 pixels apart on the x axis and 50 points apart on the y axis. Better yet, we can set `snap` to a function that determines, on each movement of the mouse, where the draggable should snap to. Advanced usage of draggables will be covered later in the chapter.

One other option deserves mention. The `handle` option lets you specify a smaller piece of the draggable—not the whole draggable—to serve as the *drag-enabled* area of the element. A drag operation intercepts all clicks and cancels the associated events, so handles can be used to enable dragging without interfering with form element interaction, link clicking, and so on.

Let's change part of our HTML to see this in action:

```
<ul id="container">
  <li id="item_1"><span class="handle">@</span> Lorem</li>
  <li id="item_2"><span class="handle">@</span> Ipsum</li>
  <li id="item_3"><span class="handle">@</span> Dolor</li>
  <li id="item_4"><span class="handle">@</span> Sit</li>
  <li id="item_5"><span class="handle">@</span> Amet</li>
</ul>
```

These `spans` will act as our "handles," so let's style them to stand out:

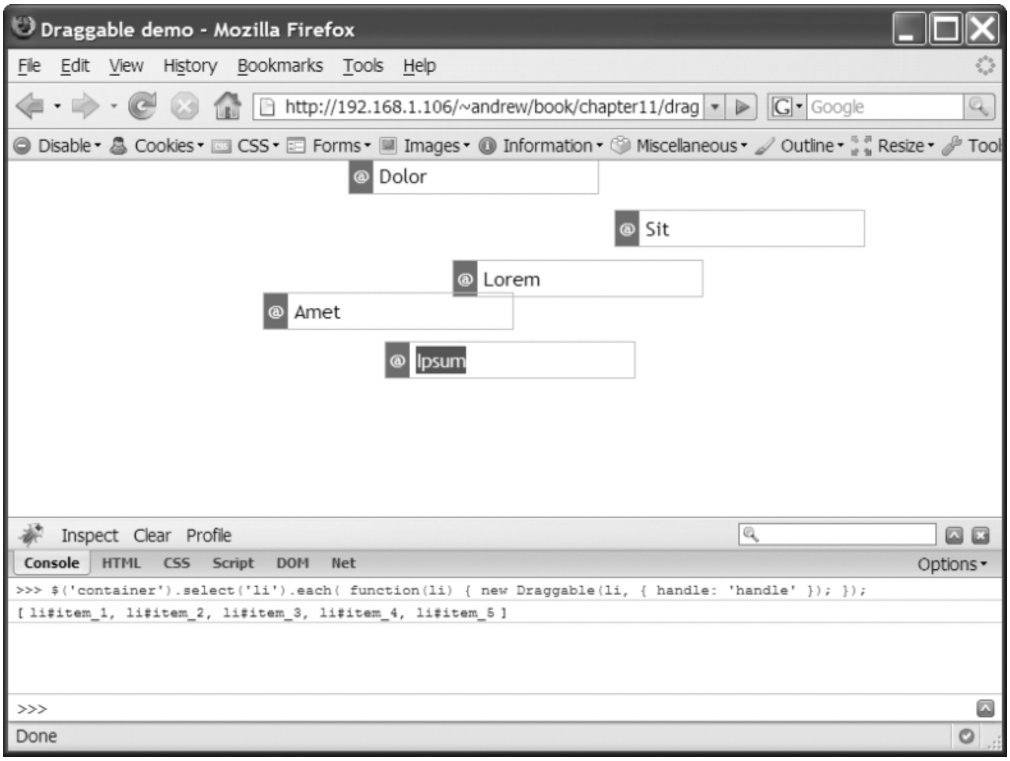
```
#container .handle {
  background-color: #090;
  color: #fff;
  font-weight: bold;
  padding: 3px;
  cursor: move;
}
```

The standard "move" cursor (which looks like arrows in four directions on Windows, and a grabbing hand on the Mac) provides a visual clue to the user that the handles are clickable.

Make these changes, reload the page, and run this in the console:

```
$('container').select('li').each( function(li) {
  new Draggable(li, { handle: 'handle' });
});
```

You'll be able to drag the items around using their handles, but the rest of the element won't respond to clicking and dragging. Notice, for instance, how you can select the text inside each element—an action that was prevented when the draggable hogged all the mouse actions on the element (see Figure 11-2).


Draggable demo - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/book/chapter11/drag
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tool
Dolor
Sit
Lorem
Amet
Ipsum
Inspect Clear Profile
Console HTML CSS Script DOM Net
Options
Done

```
>>> $('container').select('li').each( function(li) { new Draggable(li, { handle: 'handle' }); });
[ li#item_1, li#item_2, li#item_3, li#item_4, li#item_5 ]
>>>
```

Figure 11-2. *Draggables are now moved by their handles.*

Other Draggable Options

I've covered the most important options for `Draggable`, but there are others you might find handy as well.

The constraint Option

The `constraint` option can restrict a draggable to movement along one axis. Setting it to `"horizontal"` limits the draggable to horizontal movement only; setting it to `"vertical"` limits it to vertical movement only. By default, this option is not set. This option is typically used by sortables to enforce linear ordering of items.

The ghosting Option

The `ghosting` option controls how the draggable behaves while being dragged. When it is set to `false`, which is the default, dragging an element moves the element itself. When it is set to `true`, dragging an element *copies* the element and moves the *copy*. At the end of the drag operation, this "ghost" element will be removed (but not before reverting to its original spot, if it needs to).

I can think of two common scenarios in which `ghosting` should be enabled. One is when the drag implies *duplicating* an item, rather than *moving* it (i.e., the original item will remain when the drag is done). The other is when the tentativeness of the drag operation needs to be emphasized. For instance, think of the Finder or Windows Explorer; files are ghosted upon drag because the drop could fail for a number of reasons. Perhaps the destination is read-only, or perhaps there are file name conflicts at the destination. Instead of the file being visually removed from one space, it remains there until the OS can be sure it can move the file to the destination folder.

The zindex Option

script.aculo.us sets a default CSS `z-index` value of `1000` on the dragged element; in most cases, this is a high enough value that the draggable will appear above everything else on the page. If your web app defines `z-index` values higher than `1000`, you should use the `zindex` property to specify a higher value when you construct draggables.

Start, End, and Revert Effects

By default, an element that's being dragged becomes translucent, as you may have noticed. This is the draggable's *start effect*. The opacity change is a common pattern to signify something is being moved—both Windows Explorer and Mac OS X's Finder employ it—but this effect can be customized if need be.

To override the start effect, set the `starteffect` parameter to a function that takes an element as its first argument. That function will be called any time a drag starts.

```
new Droppable('foo', {
  starteffect: function(element) {
    element.morph("background-color: #090");
  }
});
```

This should remind you of the callback pattern you've encountered already—most recently in Chapter 10 in the discussion on effects.

There are two other kinds of effects: the end effect and the revert effect. The *end effect*, of course, is called when the drag ends—when the user releases the mouse and "drops" the draggable. The *revert effect* is called when a draggable reverts to its original location; keep in mind that the `revert` option must be set to `true` for this effect to play.

Exploring Droppables

The counterpart to `Draggable`, `Droppables` is the object that manages all draggable-friendly containers. When an element is made a droppable, it listens for `mouseover` events to determine when an element is being dragged into it. Then it negotiates with the given draggable to determine if they're compatible.

Making Droppables

The interface to `Droppables` is a bit different from that of `Draggable`. It's not a class, so you don't declare instances of it. Instead, you use the `Droppables.add` method:

```
Droppables.add('some_element', options);
```

To illustrate, let's add a new `ul` to our page. It will be empty at first, but will accept any `li` that's dragged into it. (Note that we've removed the handles we added in the previous exercise.)

```
<ul id="container">
  <li id="item_1">Lorem</li>
  <li id="item_2">Ipsum</li>
  <li id="item_3">Dolor</li>
  <li id="item_4">Sit</li>
  <li id="item_5">Amet</li>
</ul>

<ul id="drop_zone"></ul>
```

Let's also style the two containers so that we can clearly see their boundaries. We'll also float them to the left so that they appear side by side rather than stacked.

```
#container, #drop_zone {
  width: 200px;
  height: 300px;
  list-style-type: none;
  margin-left: 0;
  margin-right: 20px;
  float: left;
  padding: 0;
  border: 2px dashed #999;
}
```

All this typing into the Firebug console is getting tedious, so let's add some code to `draggable.html` that will run on `dom:loaded`:

```
<script type="text/javascript">
  document.observe("dom:loaded", function() {
    $('container').select('li').each( function(li) {
      new Draggable(li);
    });
    Droppables.add('drop_zone');
  });
</script>
```

Here, we're declaring our new `ul` as a droppable, but we're not providing an `options` argument. By default, this droppable will "accept" any draggable that is dropped onto it.

Reload the page and drag an item from the left box to the right box. Nothing happens. You can drag the list items wherever you like, but you could do that already.

Don't worry, this is by design. script.aculo.us does not make any assumptions about the purpose of the drag/drop operation, so it will only *do* what you tell it to do.

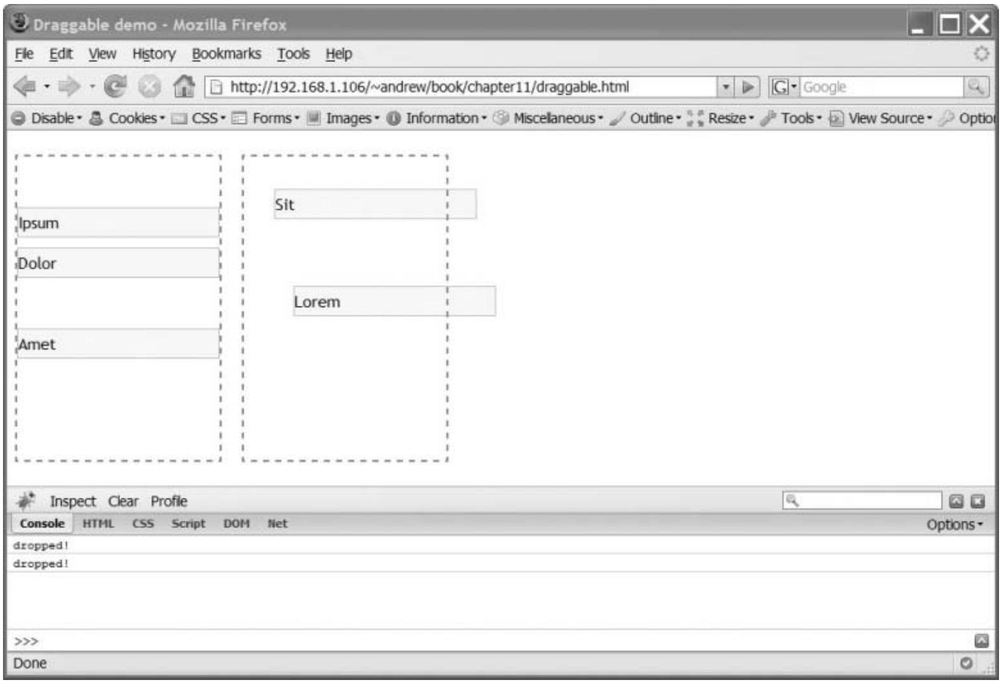
Using Callbacks for Droppables

There are two callbacks for droppables: `onHover` and `onDrop`. When a draggable is dragged over a compatible droppable (i.e., a droppable that would accept it), the `onHover` callback fires. When it is dropped onto a droppable, the `onDrop` callback fires.

To illustrate this in the simplest way possible, let's add an `onDrop` callback to the preceding code:

```
Droppables.add('drop_zone', {
  onDrop: function() { console.log("dropped!"); }
});
```

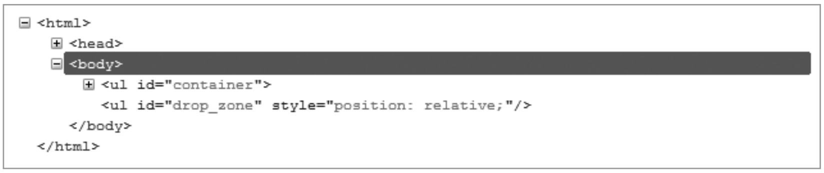
Now reload the page and try again. Make sure the Console tab of Firebug is visible so that you can see when messages are logged. You should see something like Figure 11-3.


Draggable demo - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/book/chapter11/draggable.html
Sit
Ipsum
Dolor
Lorem
Amet
Inspect Clear Profile
Console HTML CSS Script DOM Net
Options
Done

```
dropped!
dropped!
```

Figure 11-3. *Firebug logs our sanity-check event handler.*

Aha! Progress. As a developer, you might expect a drop to correspond with a movement in the DOM. The act of dragging an `li` over a `ul` *feels* like it ought to detach that `li` from its current parent and attach it to the new `ul`. But Firebug's HTML tab, pictured in Figure 11-4, shows that the target `ul` is still empty.



```
<html>
  <head>
  <body>
    <ul id="container">
      <ul id="drop_zone" style="position: relative;"/>
  </body>
</html>
```

Figure 11-4. *Firebug confirms that the drag-and-drop action did not change the draggable's position in the document.*

We can use our `onDrop` callback to append the draggable to the droppable. It takes three arguments: the draggable element, the droppable element, and the `mouseup` event linked to the drop.

Let's start by writing a general function that will get called for drops. We can use appendChild, the built-in DOM method, to add the draggable as a child of the droppable.

```
function appendOnDrop(draggable, droppable, event) {
  droppable.appendChild(draggable);
}
```

We don't have to detach the draggable first; appendChild works for moving elements. (We could have used Element#insert in the same way; it behaves just like appendChild if it's passed a single element.)

Now we should use this function as the onDrop callback for the droppable. With these changes, your script block should look like this:

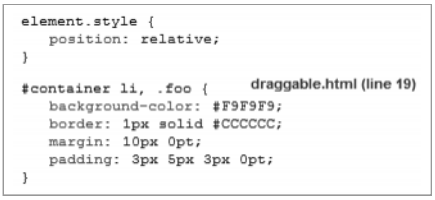
```
<script type="text/javascript">
  document.observe("dom:loaded", function() {
    function appendOnDrop(draggable, droppable, event) {
      droppable.appendChild(draggable);
    }

    $('container').select('li').each( function(li) {
      new Draggable(li);
    });

    Droppables.add('drop_zone', { onDrop: appendOnDrop });
  });
</script>
```

Now reload the page. Before you try the drag again, go to Firebug's HTML tab and look at the source tree. Expand the body tag and you'll see our two uls—one with children, one without. If our code works correctly, though, the first list will lose a child and the second will gain a child. We'll see these changes in the source tree—a node gets highlighted for a moment when its DOM subtree is modified.

Drag one of the items from the left column to the right. Whoa—the element jumps to the right when we let go of the mouse. Why does this happen?

Firebug tells us we got it right—the second ul now has a child. Expanding it in the source tree reveals that the droppable is now a child of the second ul. We need to take a closer look at this.

Reload the page and expand the Firebug source tree again. Drill down to one of the draggable elements and click it in the tree (or you can click the Inspect button, and then click a draggable). The element is now highlighted in blue in the source tree, and its CSS styling is shown in the rightmost pane (see Figure 11-5).

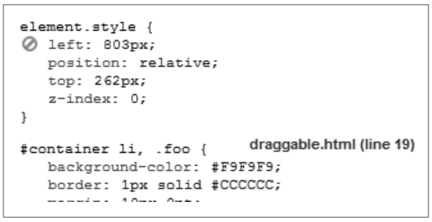


```
element.style {
   position: relative;
}

#container li, .foo {      draggable.html (line 19)
   background-color: #F9F9F9;
   border: 1px solid #CCCCCC;
   margin: 10px 0pt;
   padding: 3px 5px 3px 0pt;
}
```

Figure 11-5. *Firebug lets us inspect an element's entire style cascade.*

This pane shows the style *cascade* of the element—the properties from all CSS rules that apply to the element, with more specific selectors nearer to the top. At the very top, under `element.style`, are the styles that have been assigned to that element directly—whether through inline styles (the element's `style` attribute) or through JavaScript (the DOM node's `style` property).

Firebug shows CSS changes in real time, just like DOM changes. Click and drag on the element you're inspecting, and move it around the page—but don't drop it on the other `ul` just yet. You'll notice that the `element.style` rules update as you move the element. When you're done moving it, the CSS pane should look like Figure 11-6.



```
element.style {
⊘  left: 803px;
   position: relative;
   top: 262px;
   z-index: 0;
}

#container li, .foo {      draggable.html (line 19)
   background-color: #F9F9F9;
   border: 1px solid #CCCCCC;
```

Figure 11-6. *Firebug helps us understand how our draggable determines its position.*

So now we know how the element gets moved around the page—it's given a `position` of `relative`, along with `top` and `left` values that correspond to the position of the mouse. The style rules persist even after the drag ends.

Now drag the element over the empty `ul` and drop it. Once again, the element moves to the right. Do you understand why now?

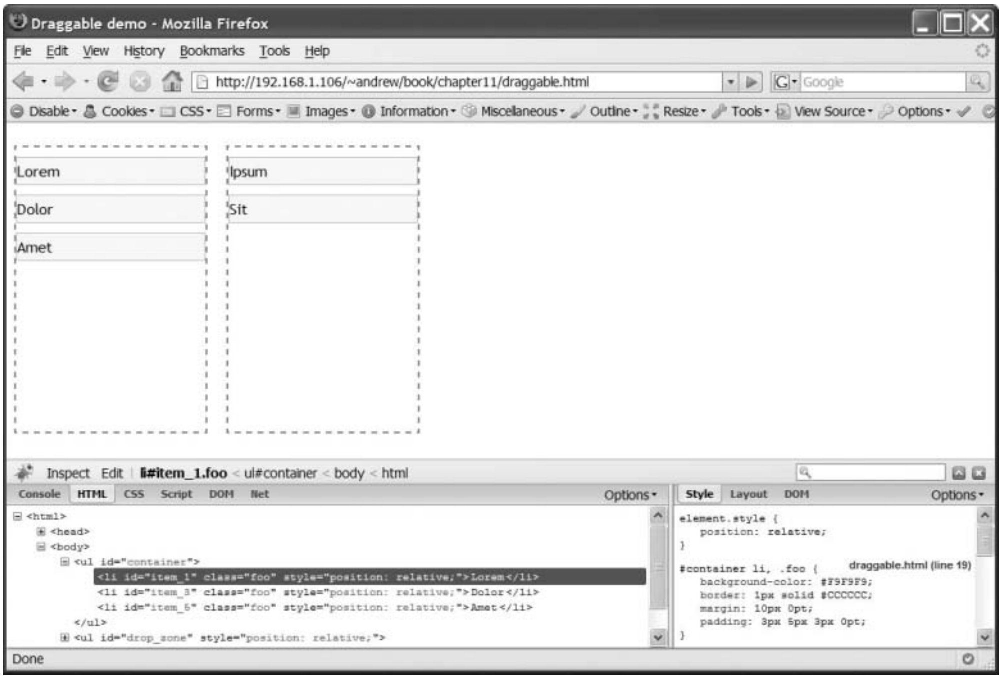
Remember that when an element has a CSS `position` of `relative`, the `top` and `left` properties move the element relative to its original position on the page. In other words, when `top` and `left` are both `0`, the element is in its normal spot—the same place it would occupy if it had a `position` of `static`.

When we moved the element from one parent to another, we changed the way its positioning was calculated. It's now offset from the normal space it would occupy in its *new* parent `ul`.

If we want to position the draggable inside the new `ul`, we can set its `left` and `top` properties to `0` in the `onDrop` callback:

```
function appendOnDrop(draggable, droppable, event) {
  droppable.appendChild(draggable);
  draggable.setStyle({ left: '0', top: '0' });
}
```

Reload `draggable.html`. Now when items are dragged from the first list to the second, they "snap" into place in the new list (see Figure 11-7).


Draggable demo - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/book/chapter11/draggable.html
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tools View Source Options
Lorem
Dolor
Amet
Ipsum
Sit
Inspect Edit li#item_1.foo < ul#container < body < html
Console HTML CSS Script DOM Net Options
Style Layout DOM Options
Done

```
<html>
  <head>
  <body>
    <ul id="container">
      <li id="item_1" class="foo" style="position: relative;">Lorem</li>
      <li id="item_3" class="foo" style="position: relative;">Dolor</li>
      <li id="item_5" class="foo" style="position: relative;">Amet</li>
    </ul>
    <ul id="drop_zone" style="position: relative;">
```

```
element.style {
   position: relative;
}

#container li, .foo {          draggable.html (line 19)
   background-color: #F9F9F9;
   border: 1px solid #CCCCCC;
   margin: 10px 0pt;
   padding: 3px 5px 3px 0pt;
}
```

Figure 11-7. *Now that we adjust the element's positioning when it's dropped, it appears to fall into its receiving droppable.*

Drag-and-Drop: Useful or Tedious?

Does this feel a bit more "manual" than usual? Do you think that some of the code you wrote should've been handled by script.aculo.us automatically?

Keep in mind that `Draggable` and `Droppables` are low-level tools. They don't make assumptions about use cases; in fact, they don't do much without your explicit mandate. They are designed to eliminate the largest headaches related to drag-and-drop. It's no fun to write your own code that determines when one element "overlaps" another. It's no fun to write the same three event handlers for every element that can be dragged. These problems are best solved by code that's as widely applicable as possible.

That said, you can (and should) build custom controls on top of the base logic of `Draggable` and `Droppables`. And that brings us to a specialized control that leverages drag-and-drop—script.aculo.us's own `Sortable` object uses them for drag-reordering of lists.

Exploring Sortables

Sortables are a specialized yet common application of drag-and-drop. Sortables are concerned with the *sequence* of a certain group of items, so rather than moving elements from one container to another, the user moves the elements relative to one another.

Use cases for sortables are everywhere to be found. Think of, for example, your favorite rental-by-mail service, for which you maintain a queue of DVDs and/or video games you'd like to play. It's crucial for you to be able to arrange that list in order of most-desired to least-desired.

Making Sortables

Making a sortable is much like making a droppable. There is one object, `Sortable`, that manages all sortables on a page. Creating a new one requires a call to `Sortable.create`:

```
Sortable.create('container');
```

As with `Droppables.add`, the first argument refers to the element that will act as a container. In fact, this simple example already works with our example markup—swap out the code you wrote earlier with this code:

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

    <style type="text/css" media="screen">
      body {
        font-family: 'Trebuchet MS', sans-serif;
      }
      #container {
        width: 200px;
        list-style-type: none;
        margin-left: 0;
        padding-left: 0;
      }

        #container li, .foo {
          background-color: #f9f9f9;
          border: 1px solid #ccc;
          padding: 3px 5px;
          padding-left: 0;
          margin: 10px 0;
        }

        #container li .handle {
          background-color: #090;
          color: #fff;
          font-weight: bold;
          padding: 3px;
        }

      #container, #drop_zone {
        width: 200px;
        height: 300px;
        list-style-type: none;
        margin-left: 0;
        margin-right: 20px;
        float: left;
        padding: 0;
        border: 2px dashed #999;
      }

    </style>

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

```
    <script type="text/javascript">
    document.observe("dom:loaded", function() {
      Sortable.create('container');
    });
    </script>
  </head>
  <body>

  <ul id="container">
    <li class="foo" id="item_1">Lorem</li>
    <li class="foo" id="item_2">Ipsum</li>
    <li class="foo" id="item_3">Dolor</li>
    <li class="foo" id="item_4">Sit</li>
    <li class="foo" id="item_5">Amet</li>
  </ul>

  </body>
</html>
```

Aside from the script element, the only change we've made is to remove the "drop zone" container, which we don't need anymore.

Reload this page, and you'll find that the container is already behaving like a sortable. The list items inside it can be dragged and reordered; when a dragged element gets near a new "slot," the elements on either side part to give it room. When dropped, the element stays in its new place.

Sortable Options

We were able to use `Sortable.create` so effortlessly because our markup conformed to many of its default options. The more specialized the usage, however, the more configuration will be needed. The following subsections describe some of the other options you might use with sortables.

The tag Option

`Sortable.create` looks for children of a certain tag name to declare as draggables. This option, `tag`, is a string that refers to that tag name. It defaults to `li`, the semantic child of unordered and ordered lists (`ul` and `ol`). If your draggables are not going to be list items, you must specify the tag name.

The only Option

The `only` option is a string that, if set, represents a CSS class that all children must have in order to be treated like draggables. This is a way to further winnow down the set of draggables—for instance, if only certain children are eligible for reordering.

The overlap Option

The `overlap` option expects a string—either `vertical` or `horizontal`. Vertical sortables are the default; horizontal sortables can be created with a smack of CSS trickery and a grasp of element floating rules.

The containment Option

The `containment` option is tricky—it expects an array of element nodes (or strings that refer to element IDs). If an array is given, sortables will be allowed to be dragged *outside* of their parent, and can be dropped into any of the given elements. Or, to be briefer, this option allows for drag-and-drop between two sortables.

The scroll Option

Picture your file manager of choice—Windows Explorer or Mac OS X's Finder. The visual model for interacting with files is easy to learn: you can select a file, drag it into a folder, and thus *move* the file into that directory as though you were filing real papers into a folder.

The model isn't leak-proof, however. An Explorer or Finder window can only show so much at once; what happens if the file and the folder are far apart? Both of these file managers solve the problem by trying to "sense" when the user wants to drag to some-place out of view. When the user moves the mouse near the edge of the viewport, it starts inching in that direction, little by little (or sometimes lightning-fast, if your motor skills aren't too sharp). When the target folder is in view, the user moves toward it and away from the edge of the viewport, and the scrolling stops.

Sortables pose a similar problem, and script.aculo.us contains a similar solution. The `scroll` parameter, if set, will scroll the given container when an item is dragged toward one of its edges.

We can test this behavior on the window itself by resizing it to a very small height. We must pass `window` as the value of the `scroll` parameter:

```
<script type="text/javascript">
  document.observe("dom:loaded", function() {
    Sortable.create('container', { scroll: window });
  });
</script>
```

Reload the page and test the behavior yourself. Notice in Figure 11-8 how the page scrolls smoothly when you drag an item to the bottom of the viewport.

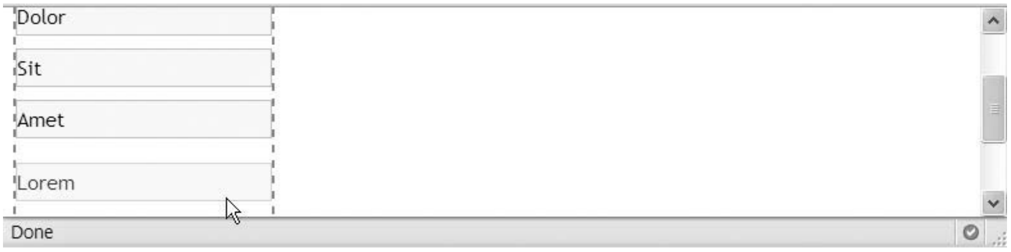


Figure 11-8. *The window scrolls when an item is dragged near the top or bottom of the viewport.*

Because any container can have scrollbars (as determined by its CSS `overflow` property), any container can be used as the value of scroll. For instance, we can tweak our CSS to make our `ul` container much smaller:

```
#container {
  width: 200px;
  height: 100px;
  overflow: auto;
  list-style-type: none;
  margin-left: 0;
  margin-right: 20px;
  float: left;
  padding: 0;
  border: 2px dashed #999;
}
```

As you can see in Figure 11-9, the sortable container itself scrolls when necessary. Another UI innovation freed from the monopolistic clutches of the desktop!

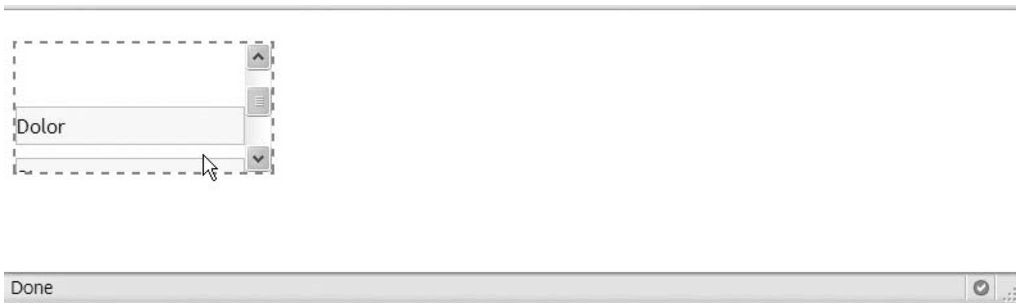


Figure 11-9. *The scroll parameter works on all elements. Here, the container scrolls when an item is dragged near its top or bottom edge.*

Summary

In this chapter, you learned that script.aculo.us contains both high-level and low-level tools. `Draggable` and `Droppables` are low-level tools—middleware for the sorts of places where you'd use drag-and-drop in your applications. Sortable is a high-level tool—a specific usage of drag-and-drop that is ready out of the box.

The next chapter will deal with more high-level tools: UI goodies that solve very specific problems.

CHAPTER 12

Advanced Controls: Autocompleters, In-Place Editors, and Sliders

In the last chapter, we discussed two types of controls provided by script.aculo.us: low-level controls (like drag-and-drop and effects) and high-level controls (like sortables). The former are building blocks for the latter.

In this chapter, we'll look at more of the high-level controls. They're UI widgets tailored for specific use cases, so we'll be discussing *where* to use them as well as *how* to use them.

Adding Autocomplete Functionality

`Autocompleter` is a control similar to the one built into browsers: when the user begins to type in a text box, a menu appears below the text offering completion suggestions.

All major web browsers use this type of UI control for their address bars—typing the beginning of a URL will display a list of URLs in your history that begin with what you've typed. Most also use it on any input field you've typed text into before, although that depends on whether you've configured your browser to remember those values.

The script.aculo.us autocompleter replicates this control, but gives the *developer* control of the suggestion list. It does so by augmenting an ordinary text box (an `input` with a `type` of `text`) with an element for displaying the results (typically a `div` with a `ul` inside) and a listener on the text box that observes what's being typed.

When to Use Autocompleter

You can ask yourself one question to figure out whether an autocompleter is the right solution in a certain place: would this degrade to a *text box* or a *drop-down menu*? In other words, if you weren't able to use an autocompleter and had to choose between the ordinary controls provided by HTML, which would you choose?

The difference between them, of course, is that a text box allows for free-form input, while a drop-down menu restricts input to whatever choices have been set in the HTML.

`Autocompleter` degrades to an ordinary text box. When JavaScript is turned off, it behaves *exactly* like a text box.

Keep your eye on the usability ball—UI controls are meant to solve problems first and foremost. If you can't decide whether to use an autocompleter or a `select` element, choose the latter. Fight the urge to treat the autocompleter as if it were a flashier version of the standard drop-down menu.

Use Case: Suggesting Players

A fantasy football league has legions upon legions of players. Many belong to a specific team in the league; many more are free agents, able to be picked up at any time. They're the leftovers, the players nobody picked in the fantasy draft.

Every season, several no-name players break out, amassing yards and touchdowns as they introduce themselves to a national audience. A clever fantasy owner can spot these diamonds in the rough before anyone else if he pays attention to the numbers.

Create a new folder called `chapter12` and add an `index.html` file from the standard template. To start off, we'll need only two elements inside the `body` of the page: the text box and the container to hold suggestions.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
    <title>Chapter 12</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
    <script src="../js/prototype.js" type="text/javascript"""></script>
    <script src="../js/scriptaculous.js" type="text/javascript"""></script>
  </head>
  <body>
    <input type="text" name="player_name" id="player_name" size="30" />
    <div id="player_suggestions"></div>
  </body>
</html>
```

Using Autocompleter

`Autocompleter` comes in two flavors. One, named `Autocompleter.Local`, keeps an array of possible suggestions for instantaneous feedback. The other, `Ajax.Autocompleter`, relies on a remote server to give it suggestions—making an Ajax call whenever suggestions need to be retrieved.

Naturally, `Autocompleter.Local` has less overhead, since it does not need to talk to the server; as a result, its suggestions will usually appear far more quickly. You'll only want to fall back to `Ajax.Autocompleter` if the pool of suggestions is simply too large to keep on the client side, or if the logic for picking suggestions is more complicated than an ordinary string match. We'll cover these cases several pages from now, but for now let's use the local version for simplicity's sake.

Using Autocompleter.Local

The syntax for declaring a new `Autocompleter.Local` is

```
 new Autocompleter.Local(inputElement, updateElement, array, options);
```

where `inputElement` is the text box, `updateElement` is the element that will hold the suggestions, `array` is the list of possible suggestions, and `options` is the configuration object we've come to know so well.

Let's add a script tag to the page and declare an `Autocompleter.Local` on page load:

```
<script type="text/javascript">
  document.observe('dom:loaded', function() {
    new Autocompleter.Local('player_name', 'player_suggestions',
     ['James Polk', 'James Buchanan', 'Franklin Pierce',
      'Millard Fillmore', 'Warren Harding', 'Chester Arthur',
      'Rutherford Hayes', 'Martin Van Buren']);
  });
</script>
```

Our array contains a short list of nonnotable American presidents—the benchwarmers of our fictional league. Rather than make the user choose from an endless drop-down menu or force her to remember how many *l*s are in *Fillmore*, `Autocompleter` will offer possible name completions after only a few keystrokes.

Save `index.html` and open it in a browser. You'll see a text box, of course, as shown in Figure 12-1.

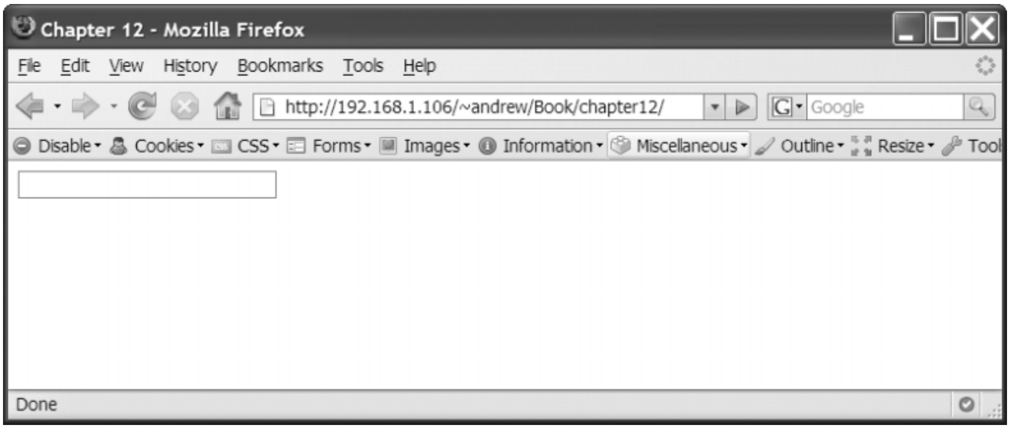
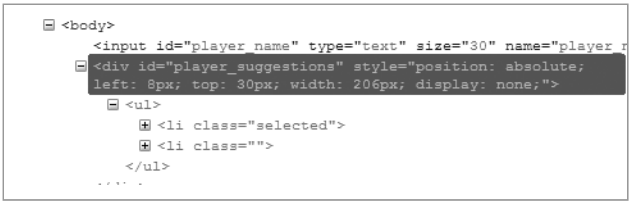

Chapter 12 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/Book/chapter12/
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tool
Done

Figure 12-1. *The autocompleter in its ordinary state*

There's more going on here than would appear. Bring up the Firebug pane and move to the HTML tab (see Figure 12-2).

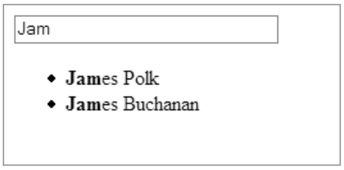


```
<body>
    <input id="player_name" type="text" size="30" name="player_r
  <div id="player_suggestions" style="position: absolute;
  left: 8px; top: 30px; width: 206px; display: none;">
    <ul>
        <li class="selected">
        <li class="">
      </ul>
```

Figure 12-2. *Firebug's view of the panel that will hold our suggestions*

Two things have already happened. Our container `div` is hidden; it won't appear until it has at least one suggestion to present. But expanding the `div` shows that there's now an empty `ul` inside.

Let's put that `div` to work. Click the text box and start typing the name "James." You won't even need to finish the name—two suggestions will appear before you're done (see Figure 12-3).



Jam

- **Jam**es Polk
- **Jam**es Buchanan

Figure 12-3. *Two possible completions appear.*

You should notice two things. First, the list of possible suggestions is populated dynamically by `Autocompleter`. When there is at least one match from the suggestion bank, the container `div` is shown and its child `ul` is filled with one list item for each match. When there are *no* matches, the container `div` is hidden, and its child `ul` is stripped of all its items.

Second, the list looks *awful*. It doesn't even look like a menu. There's no *affordance*, nothing that says, "I can be clicked." But that's by design. `Autocompleter` constructs the HTML for your list, but leaves the styling up to you. Look at the HTML tree in Firebug to see what I mean:

```
<div id="player_suggestions">
  <ul>
    <li class="selected">
      <strong>Jame</strong>s Polk
    </li>
    <li>
      <strong>Jame</strong>s Buchanan
    </li>
  </ul>
</div>
```

So we know that the individual choices are list items, and that the "active" choice will have a class name of `selected`. A portion of each choice is wrapped in a `strong` element, highlighting the substring that matches what you've typed. As you type more characters, the strongly emphasized part of the phrase will grow.

We need to style this HTML to look more like the drop-down menus that users are familiar with. It needn't look exactly like a *native* drop-down menu, of course, but it should resemble one enough that a user can recognize its purpose.

Add a `style` tag in the head of `index.html`:

```
<style type="text/css">
body {
  font: 67.5% "Lucida Grande", Verdana, sans-serif;
}

/* a thin border around the containing DIV */
#player_suggestions {
  border: 1px solid #999;
  background-color: #fff;
}
```

```
  /* get rid of the bullets and un-indent the list */
  #player_suggestions ul {
    list-style: none;
    margin: 0;
    padding: 0;
  }

    #player_suggestions li {
      padding: 2px 3px;
    }

    #player_suggestions strong {
      font-weight: bold;
      text-decoration: underline;
    }

    /* the "active" item will have white text
       on a blue background */
    #player_suggestions li.selected {
      color: #fff;
      background-color: #039;
    }
</style>
```

The devil is in the details when you're trying to make styled HTML look like built-in controls and widgets (see Figure 12-4). If we wanted to do something more elaborate, we could replace the selected item's background color with a background image—one that has a slight gradient—to give the item some texture (as an homage to Mac OS X). Or we could decrease the opacity of the menu itself, matching the transparency of Windows Vista's pull-down and drop-down menus. It's up to you. HTML is your canvas and you're Bob Ross.

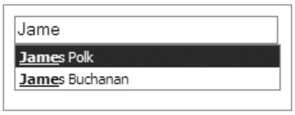


Figure 12-4. *CSS makes the list of suggestions look more like a native control.*

Time to see what this thing's made of. Bring focus to the text box and start typing "James" once again. Experiment with the several ways you can choose from the menu:

- Click a suggestion.
- Use the up/down arrow keys to highlight a suggestion, and then press Enter.
- Use the up/down arrow keys to highlight a suggestion, and then press Tab.

Now try several ways to dismiss the menu:

- Click outside of the text box or move focus somewhere else with the keyboard.
- Press the Escape key.
- Finish typing the desired choice. For example, if you type "James Buchanan," the menu will disappear as you type the final *n*, since there's no more completing to be done.

When a user recognizes our list as a drop-down menu, he'll expect it to *act* like one. For the user's sake, `Autocompleter` tries very hard to ape the behavior of a drop-down.

Robust Autocompleter, the Ajax Version

The more robust version of `Autocompleter` is `Ajax.Autocompleter`. Rather than store the suggestion bank locally, it gets a list of suggestions from the server. This approach has two large advantages:

- `Autocompleter.Local` does a simple text match against the string. In an autocompleter that suggests US cities, when I type "New," I'll get offered "New Orleans" and "New York" (among others) as suggestions.

  But when I type MSY, the airport code for New Orleans (as can be done on many travel sites), I receive no suggestions. The logic that matches an airport code to a city is beyond the scope of `Autocompleter.Local`.

  `Ajax.Autocompleter` punts on that logic. It tells the server what the user typed and accepts whatever is returned. On the server side, I can do a more complicated match—I can search for string matches against city names, airport codes, or any other identifiers I choose.

- `Ajax.Autocompleter` makes perfect sense when dealing with humongous data sets. In our case, we're pulling from a small list, but a professional football league has around 30 teams and 53 active players per team—roughly 1,600 players. It's *possible* to use `Autocompleter.Local` with such a huge data set, but do you really want to place a 1,600-item array in your page and transfer it over the pipe?

To demonstrate `Ajax.Autocompleter`, we'll first need to write a script that can return suggestions when it's called via Ajax.

■ **Note** For this part, you'll need to be running a web server (either locally or remotely) that supports PHP. Naturally, the concept is similar for other server environments.

Create a file called `autocomplete_players.php` in the same folder as your `index.html` file:

```
<?php
  // For this sample script we'll use another array; in a
  // real-world app, we'd probably search a database instead.
  $suggestions = array(
    'James Polk', 'James Buchanan', 'Franklin Pierce',
    'Millard Fillmore', 'Warren Harding', 'Chester Arthur',
    'Rutherford Hayes', 'Martin Van Buren'
  );

  $value = isset($_REQUEST['player_name']) ? $_REQUEST['player_name'] : "";
  $matches = array();
  foreach ($suggestions as $suggestion) {
    // Look for a match (case-insensitive).
    // If found, wrap the matching part in a STRONG element,
    // wrap the whole thing in a LI,
    // and add it to the array of matches.
    if (FALSE !== stripos($suggestion, $value)) {
      $match = preg_replace('/' . preg_quote($value) . '/i',
       "<strong>$0</strong>", $suggestion, 1);
      $matches[] = "<li>${match}</li>\n";
    }
  }

  // Join the matches into one string, then surround it
  // with a UL.
  echo "<ul>\n" . join("", $matches) . "</ul>\n";
?>
```

The output of this PHP script is the literal HTML to be inserted into the page.

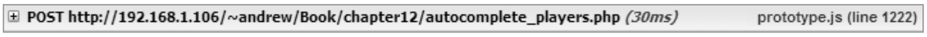
Now we'll add the JavaScript that sets up the autocompleter on page load. The syntax for `Ajax.Autocompleter` is identical to that of `Autocompleter.Local`, except for the third argument:

```
new Ajax.Autocompleter(inputElement, updateElement, url, options);
```

Instead of `array`, we provide the URL for the Ajax request. We can give a simple relative URL, since `index.html` and `autocomplete_players.php` are in the same directory:

```
<script type="text/javascript">
  document.observe('dom:loaded', function() {
    new Ajax.Autocompleter('player_name', 'player_suggestions',
     'autocomplete_players.php');
  });
</script>
```

Reload the page. Make sure the Firebug console is visible. Click the text box, but this time *type very slowly*. First, type a **J**, and then look at the console (see Figure 12-5).



```
⊞ POST http://192.168.1.106/~andrew/Book/chapter12/autocomplete_players.php (30ms)        prototype.js (line 1222)
```

Figure 12-5. *Line indicating an Ajax request went out*

A line appears in the console to tell us that an Ajax request went out. Look at the details of the request, specifically the Post and Response tabs.

Now move back to the text field and add an *A*, and then an *M*. There will be two more logged Ajax calls in the console. You've probably figured it out: `Ajax.Autocompleter` is sending out an Ajax request each time the text box's value changes.

Seems wasteful, doesn't it? If the Internet *were* a series of tubes, this is the sort of thing that would clog them up.

In fact, `Ajax.Autocompleter` does some clever throttling: it waits for a pause in the input before it sends out a request. To see for yourself, clear the text field, and then type "James" at your normal typing speed. Unless you're a hunt-and-peck typist, `Ajax.Autocompleter` won't make a request until you've pressed all five keys.

Common Options and Features

Let's not forget about the *fourth* argument—the one that lets us go under the hood. The two versions of `Autocompleter` share some configuration options:

- `tokens` lets you "reset" the suggestions every time a given key is pressed. Think of an e-mail client: when you compose a new message, most clients display suggestions as you type in the To and CC fields of the message window. Addresses are delimited by semicolons; pressing the semicolon key tells the client that you've entered one address and are ready to enter another.

  The `tokens` option accepts either a string or an array of strings. It lets you specify delimiters for your suggestions. The following code will split on commas and semicolons:

  ```
  new Ajax.Autocompleter('player_name', 'player_suggestions',
   'autocomplete_players.php', { tokens: [',', ';' ] });
  ```

- `frequency` controls how long of a pause, in seconds, is needed to trigger suggestions. (This is the source of the "throttling" described previously.) Naturally, this behavior is more useful with the Ajax flavor—if the remote server is sluggish to respond, you may want to make this value larger than the default of `0.4`.

- `minChars` controls how many characters need to be typed before the autocompleter presents suggestions. It's set to `1` by default, but if the suggestion bank is especially large, the list of suggestions after one character will be long, unwieldy, and unhelpful. Also, raising this value is another way to reduce the number of Ajax requests made by `Ajax.Autocompleter`.

  This option is token-aware; if you've specified any tokens, the autocompleter will wait the proper number of keystrokes after *each* token before it starts offering suggestions.

Several callback options let you hook into `Autocompleter` at important points:

- `onShow` and `onHide` control how the completion menu reveals and hides itself. If specified, they will *replace* the default hide/show behaviors. (By default, the menu uses `Effect.Appear` and `Effect.Fade`.) If you override `onShow`, be prepared to handle the sizing and positioning of the menu on your own.

  These callback functions take two parameters: the text box and the menu container (i.e., the first two arguments passed to the constructor).

- `updateElement` and `afterUpdateElement` are used to replace or augment what takes place when the user selects a suggestion from the menu.

  `updateElement` takes one argument—the `li` that was chosen—and *replaces* the default logic (i.e., set the value of the text box to the text content of the `li`). `afterUpdateElement` takes two arguments—the `input` element and the `li` element—and fires *after* the `updateElement` callback.

Adding In-Place Editing Functionality

`Ajax.InPlaceEditor` is a script.aculo.us class for a UI pattern that is becoming more and more common on web sites. Picture the following:

1. You're on the page for your fantasy football team. It displays your team's name and a roster of your players.

2. You move your mouse over the team name and notice that the background color changes slightly. You click. The ordinary text transforms into a compact form—a text box prepopulated with your existing team name and a save button alongside it. Farther to the right is a cancel link that restores the original view.

3. You place focus in the text box and make a change to your team name. You click the save button. After a short pause, the original view is restored—except that the new team name is now shown (see Figure 12-6).

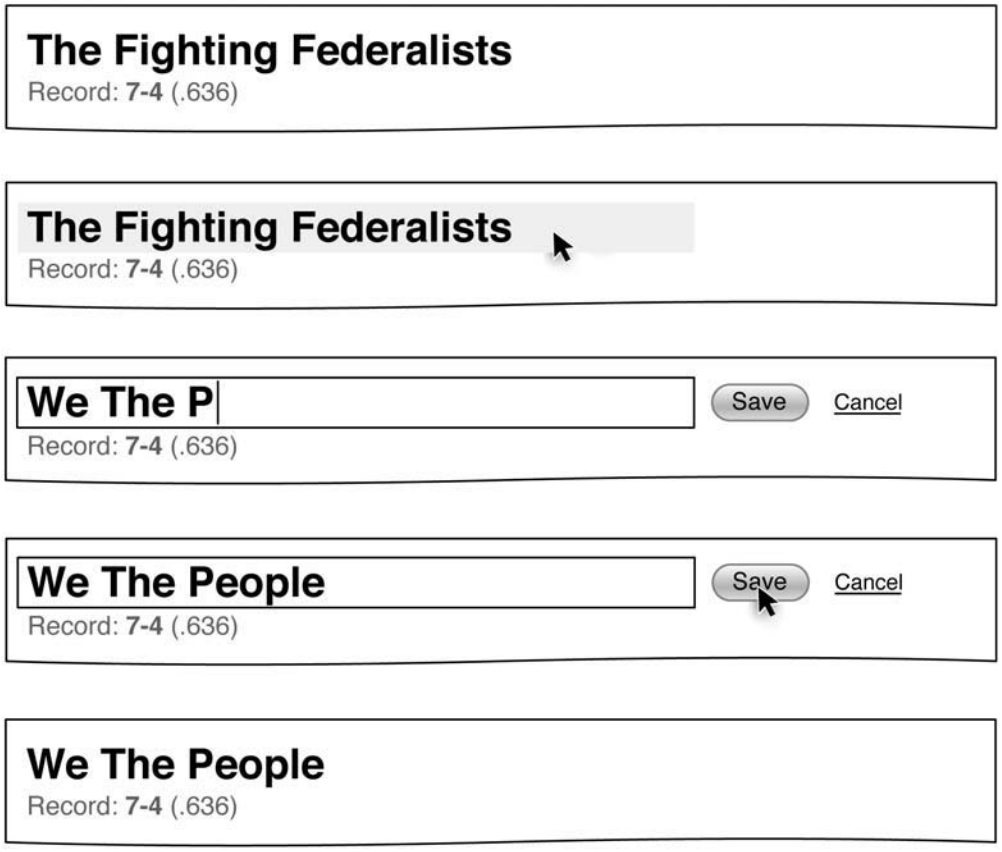


Figure 12-6. *The user workflow for an Ajax in-place editor*

For obvious reasons, this is called an *in-place editor*. This pattern, when executed well, can obviate the *administration section*—the back end of a content management system where all the editing is done. Instead, the read mode and edit mode are merged.

The scenario just described illustrates how this pattern allows a fantasy owner to change her team name. Naturally, permissions would be important—when she's on her own team's page, her team's name could be edited, but she'd be prevented from editing the names of her opponents' teams.

Let's think through how to turn this user workflow into code. We'll need to represent the content in two different ways: read mode and edit mode. *Read mode* will be the element's ordinary markup; *edit mode* will be an HTML form with a text box and an OK button. We'll also insert a cancel link in case the user changes his mind.

To pull this off, we'll need help from the DOM (to switch dynamically from a read view to an edit view) and Ajax (to tell the server when a value has been changed).

As you may have guessed, the script.aculo.us `Ajax.InPlaceEditor` handles all these details. The wonders never cease.

Using Ajax.InPlaceEditor

The syntax for declaring a new `Ajax.InPlaceEditor` is the following:

```
new Ajax.InPlaceEditor(element, url, options);
```

As usual, `element` refers to the element we want to make editable, and `options` refers to our object of configuration parameters. The second argument, `url`, specifies the URL that should be contacted in order to save the data.

Let's create a new page called `inplaceeditor.html`. It won't need much markup—just an element with text that we can edit.

■ **Note** As with the previous example, for this part you'll need to be running a web server (either locally or remotely) that supports PHP. Naturally, the concept is similar for other server environments.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
    <title>Chapter 12</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
    <script src="prototype.js" type="text/javascript"""></script>
    <script src="scriptaculous.js" type="text/javascript"""></script>
    <style type="text/css" media="screen">
      body {
        font: 67.5% "Lucida Grande", Tahoma, sans-serif;
      }
    </style>
  </head>

  <body>

    <h1 id="team_name">The Fighting Federalists</h1>

  </body>
</html>
```

The h1 that contains the team name is annotated with an ID so that we can pass it easily into the Ajax.InPlaceEditor constructor. So let's add a code block to initialize the in-place editor when the DOM is ready. Add this to the head of your document:

```
<script type="text/javascript">
  document.observe('dom:loaded', function() {
    new Ajax.InPlaceEditor('team_name', 'save.php');
  });
</script>
```

We don't need to add any configuration options yet. We've set the url argument to a URL that doesn't exist yet, but we'll take care of that later. This one line is enough to hook up all the client-side behaviors for the in-place editor.

Open the page in a browser, and check that all of these behaviors work:

- When you move your mouse over the h1 element, you should see its background color change to a subtle yellow, inviting you to click it.
- When you click, the element should be replaced with a text box, a button that says "ok," and a link that says "cancel."
- The text in the text box should be highlighted already, so that you can type over it immediately.
- Clicking the "cancel" link should restore the h1 to its initial state.
- Clicking the h1 should bring up the form once again.

We've tested everything except submitting the form—for that, we'll need to write a script that will receive the save request.

Create a new file called `save.php` in the same directory as `inplaceeditor.html`. The in-place editor will, when saved, make an HTTP POST request to the given URL with a `value` parameter; that's the name of our text box. Its value will be whatever the user typed into the text box.

In response, the script should send as output whatever the *new* value of the element should be. This value will nearly always be the same one that the script received. (A production script would also want to store the new value, but we needn't bother.)

So we need only write a PHP script that echoes the value we give it. Here's what your `save.php` file should look like:

```
<?php echo $_REQUEST['value']; ?>
```

Yeah, that's the whole file. Save it.

Now we'll reload `inplaceeditor.html` and try an actual save. Click the element, rename your team, and click the "ok" button.

That was fast, wasn't it? An instant after you clicked the button, the in-place editor returned to its read state, but with the *new value*. An `Effect.Highlight` call makes it clear to the user that the value is "fresh." If you're trying out these examples on a remote server, it probably took a bit longer; but those of us who are running a local web server will need to introduce some fake latency to better simulate the lag of an HTTP round trip. We can tell our `save.php` script to wait for a second before responding:

```
<?php
  sleep(1);
  echo $_REQUEST['value'];
?>
```

After adding this line to `save.php`, try modifying the value in our in-place editor once more (see Figure 12-7). Now you can see much more clearly how it works. As soon as you click the "ok" button, the edit mode disappears and is replaced with a "Saving . . ." message while the Ajax request is made. When the browser receives the response, it switches back to the in-place editor's read mode.

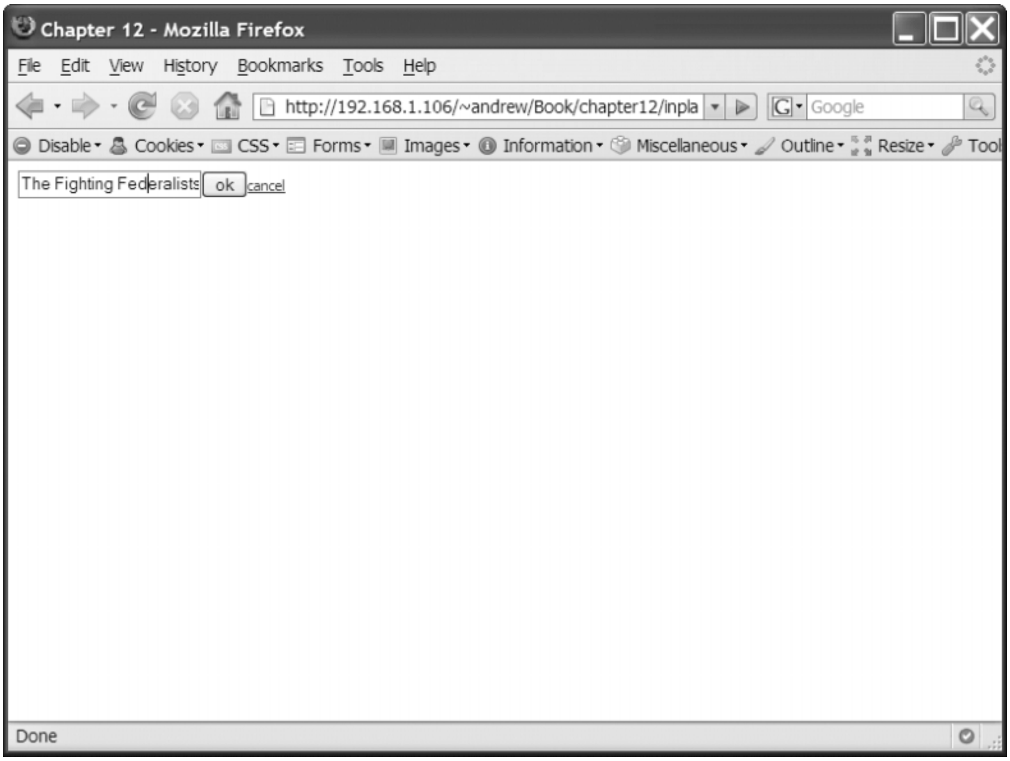

Chapter 12 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/Book/chapter12/inpla
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tool
The Fighting Federalists
ok
cancel
Done

Figure 12-7. *The in-place editor's edit mode*

Making It Pretty

Astute readers will have already noticed that an `h1` element looks much different from a text box with the same text content. We can fix that quite easily with CSS.

Click the in-place editor to put it into edit mode once again—then use Firebug to inspect the text box that appears. (With the Firebug pane visible, click Inspect, and then click in the text box.) The HTML inspector shows us that the `input` element has a class name of `editor_field`, and that its `form` parent node has an ID of `team_name-inplaceeditor` (in other words, it adds `-inplaceeditor` to the ID of the original element.

Armed with this information, we can write a CSS selector that targets both read mode and edit mode. Add this to the `style` element in your document's head:

```
h1#team_name,
form#team_name-inplaceeditor .editor_field {
  font-size: 19px;
  font-weight: bold;
}
```

Now the element looks the same in either mode, as shown in Figure 12-8. Maintaining the text's styling helps the user navigate a new and perhaps unfamiliar UI pattern.

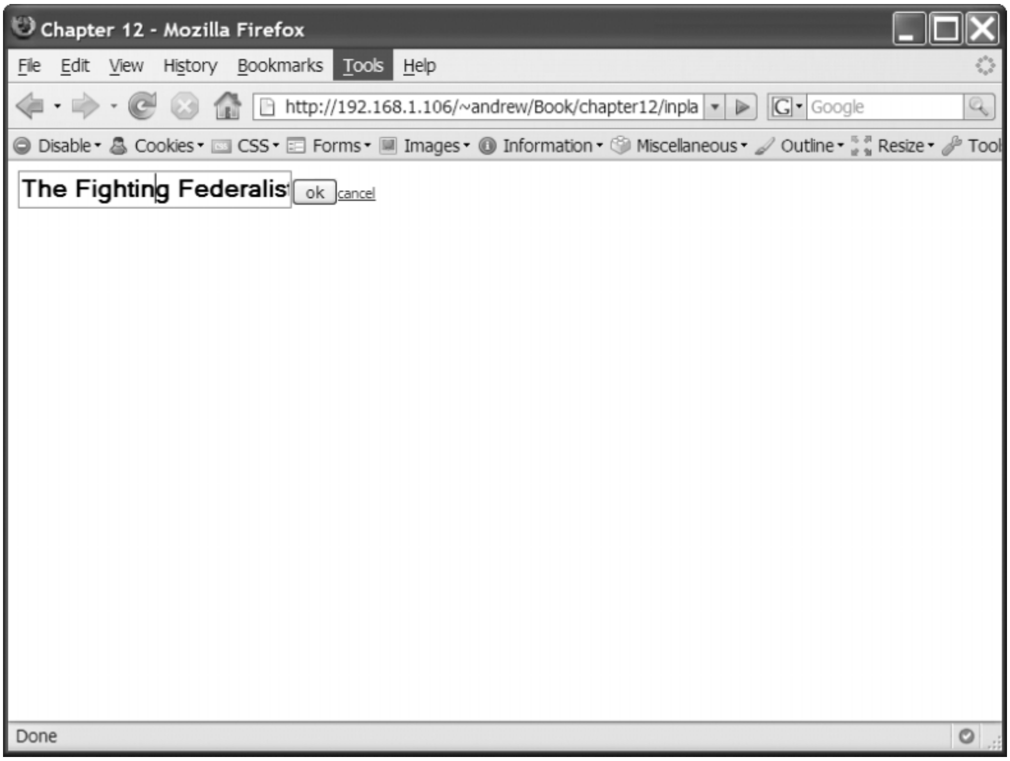

Chapter 12 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.106/~andrew/Book/chapter12/inpla
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tool
The Fighting Federalis
ok
cancel
Done

Figure 12-8. *The read mode (on the left) and the edit mode (on the right) are now far less disjointed-looking, resulting in a friendlier user interface.*

Common Options and Features

`Ajax.InPlaceEditor` is, like its script.aculo.us brethren, endlessly configurable. Here are some of the most useful options:

- `okButton` and `okText` control the display of the button that submits the value to the server. Setting `okButton` to `false` hides the button altogether—the form can only be submitted via the Enter key. The `okText` parameter (which defaults to `"ok"`) lets you change the label of the button (e.g., `"Save"`, `"Go"`, or `"Make it Happen!"`).

- `cancelLink` and `cancelText` control the display of the link that cancels the edit operation. Setting `cancelLink` to `false` hides the link. The `cancelText` parameter lets you change the label of the link from the default `"cancel"` to something far sillier, like `"oops!"` or `"I've Made a Huge Mistake."`.

- `savingText` changes the message that is displayed after you submit the form, but before the server responds. As shown, it defaults to `"Saving..."`.
- `rows` defaults to `1`. If a greater value is given, the in-place editor's edit mode will show a multiline `textarea` (of the given number of rows) rather than the single-line `input`.
- `cols` is undefined by default; if specified, it controls how large the text entry field is. (For `textarea`s, this is dictated by the `cols` attribute; for `input`s, it's dictated by the `size` attribute.)

Adding Sliders

The script.aculo.us slider control is one implementation of a control that, for whatever reason, has no native implementation in web browsers. The ability to drag a "handle" to a specific spot on a horizontal or vertical "track" has a quasi-analog feel that makes it ideal for zooming in and out, setting the size of an item, distributing resources among several tracks, and countless other scenarios.

Creating a Slider

The syntax for creating a slider is the following:

```
new Slider(handle, track, options);
```

Here, `handle` and `track` refer to the DOM elements that will serve as the "handle" and "track" of the slider, respectively. Configuration can be done with the `options` argument. But for our first example, we won't need to do any configuration.

Create a file called `slider.html` and make it look like this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
    <title>Chapter 12</title>

    <script src="js/prototype.js" type="text/javascript></script>
    <script src="js/scriptaculous.js" type="text/javascript"></script>
    <style type="text/css" media="screen">
      body {
        font: 67.5% "Lucida Grande", Tahoma, sans-serif;
      }

      #track {
        width: 300px;
        height: 25px;
        border: 2px solid #555;
      }

      #slider {
        width: 50px;
        height: 25px;
        background-color: #ccc;
        font-size: 20px;
        text-align: center;
      }
    </style>

    <script type="text/javascript">
      document.observe('dom:loaded', function() {
        new Control.Slider('handle', 'track');
      });
    </script>
  </head>
  <body>

    <div id="track">
      <div id="handle"></div>
    </div>

  </body>
</html>
```

The highlighted parts of this HTML file show that we're creating two elements with IDs of `track` and `handle` and placing the latter inside the former. We're also styling these elements so that they look roughly like a scrollbar thumb and a scrollbar track.

The highlighted line of JavaScript is enough to attach the event listeners to make the handle draggable within the confines of the track. Open up `slider.html` in a web browser and you should see the slider shown in Figure 12-9.

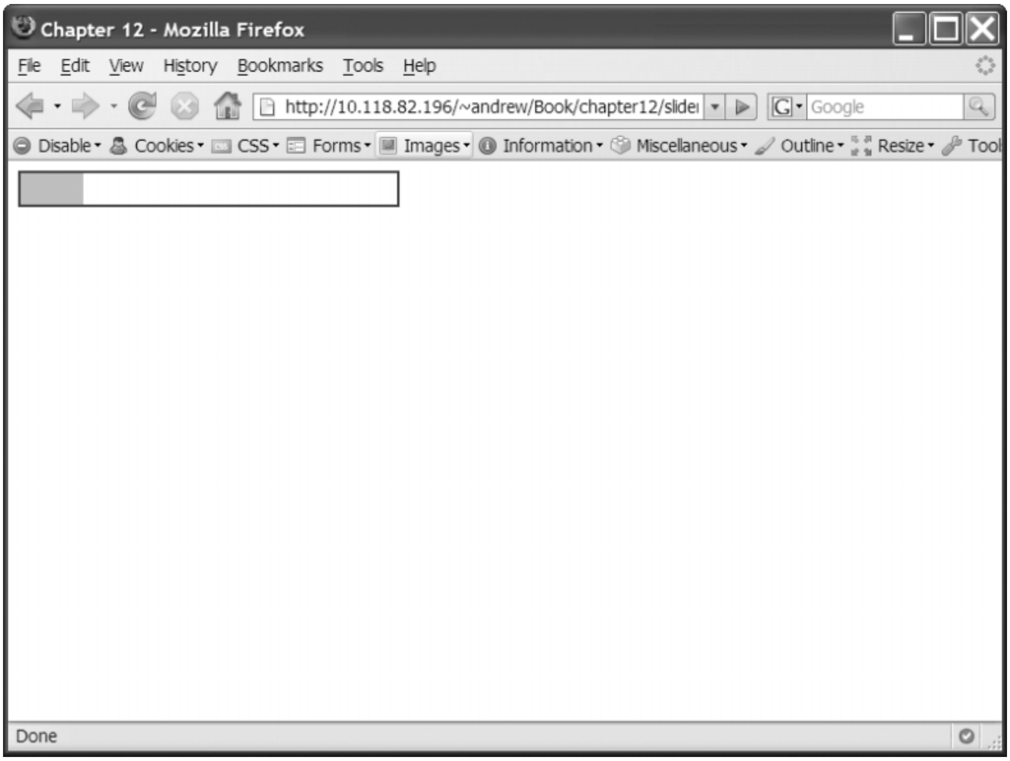

Chapter 12 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://10.118.82.196/~andrew/Book/chapter12/slider
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tool
Done

Figure 12-9. *The slider with draggable handle*

In other words, the handle knows the dimensions of both itself and its parent (the track); it knows to stop whenever its left edge hits the left edge of the track *or* its right edge hits the right edge of the track.

We can add more polish to this slider with some of `Control.Slider`'s options.

Common Options

A slider is used to set a value within a given range—to map a pseudo-analog control to a digital value. If your slider is 300 pixels long and a valid value is any integer from 0 to 11, then each pixel in the slider will map to a value between 0 and 11.

Many of the following options control the relationship between the visual representation of the slider and the internal range of values to which it maps.

Here are the options you'll use most often to configure your slider:

- `axis` controls the slider's direction of movement. It can be set to `horizontal` (the default) or `vertical`.
- `range` allows you to set the minimum and maximum values for your slider. It expects an `ObjectRange`—the construct discussed in Chapter 3. For example, `$R(0, 11)` describes all integers between 0 and 11 (inclusive).
- Instead of using `range` to set an explicit minimum and maximum, you can use `increment` to describe how pixels map to values. For example, setting `increment` to `1` means that the value will change by 1 for every pixel moved.
- Finally, you can use `values` to specify, in array form, the exact values that will be allowed (e.g., `[1, 2, 5, 10, 20]`). The slider handle will snap to those points on the track that represent these values.

Callbacks

Naturally, a slider is useless without callbacks to hook into. `Control.Slider` provides two callbacks:

- The `onSlide` callback is called continuously as the handle is dragged along the track.
- The `onChange` callback is called when the user *finishes* moving the handle and lifts the mouse button.

Both callbacks take the slider's current value as a first argument.

Summary

You learned about three specific sorts of controls in this chapter, but in the process you also saw several good examples of how JavaScript behaviors should be encapsulated. Notice how each of these examples involves declaring an instance of a class with a particular element as the "base" of the control.

In Chapter 13, we'll try to spot more patterns like these and glean some development principles and best practices.

CHAPTER 13

Prototype As a Platform

Prototype's features exemplify the functionality that distinguishes frameworks from libraries. It provides more than just shortcuts—it gives you new ways to structure your code.

In this chapter, we'll look at some of these tactics and patterns. We'll move beyond an explanation of what the framework does and into higher-level strategies for solving problems. Some of these are specific code patterns to simplify common tasks; others make code more modular and adaptable.

Using Code Patterns

A script written atop Prototype has a particular style (to call it *flair* would perhaps be overindulgent). It's peppered with the time-saving patterns and terse syntactic shortcuts that are Prototype's trademark.

I'm calling these *code patterns*, but please don't treat them as copy-and-paste sections of code. They're more like recipes; use them as a guide, but feel free to modify an ingredient or two as you see fit.

Staying DRY with Inheritance and Mixins

Prototype's class-based inheritance model lets you build a deep inheritance tree. Subclasses can call all the methods of their parents, even those that have been overridden.

Prototype itself uses inheritance with the `Ajax` classes. The simplest of the three, `Ajax.Request`, serves as a superclass for both `Ajax.Updater` and `Ajax.PeriodicalUpdater`. script.aculo.us uses inheritance even more liberally. For instance, all core effects inherit from an abstract class called `Effect.Base`. Any Prototype or script.aculo.us class can be subclassed by the user and customized.

Inheritance is a simple solution for code sharing, but it isn't always the best solution. Sometimes several classes need to share code but don't lend themselves to any sort of hierarchical relationship.

That's where mixins come in. *Mixins* are sets of methods that can be added to any class, independent of any sort of inheritance. A class can have only one parent class—multiple inheritance is not supported—but it can have any number of mixins.

Prototype uses one mixin that you'll recognize instantly: `Enumerable`. `Enumerable` contains a host of methods that are designed for working with collections of things. In Prototype, mixins are simply ordinary objects:

```
var Enumerable = {
  each: function() { /* ... */ },
  findAll: function() { /* ... */ },
  // ... and so on
};
```

We can use mixins in two ways:

- We can pass them to `Class.create`. The arguments we give to `Class.create` are simply groups of methods that the class will implement. The order of the arguments is important; later methods override earlier methods.

- We can use `Class#addMethods` to add these methods after initial declaration.

```
var Foo = Class.create({
  initialize: function() {
    console.log("Foo#Base called.");
  },

  /* implement _each so we can use Enumerable */
  _each: function(iterator) {
    $w('foo bar baz').each(iterator);
  }
});

// mixing in Enumerable after declaration
Foo.addMethods(Enumerable);

// mixing in Enumerable at declaration time
var Bar = Class.create(Enumerable, {
  initialize: function() {
    console.log("Bar#Base called.");
  },
  /* implement _each so we can use Enumerable */
  _each: function(iterator) {
    $w('foo bar baz').each(iterator);
  }
});
```

The difference between a mixin and a class is simple: mixins can't be instantiated. They're morsels of code that are meaningless on their own but quite powerful when used in the right context.

Enough about `Enumerable`. Let's look at a couple of modules you can write yourself.

Example 1: Setting Default Options

Many classes you write will follow the argument pattern of Prototype/script.aculo.us: the last argument will be an object containing key/value pairs for configuration. Most classes' `initialize` methods have a line of code like this:

```
var Foo = Class.create({
  initialize: function(element, options) {
    this.element = $(element);
    this.options = Object.extend({
      duration: 1.0, color: '#fff', text: 'Saving...'
    }, options || {});
  }
});
```

Here we're starting with a default set of options and using `Object.extend` to copy over them with whatever options the user has set. We can extract this pattern into one that's both easier to grok *and* friendlier to inherit.

First, let's move the default options out of the constructor and into a more permanent location. We'll declare a new "constant" on the `Foo` class; it won't *technically* be a constant, but it'll have capital letters, which will make it look important. That's close enough.

```
Foo.DEFAULT_OPTIONS = {
  duration: 1.0,
  color:    '#fff',
  text:     'Saving...'
};
```

Now, let's create a mixin called `Configurable`. It'll contain code for working with options.

```
var Configurable = {
  setOptions: function(options) {
    // clone the defaults to get a fresh copy
    this.options = Object.clone(this.constructor.DEFAULT_OPTIONS);
    return Object.extend(this.options, options || {});
  }
};
```

To appreciate this code, you'll need to remember two things. First, observe how we clone the default options. Since objects are passed by reference, we want to duplicate the object first, or else we'll end up modifying the original object in place. And, as the imposing capital letters suggest, `DEFAULT_OPTIONS` is meant to be a constant.

Second, remember that the `constructor` property always refers to an instance's class. So an instance of `Foo` will have a `constructor` property that references `Foo`. This way we're able to reference `Foo.DEFAULT_OPTIONS` without using `Foo` by name.

Now we can simplify the code in our `Foo` class:

```
var Foo = Class.create({
  initialize: function(element, options) {
    this.element = $(element);
    this.setOptions(options);
  }
});
```

Now, if you've been counting lines of code, you'll have discovered that we wrote about eight lines in order to eliminate about two. So far we're in the hole. But let's take `Configurable` one step further by allowing default options to *inherit*:

```
var Configurable = {
  setOptions: function(options) {
    this.options = {};
    var constructor = this.constructor;
    if (constructor.superclass) {
      // build the inheritance chain
      var chain = [], klass = constructor;
      while (klass = klass.superclass) chain.push(klass);
      chain = chain.reverse();
      for (var i = 0, len = chain.length; i < len; i++)
        Object.extend(this.options, klass.DEFAULT_OPTIONS || {});
    }
    Object.extend(this.options, constructor.DEFAULT_OPTIONS);
    return Object.extend(this.options, options || {});
  }
};
```

OK, this one was a *big* change. Let's walk through it:

1. First, we set `this.options` to an empty object.

2. We check to see whether our constructor has a superclass. (Remember that magical `superclass` property I told you about? It has a purpose!) If it inherits from nothing, our task is simple—we extend the default options onto the empty object, we extend our *custom* options, and we're done.

3. If there *is* a superclass, however, we must do something a bit more complex. In short, we trace the inheritance chain from superclass to superclass until we've collected all of the class's ancestors, in order from nearest to furthest. This approach works no matter how long the inheritance chain is. We collect them by pushing each one into an array as we visit it.

4. When there are no more superclasses, we stop. Then we *reverse* the array so that the furthest ancestor is at the beginning.

5. Next, we loop through that array, checking for the existence of a `DEFAULT_OPTIONS` property. Any that exist get extended onto `this.options`, which started out as an empty object but is now accumulating options each time through the loop.

6. When we're done with this loop, `this.options` has inherited all the *ancestors'* default options. Now we copy over the default options of the *current* class, copy over our custom options, and we're done.

Still with me? Think about how cool this is: default options now inherit. I can instantiate `Grandson` and have it inherit all the default options of `Son`, `Father`, `Grandfather`, `GreatGrandfather`, and so on. If two classes in this chain have different defaults for an option, the "younger" class wins.

You might balk at the amount of code we just wrote, but think of it as a trade-off. Setting options the old way is slightly ugly *every time we do it*. Setting them with the `Configurable` mixin is *really* ugly, but we need to write it only *once*!

So much of browser-based JavaScript involves capturing this ugliness and hiding it somewhere you'll never look. Mixins are perfect for this task.

Example 2: Keeping Track of Instances

Many of the classes we've written are meant to envelop one element in the DOM. This is the element that we typically pass in as the first argument—the element we call `this.element`.

There should be an easy way to associate an element and the instance of a class that *centers* on said element. One strategy would be to add a property to the element itself:

```
Widget.Foo = Class.create({
  initialize: function(element, options) {
    this.element = $(element);
    // store this instance for later
    this.element._fooInstance = this;
  }
});
```

This approach works, but at a cost: we've just introduced a memory leak into our application. Internet Explorer 6 has infamous problems with its JavaScript garbage collection (how it reclaims memory from stuff that's no longer needed): it gets confused when there are *circular references* between a DOM node and a JavaScript object. The `element` property of my instance refers to a DOM node, and that node's `_fooInstance` property refers back to the instance.

So in Internet Explorer 6, neither of these objects will be garbage collected—even if the node gets removed from the document, and even if the page is reloaded. They'll continue to reside in memory until the browser is restarted.

Memory leaks are insidious and can be very, very hard to sniff out. But we can avoid a great many of them if we follow a simple rule: *only primitives should be stored as custom properties of DOM objects*. This means strings, numbers, and Booleans are OK; they're all passed by *value*, so there's no chance of a circular reference.

So we've settled that. But how else can we associate our instance to our element? One approach is to add a property to the class itself. What if we create a lookup table for `Widget.Foo`'s instances *on* `Widget.Foo` itself?

```
Widget.Foo.instances = {};
```

Then we'll store the instances as key/value pairs. The key can be the element's ID—it's unique to the page, after all.

```
Widget.Foo = Class.create({
  initialize: function(element, options) {
    this.element = $(element);
    Widget.Foo[element.id] = this;
  }
});
```

Brilliant! Now, to grab a class instance from the associated element, we can use its ID:

```
var someInstance = Widget.Foo.instances[someElement.id];
```

We could stop right here and be happy with ourselves. But let's consider some edge cases first:

What if the element doesn't have an ID? Then our key will be `null`. To ensure that the element has an ID, we can use `Element#identify`. The method returns the element's ID if it already exists; if not, it assigns the element an arbitrary ID and returns it to us.

What if we don't know the name of the class? In order to move this code into a mixin, we'll have to remove any explicit references to the class's name. Luckily, we've already got the answer to this one: the instance's `constructor` property, which points back to the class itself.

What if that element already has an instance and another one gets created? For this case, we'll *assume* that only one instance per element is needed. When a new one gets declared, it would be nice if we cleaned up the old one somehow.

First, let's make a mixin called `Trackable`. It'll contain the code for keeping track of a class's instances. Let's also create a `register` method, which should be the only one we need for this exercise. It'll add the instance to the lookup table.

```
var Trackable = {
  register: function() {
  }
};
```

Now we'll solve our problems one by one. First, let's grab the element's ID. If the class doesn't have an `element` property, we'll simply return `false`. (You may choose to throw an exception instead; just make sure you handle this case one way or another.)

```
var Trackable = {
  register: function() {
    if (!this.element) return false;
    var id = this.element.identify();
  }
};
```

Next, we'll use the `constructor` property to reach the class itself. This way we don't have to call it by name. We'll also create the `instances` property if it doesn't already exist.

```
var Trackable = {
  register: function() {
    if (!this.element) return false;
    var id = this.element.identify();

    var c = this.constructor;
    if (!c.instances) c.instances = {};

    c.instances[id] = this;
  }
};
```

Now we'll address that last edge case. If a class needs some sort of cleanup before it gets removed, we'll have to rely on the class itself to tell us how. So let's adopt a convention: assume the cleanup "instructions" are contained in a method called `destroy`. This method might remove event listeners, detach some nodes from the DOM, or stop any timers set using `setTimeout` or `setInterval`.

This method will handle cleanup when we need to replace an instance that's already in the table. We can check the instance to be replaced to see whether it has a `destroy` method; if so, we'll call it before replacing the instance in the lookup table.

```
var Trackable = {
  register: function() {
    if (!this.element) return false;
    var id = this.element.identify();

    var c = this.constructor;
    if (!c.instances) c.instances = {};

    if (c.instances[id] && c.instances[id].destroy)
      c.instances[id].destroy();

    c.instances[id] = this;
  }
};
```

And we're done. Our new mixin is small but useful. And including it in a class is as simple as passing it into `Class.create`. The only other thing to remember is to call the `register` method sometime after assigning the `element` property.

```
Widget.Foo = Class.create(Trackable, {
  initialize: function(element, options) {
    this.element = $(element);
    this.register();

    this.addObservers();
  },

  addObservers: function() {
    // We store references to this bound function so that we can remove them
    // later on.
    this.observers = {
      mouseOver: this.mouseOver.bind(this),
      mouseOut:  this.mouseOut.bind(this);
    }

    this.element.observe("mouseover", this.observers.mouseOver);
    this.element.observe("mouseout",  this.observers.mouseOut);
  },

  destroy: function() {
    this.element.stopObserving("mouseover", this.observers.mouseOver);
    this.element.stopObserving("mouseout",  this.observers.mouseOut);
  }
});
```

The mixin takes care of the rest. Write it once and it'll be useful for the remainder of your scripting career.

Solving Browser Compatibility Problems: To Sniff or Not to Sniff?

So, if some browsers are more ornery than others, how can we tell which is which? The obvious approach would be *sniffing*—checking the browser's user-agent string. In JavaScript, this string lives at `navigator.userAgent`. Looking for telltale text (e.g., "MSIE" for Internet Explorer or "AppleWebKit" for Safari) usually lets us identify the specific browser being used, even down to the version number.

Browser sniffing is problematic, though—the sort of thing you'd get dirty looks for at web design meetups and tech conferences. Among the biggest problems is that there are more browsers on earth than the average web developer knows about, and when doing browser sniffing, it's too easy to write code that leaves some poor saps out in the cold.

Also troublesome is that browsers have an incentive to imitate one another in their user-agent strings, thereby diluting the value of the information. For years, Opera (which supports a number of Internet Explorer's proprietary features) masqueraded as Internet Explorer in its user-agent string. Better to do so than to be arbitrarily shut out of a site that would almost certainly work in your browser.

Finally, though, the problem with browser sniffing is arguably one of coding philosophy: is it the right question to ask? Quite often we need to distinguish between browsers because of their varying levels of support for certain features. The real question, then, is "Do you support feature X?" instead of "Which browser are you?"

This debate is oddly complex. It's important because we need to assess what a user's browser is capable of. But before we go further, we ought to make a distinction between *capabilities* and *quirks*.

Capabilities Support

Capabilities are things that some browsers support and others don't. DOM Level 2 Events is a capability; Firefox supports it, but Internet Explorer does not (as of version 7). DOM Level 3 XPath is a capability; Safari 3 supports it, but Safari 2 does not.

Other capabilities are supported by all modern browsers, so we take them for granted. All modern browsers support `document.getElementById` (part of DOM Level 1 Core), but once upon a time this wasn't true. Nowadays only the most paranoid of DOM scripters tests for this function before using it.

Capabilities are not specific to browsers. They're nearly always supported by specifications (from the W3C or the WHATWG, for instance) and will presumably be supported by all browsers eventually.

To write code that relies on capabilities, then, you ought to be singularly concerned with the features a browser claims to support, not the browser's name. Since functions are objects in JavaScript, we can test for their presence in a conditional:

```
// document.evaluate is an XPath function
if (document.evaluate) {
  // fetch something via XPath
} else {
  // fetch something the slower, more compatible way
}
```

Here we're testing whether the `document.evaluate` function exists. If so, the conditional evaluates to `true`, and we reap the benefits of lightning-fast DOM traversal. If not, the conditional evaluates to `false`, and we reluctantly traverse the DOM using slower methods.

Testing for capabilities makes our code future-proof. If a future version of Internet Explorer supports XPath, we don't have to change our detection code, because we're testing for the *feature*, not the *browser*.

Therefore, it's a far better idea to *test* for capabilities than to *infer* them based on the name of a browser. It's not the meaningless distinction of a pedant. Code written with a capability-based mindset will be hardier and of a higher quality.

Quirks and Other Non-Features

There's a dark side, though. JavaScript developers also have to deal with quirks. A *quirk* is a polite term for a *bug*—an unintended deviation from the standard behavior. Internet Explorer's aforementioned memory leaks are a quirk. Internet Explorer 6, a browser that many web users still run today, has been around since 2001, enough time to find all sorts of bizarre bugs in rendering and scripting.

To be clear, though, *all* browsers have quirks (some more than others, to be sure). But quirks are different from capabilities. They're nearly *always* specific to one browser; two different browsers won't have the same bugs.

I wish I could present some sort of uniform strategy for dealing with quirks, but they're too varied to gather in one place. Let's look at a few examples.

Quirk Example 1: Internet Explorer and Comment Nodes

The DOM specs treat HTML/XML comment nodes (`<!-- like these -->`) differently from, say, element nodes. Comments have their own node type, just like text nodes or attribute nodes.

In Internet Explorer, comment nodes are treated as element nodes with a tag name of `!`. They report a `nodeType` of `1`, just like an element would. Calling `document.getElementsByTagName('*')` will, alongside the element nodes you'd expect, return any comments you've declared in the body.

This is incorrect, to put it mildly. More vividly, it's the sort of bug that would make a developer embed her keyboard into her own forehead if she weren't aware of it and had encountered it on her own.

So how do we work around quirks? It depends. One strategy is to treat them just like capabilities—see if you can reproduce the bug, and then set some sort of flag if you can:

```
var thinksCommentsAreElements = false;
if (document.createElement('!').nodeType === 1) {
  thinksCommentsAreElements = true;
}
```

Once you've set this flag, you can use it inside your own functions to give extra logic to Internet Explorer.

This approach has the same upsides of capability detection: instead of blindly assuming that all versions of Internet Explorer exhibit this quirk, we find out for sure. If Internet Explorer 8 fixes this bug, it avoids the workaround altogether.

Quirk Example 2: Firefox and Ajax

Versions of Firefox prior to 1.5 exhibit a behavior that can throw a wrench into the Ajax gears. An affected browser will, in an Ajax context, sometimes give the wrong `Content-Length` of an HTTP POST body—thereby flummoxing servers that find a line feed after the request was supposed to have ended.

The workaround is simple enough: force a `Connection: close` header so that the server knows not to keep listening (in other words, tell the server that the line feed can be ignored). But figuring out when the workaround is needed turns out to be very, very ugly.

Here are a few lines from the Prototype source code. We've dealt with this bug so that you won't have to, but here's the workaround:

```
/* Force "Connection: close" for older Mozilla browsers to work
 * around a bug where XMLHttpRequest sends an incorrect
 * Content-length header. See Mozilla Bugzilla #246651.
 */
if (this.transport.overrideMimeType &&
    (navigator.userAgent.match(/Gecko\/(\d{4})/) ||
     [0,2005])[1] < 2005)
      headers['Connection'] = 'close';
```

I hate to bring this code out for exhibition. It's like bringing your angst-ridden teenage poetry to a first date. But unlike the melodramatic sonnets you wrote after your junior prom, this code is quite purposeful.

We can't treat this quirk like a capability because we can't *test* for it. To test for it, we'd need to send out an Ajax request while the script initializes. Likewise, we can't apply the workaround to all browsers, because we'd interfere with use cases where the connection should not be closed (like HTTP keep-alive connections).

So we must search the browser's user-agent string to see whether it uses the affected engine—then we must look at the release year of the engine to figure out whether it's old enough to be affected.

Of course, Prototype fixes this so that you, as a developer, need not worry about it. And the quirks you'll encounter probably won't be so tenacious. But eventually, if you write enough code, you'll need to do some occasional browser sniffing. Do it, apologize to yourself, and move on.

If it makes you queasy, good! It *should* make you queasy. That'll stop you from using it more often than you ought to. And the fact that you're ashamed of your old poetry simply affirms the sophistication of your adult tastes.

If You Must . . .

So if you've got no other options . . . yes, it's OK to sniff. But you've got to do it right. Perform the following steps or else suffer the wrath of the browser gods.

Get a Second Opinion

First, assuage your guilt. Find a friend or coworker—across the hall, on Instant Messenger, in IRC—and summarize your dilemma. Often your consultant will suggest an approach you hadn't thought of. But if he *can't* think of a better way, you'll feel more secure in your decision.

Take Notes

Write a comment that explains the problem and why you've got to sniff. Put it as close as possible to the offending line of code. Be verbose. This is for *your own* benefit: six months from now you won't remember why you wrote that code the way you did, so think of it as a message to Future You.

Walk a Straight Code Path

Most importantly, write code without unnecessary kinks and contortions. If Internet Explorer needs one thing, but all other browsers need another, write your function to handle the other browsers. If one approach uses the DOM standard and the other uses a proprietary Internet Explorer method, write your function to use the DOM standard—then, at the beginning of the function, send Internet Explorer into a different function to handle the weird case.

The purpose is to avoid the "black holes" that come from excessive sniffing. Consider this code:

```
function handleFoo() {
  if (navigator.userAgent.match(/Gecko\//))
    return handleFoo_Firefox();
  if (navigator.userAgent.match(/MSIE/))
    return handleFoo_MSIE();
}
```

Safari, Opera, OmniWeb, iCab, and browsers far off the beaten path will fall straight through this function—because your code never "caught" them. Again, you're not concerned with what the browser *is*; you're concerned with what it says it *can do*. You can't possibly test in every browser on earth, so embrace standards as a compromise: if a browser follows web standards, it ought to be able to read your code, even if you didn't code with it in mind.

Holding Up Your End of the Bargain

Even though I'm giving you permission to write "dirty" code once in a while, I mean to open only the tiniest of loopholes. The early days of JavaScript taught us that bad things happen when developers abuse user-agent sniffing. I'd recommend against it altogether if it weren't for the handful of edge cases that require sniffing.

In other words, when we as developers sniff unnecessarily, it's our fault. When we discover situations in which sniffing is the *only* option, it's the *browser maker's* fault. So think of developing with web standards as a social contract between developers and vendors: do your part, and we'll do ours. Make your scripting environment behave predictably and rationally, and we won't need to take drastic steps to code around bugs.

Making and Sharing a Library

Written something useful? Something you think would be valuable to others? All the major JavaScript toolkits have an ecosystem of plug-ins and add-ons. If you've created a script that makes your life easier, there's a good chance it will make someone else's life easier, too.

Maybe you've done something big, like a really cool UI widget or client-side charting. Maybe it's big enough to warrant a Google Code project and a release schedule. Or maybe you've written a way to do simple input validation in 30 lines of code and just want to put it on the Web, as is, so that others can learn from it.

Here are some best practices for releasing your add-on. Most of them relate to the difficulty of writing code that satisfies both *your* needs and the needs of the public.

Make Your Code Abstract

The hardest thing about turning *your* code into *public* code is handling abstraction. When you first wrote it, you might have embraced the conventions of your own circumstances in order to simplify things; now you've got to go back and handle scenarios you didn't foresee.

Do One Thing Well (or Else Go Modular)

Don't try to be a Swiss Army knife. Code that does one thing well is easier to understand, easier to set up, and faster for the end user to download. It's one thing to write a 5 KB script that depends on Prototype; it's another thing to write 80 KB of JavaScript that depends on Prototype, most of which John Q. Developer won't even *need*.

I should clarify: it's fine to do *many* things well, as long as those things are not interdependent. If your script does three unrelated things, break it up into three unrelated scripts. Bundle them together if you like, but don't require all three unless you've got a *very* good reason. Notice that script.aculo.us is modular: you don't have to load all the UI stuff if the effects are all you want.

Embrace Convention

With apologies to Jakob Nielsen, a developer will spend far more time working with other people's code than with *your* code. Each major framework has a distinct coding style that can be seen in the structure of its API, the order of arguments, and even its *code formatting*. (Spaces, not tabs! No—tabs, not spaces!)

Follow those conventions! Make your add-on feel like a part of Prototype. Design your classes to take an `options` argument. Embrace the patterns, coding style, and lexicon. Nobody reads the documentation (at least not at first), so even the smallest details of code writing can help a person intuit how to use your code.

Make Things Configurable

All of the classes in Prototype and script.aculo.us share a powerful design principle: they're able to be exhaustively configurable *and* dead-simple to use at the same time.

Is there anything about your code that someone might need to tailor to his needs? Does your widget set a background color? Make it a configurable option. Does your date picker control support arbitrary date formats (like DD/MM/YYYY, which is the most common format outside of North America)? If not, write it now; someone will ask for that feature. Does your widget fade out over a span of 0.5 seconds? Someone will argue *passionately* for it to be 1 second instead.

That takes care of the "exhaustively configurable" part. To make your add-on dead-simple to use, take care to *hide* this complexity below the surface until it's needed. Make as many options as you like, but give an *intelligent default* for each one.

Make sure the `options` argument can be omitted together. They're called options because they're *optional*; if a particular parameter can't have a default and can't be omitted, move it out of the `options` object and into a positional argument.

Add Hooks

Often your add-on will be an almost-but-not-quite-perfect solution to someone's problem. "If only the tooltips *stayed* in the DOM tree after they fade out!" "This would be *perfect* if the widget could follow my cursor around."

These requests are usually too complex or obscure than can be solved with extra configuration, but they're important nonetheless. Don't bring your fellow developer 95 percent of the way to his destination, and then leave him stranded in the mysterious town of Doesn't-Quite-Do-What-I-Want. Give him the tools to travel that final 5 percent on his own.

There are two prevailing ways to empower the user to make those tweaks: callbacks and custom events.

We've seen callbacks before in nearly all the script.aculo.us controls. They're functions the user can define that will get called at a certain point in the control's operation. They can be called at certain points in your add-on's life cycle.

For instance, we can leverage the `Configurable` mixin we wrote earlier to set up default callbacks that are "empty," letting the user override them if necessary:

```
var Widget = Class.create(Configurable, {
  initialize: function(element, options) {
    this.element = $(element);
    this.setOptions(options);

    this.options.onCreate();
  }
});

Widget.DEFAULT_OPTIONS = {
  onCreate: Prototype.emptyFunction
};

// using the callback
var someWidget = new Widget('some_element', {
  onCreate: function() { console.log('creating widget'); }
});
```

Notice the reference to Prototype.emptyFunction: it's a function that does nothing. Instead of checking whether the onCreate option exists, we include it in the default options; this way, whether the user specifies it or not, the onCreate property refers to a function.

Custom events are a way to approach this problem from a different angle. Imagine a tooltip:hidden event that a developer can listen for in order to apply her own logic for dealing with hidden tooltips. Firing a custom event is a lot like invoking a callback:

```
var Widget = Class.create({
  initialize: function(element, options) {
    this.element = $(element);
    // fire an event and pass this instance as a property of
    // the event object
    this.element.fire("widget:created", { widget: this });
  }
});

// using the custom event
$('some_element').observe('widget:created', function() {
  console.log("widget created");
});

// or observe document-wide:
document.observe("widget:created", function() {
  console.log("widget created");
});

var someWidget = new Widget('some_element');
```

They're a little more work to set up, but they're also more robust. Notice that we can listen for the event on the element itself *or* on any one of its parent nodes, all the way up to document. Callbacks don't provide that kind of flexibility, nor do they allow you to easily attach more than one listener.

Whichever way you go, be sure to *document your hooks*. They won't get used if nobody knows they exist. Be clear about when the hook fires and what information accompanies it—parameters passed to the callback or properties attached to the event object.

Summary

In this chapter, we explored a number of ways to turn code that's useful to *you* into code that's useful to *others* as well. Conciseness, modularity, documentation, and extensibility are the only things separating the scripts you write from libraries like Prototype and script.aculo.us.

Have you got an idea for a script? A UI control, a clever use of Ajax, or even a way to automate a repetitive task? Write it! Get your code out into the wild! Focusing on the polish needed to write scripts for the public will make you a better developer.

Index